Architectures intergicielles pour la tolérance aux fautes et le consensus

Khaled Barbaria

▶ To cite this version:

Khaled Barbaria. Architectures intergicielles pour la tolérance aux fautes et le consensus. domain_other. Télécom ParisTech, 2008. English. NNT : . pastel-00004308

HAL Id: pastel-00004308
https://pastel.hal.science/pastel-00004308

Submitted on 9 Jan 2009

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

Architectures intergicielles pour la tolérance aux fautes et le consensus

Thèse

présentée et soutenue publiquement le 15 septembre 2008

pour l'obtention du grade de

Docteur de l'École Nationale Supérieure des Télécommunications

spécialité : informatique et réseaux

par

Khaled Barbaria

Composition du jury

Président :	**Elie Najm**, Professeur à l'École Nationale Supérieure des Télécommunications
Rapporteurs :	**Yvon Kermarrec**, Professeur à l'École Nationale Supérieure des Télécommunications de Bretagne **Lionel Seinturier**, Professeur à l'Université des Sciences et Technologies de Lille
Examinateurs :	**Olivier Marin**, Maître de conférences à l'Université Pierre et Marie Curie (Paris VI) **Frank Singhoff**, Maître de conférences à L'université de Bretagne Occidentale
Directeur de thèse :	**Laurent Pautet**, Professeur à l'École Nationale Supérieure des Télécommunications

GET-Télécom Paris, département INFRES **LTCI-UMR 5141 CNRS**

Mis en page avec la classe thloria.

À ma mère
À l'âme de mon père
À toute ma famille

Remerciements

Tout d'abord, je rends grâce à dieu pour m'avoir permis d'arriver au bout de ce travail.

Je tiens à remercier tous les membres de ma famille pour m'avoir supporté et aidé tout au long des années de cette thèse. Merci à ma mère Alia, ma femme Ines, ma grand mère Habiba (Bellia), ma soeur Wiem et mon frère Sami. Merci à mes oncles et tantes : Abel Aziz, Mondher, Mohamed Salah (Hammadi), Chokri, Salwa, Basma, Sonia, Boutheina, Beya, Jamila, Assia.

Un grand merci à Toufik Den Dali pour ses nombreux et précieux conseils.

Merci à Laurent Pautet pour avoir encadré cette thèse. Son experience et ses conseils m'ont beaucoup aidé.

Merci aux différents enseignants chercheurs qui m'ont honoré en faisant partie du Jury de cette thèse. Merci à Yvon Kermarrec et à Lionel Seinturier pour avoir accepté d'être rapporteurs de ce travail. Merci à Olivier Marin et à Frank Singhoff pour avoir accepté d'être exminateurs de cette thèse. Merci à Elie Najm pour avoir accepté d'être président de mon jury de thèse.

Je tiens à remercier les différents collègues de l'ENST avec qui j'ai partagé une expérience enrichissante tant sur le plan scientifique et technique que sur le plan humain : Raja Chiky, Ahmed Fadhlallah, Bilel Guenni, Irfan Hamid, Jérome Hugues, Zakia Kazi-Aoul, Isabelle Perseil, Thomas Vergnaud et Bechir Zalila.

Je n'oublie pas les les joyeux moments que j'ai pu partager avec plusieurs amis. Merci à Akram Aidani, Houssemeddine Bedoui, Mohamed Habib Ben Abid, Taha Dridi, Walid Dridi, Haythem Gadacha, Mohamed Chedly Ghedira et Mohamed Khelij.

Résumé

Le succès des intergiciels dans le cadre du développement de systèmes d'information "généralistes" comme les applications Web, encourage leur utilisation pour le développement d'autres applications plus spécifiques, comme les applications temps réel ou même certaines applications critiques.

Ces applications présentent des contraintes et des exigences en qualité de service généralement difficiles à satisfaire. Par conséquent, elles présentent de grand défis à relever par les technologies intergicielles qui doivent satisfaire simultanément à plusieurs exigences et contraintes.

Nous partons d'une architecture intergicielle dite schizophrène ayant des propriétés de généricité et de configuration. Cette architecture est renforcée pour supporter deux catégories de services pour la tolérance aux fautes et le consensus. La conservation des propriétés de l'architecture de base ainsi que le respect des contraintes posées par les applications critiques et sûres de fonctionnement sont les principaux objectifs de nos propositions.

Les principes et les propriétés de l'architecture schizophrène sont détaillés. Ensuite, nous menons des études approfondies de la théorie de la tolérance aux fautes et du consensus ainsi que de la norme FT CORBA. Ces études nous permettent de généraliser les différents concepts et d'isoler les différentes abstractions utiles afin de proposer deux architectures pour un service de tolérance aux fautes compatible avec la norme FT CORBA et pour un service générique de consensus. Nous montrons que la conception de ces services maximise leur configurabilité.

Après les propositions d'architectures, nous décrivons la réalisation effective de ces deux services. Nous nous basons sur PolyORB, un integriciel développé à l'ENST servant à l'expérimentation des propositions autour de l'architecture de schizophrène. Des scénarios de test et des mesures de performances complètent notre étude et valident nos différentes propositions.

Mots-clés: Ada, Consensus, Intergiciels, Temps réel, Tolérance aux fautes

Abstract

Middleware technologies have been widely adopted to design and develop general purpose distributed applications. The efficiency of these technologies encourages to study their compatibility with the stringent requirements of some domain-specific real time and critical applications. These requirements are hard to achieve and may be incompatible with the objectives of reusability and cost reduction.

We enhance a middleware architecture said “schizophrenic”, that enables for genericity and extreme reusability with services for consensus and fault tolerance. The integration of these services follows two principles. On one hand, we conserve the properties of the original architecture. On the other hand, we capture and respect the the constraints and requirements that come with critical and dependable applications.

First, the principles and properties of the schizophrenic architecture are detailed. Then, the theory of fault tolerance and of consensus as well as the FT CORBA standard are presented. These theoretical studies allowed us to isolate the useful abstractions and concepts to design services for fault tolerance and consensus. Finally, we present the realisation of these services. Our proposal is based on PolyORB a middleware developed at ENST. Test scenarios and performance measures complete our study and validate our propositions.

Keywords: Ada, Consensus, Middleware, Fault Tolerance, Real Time

Table des matières

Introduction **1**

1 Position du problème . . . 1

2 Objectifs et démarche . . . 2

3 Plan . . . 3

Partie I État de l'art **5**

1 Cadre général, notions de base et problématique **7**

1.1 Cadre général . . . 8

1.2 Intergiciels . . . 9

1.2.1 Définition et généralités . . . 9

1.2.2 Architecture de l'intergiciel . . . 10

1.2.3 Apports des intergiciels . . . 10

1.3 Tolérance aux fautes . . . 11

1.3.1 Définitions et notions de base . . . 11

1.3.2 Besoins en tolérance aux fautes . . . 13

1.3.2.1 Attributs de la tolérance aux fautes . . . 13

1.3.2.2 Domaines d'applications . . . 14

1.3.3 Principes de la tolérance aux fautes . . . 15

1.3.3.1 Redondance : base de la tolérance aux fautes . . . 15

1.3.3.2 Réplication . . . 16

1.3.4 Conclusion . . . 18

1.4 Consensus . . . 18

1.4.1 Définition . . . 18

1.4.2 Besoins . . . 19

1.4.2.1 Utilisation explicite . . . 19

1.4.2.2 Utilisation implicite . . . 19

1.4.2.3 Support de la tolérance aux fautes . . . 19

1.4.3 Résultats théoriques 20
1.4.3.1 Modèles de défaillances 20
1.4.3.2 Modèles de calcul et Synchronisme 21
1.4.3.3 Problème du consensus dans les systèmes asynchrones 22
1.4.3.4 Détection des défaillances dans les systèmes asynchrones . . 22
1.4.4 Conclusion 23
1.5 Production d'applications critiques distribuées 23
1.5.1 Architecture 24
1.5.2 Utilisation des langages de programmation de haut niveau 25
1.5.3 Validation 26
1.5.4 Conclusion 27
1.6 Objectifs et axes de recherche 28
1.6.1 Objectifs 28
1.6.2 Propositions et contributions 29

2 Intergiciels, tolérance aux fautes et consensus 31
2.1 Introduction 31
2.2 Intergiciels 32
2.2.1 Modèles de distribution 32
2.2.1.1 Intergiciels orientés messages 32
2.2.1.2 Appels de procédures distants 33
2.2.1.3 Objets distribués 34
2.2.1.4 Conclusion 36
2.2.2 Propriétés des intergiciels 37
2.2.2.1 Adaptation et réduction des coûts 37
2.2.2.2 Propriétés Comportementales 39
2.2.3 Synthèse 41
2.3 Tolérance aux fautes dans les systèmes distribués 41
2.3.1 Architectures et intergiciels tolérants aux fautes 41
2.3.1.1 Intergiciels propriétaires 42
2.3.1.2 Architectures génériques 43
2.3.1.3 Intergiciels standards 45
2.3.1.4 Synthèse 46
2.3.2 Tolérance aux fautes et réplication dans CORBA 47
2.3.2.1 Architectures 47
2.3.2.2 Comportement temporel 48
2.3.2.3 Adaptation, configuration et modélisation 49

2.3.3 Synthèse 50
2.4 Support de l'abstraction du consensus par les intergiciels 50
2.4.1 Consensus, détection de défaillances : de la théorie à la pratique . . . 51
2.4.1.1 Consensus 51
2.4.1.2 Détection de défaillances 52
2.4.1.3 Conclusion 53
2.4.2 Patrons et architectures pour le consensus 53
2.4.2.1 `MAFTIA` 53
2.4.2.2 `Thema` 53
2.4.2.3 `Bast` 54
2.4.2.4 `OGS` 54
2.4.2.5 `CORE/SONiC` 54
2.4.2.6 `AspectIX` 55
2.4.3 Conclusion 55
2.5 Synthèse 56
2.5.1 Technologies intergicielles 56
2.5.2 Tolérance aux fautes 56
2.5.3 Consensus 56

Partie II Conception **57**

3 Conception et intégration d'un service de tolérance aux fautes dans une architecture schizophrène **59**
3.1 Introduction 59
3.2 Architecture schizophrène 60
3.2.1 Définitions, objectifs et principes 60
3.2.2 Personnalités applicatives et protocolaires 61
3.2.2.1 Personnalités applicatives 61
3.2.2.2 Personnalités protocolaires 61
3.2.3 Couche neutre et services canoniques de distribution 62
3.2.4 Avantages de l'architecture 64
3.2.4.1 Généricité 64
3.2.4.2 Configurabilité des services 64
3.2.4.3 Services fondamentaux 65
3.2.4.4 Vérification comportementale 65
3.2.5 Conclusion 65

3.3 Conception et intégration d'un service de tolérance aux fautes dans l'architecture schizophrène 66
3.3.1 Problématique 66
3.3.1.1 Architecture de `FT CORBA` 66
3.3.1.2 Besoins vis à vis de l'intergiciel 68
3.3.1.3 Conclusion 69
3.3.2 Mécanismes d'interception dans `CORBA` 70
3.3.3 Mise en oeuvre de la réplication à l'aide des intercepteurs 72
3.3.3.1 Architecture 72
3.3.3.2 Intercepteurs coté client 73
3.3.3.3 Intercepteurs coté serveur 73
3.3.3.4 Respect des principes de l'architecture schizophrène 74
3.3.3.5 Conclusion 75
3.3.4 Détection et notification des défaillances 75
3.3.4.1 Détection des défaillances 76
3.3.4.2 Notification des défaillances 77
3.3.4.3 Conclusion 77
3.3.5 Avantages de l'architecture 78
3.3.5.1 Indépendance des personnalités protocolaires 78
3.3.5.2 Support de la vérification formelle 78
3.3.5.3 Configurabilité de la tolérance aux fautes 79
3.4 Conclusion 79

4 Architecture d'un service générique de consensus **81**
4.1 Conception d'un service générique de consensus 82
4.1.1 Architecture générale 82
4.1.1.1 Choix des modèles de calcul et de défaillances 83
4.1.1.2 Architecture générale 83
4.1.1.3 Services intergiciels de base 84
4.1.1.4 Conclusion 85
4.1.2 Service du consensus 86
4.1.2.1 Patrons de conception "Pont" et "Stratégie" 86
4.1.2.2 Consensus 88
4.1.2.3 Détection des défaillances 89
4.1.2.4 Échanges des messages 90
4.1.2.5 Conclusion 91
4.1.3 Interactions entre les composants du service du consensus 92

4.1.3.1 Problématique 92
4.1.3.2 Architectures orientées évènements 94
4.1.3.3 Application de l'architecture orientée évènements 96
4.1.3.4 Conclusion 98
4.2 Intégration dans l'architecture schizophrène et cas d'applications 98
4.2.1 Groupe de participants 98
4.2.2 Intégration dans l'architecture schizophrène 100
4.2.2.1 Partitions et groupes de participants 101
4.2.2.2 Échanges des messages 101
4.2.3 Configuration de l'intergiciel 102
4.2.4 Application à la personnalité `CORBA` 104
4.3 Conclusion 105

Partie III Réalisation et expérimentation 107

5 Composants et services pour la tolérance aux fautes et le consensus 109
5.1 Introduction 109
5.2 `Ada` et `PolyORB` 110
5.2.1 Le langage `Ada` 110
5.2.2 Architecture de `PolyORB` 112
5.2.2.1 Présentation de `PolyORB` 112
5.2.2.2 Vue globale de l'architecture 112
5.2.2.3 Mise en oeuvre de la couche neutre 112
5.2.2.4 Services utilitaires 113
5.3 Réalisation d'un service de tolérance aux fautes 114
5.3.1 Impact sur l'architecture de `PolyORB` 114
5.3.2 Gestion des groupes d'objets 115
5.3.3 Fonctionnalités avancées de `GIOP` 117
5.3.4 Gestion de la réplication et détection des défaillances 119
5.3.4.1 `ReplicationManager` 119
5.3.4.2 Détection et notification des défaillances 119
5.3.5 Styles de réplication 121
5.3.5.1 Définition des intercepteurs 121
5.3.5.2 Mise en oeuvre des styles de réplication 123
5.3.6 Construction d'applications basées sur `FT CORBA` 124
5.3.6.1 Configuration des services dans `PolyORB` 124
5.3.6.2 Configuration des applications tolérantes aux fautes 124

5.3.6.3 Limites et perspectives 126
5.3.7 Conclusion 126
5.4 Réalisation d'un service générique de consensus 127
5.4.1 Composants du service du consensus 127
5.4.1.1 Projection sur Ada 127
5.4.1.2 Composants de consensus et de détection de défaillances . . 129
5.4.1.3 Gestion des messages 130
5.4.2 Gestion des évènements 132
5.4.2.1 Production 132
5.4.2.2 Consommation 133
5.4.2.3 Conclusion 133
5.4.3 Construction d'applications basées sur le consensus 133
5.4.3.1 Transport de données 134
5.4.3.2 Configurations locale et globale d'une partition 134
5.4.3.3 Assemblage du service du consensus 136
5.4.3.4 Exemple 1 : Application embarquée 136
5.4.3.5 Exemple 2 : Intégration dans l'architecture schizophrène . . 136
5.4.4 Conclusion 138
5.5 Conclusion 139

6 Validation et mesures **141**
6.1 Introduction 141
6.2 Évaluation de la mise en oeuvre de FT CORBA 142
6.2.1 Gestion des groupes de répliques 142
6.2.1.1 Gestion des références sur les groupes d'objets 142
6.2.1.2 Mise en oeuvre des styles de réplication de FT CORBA 143
6.2.2 Comportement temporel des styles de réplication 143
6.2.2.1 Distribution des latences 144
6.2.2.2 Les latences en fonction du degré de réplication 145
6.2.3 Détection et notification des défaillances 146
6.2.4 Conclusion 148
6.3 Évaluation du service générique de consensus 149
6.3.1 Analyse du code source 149
6.3.2 Mesures de performances 149
6.3.2.1 Configuration de base 150
6.3.2.2 Compatibilité avec l'architecture schizophrène 152
6.3.2.3 Configuration déterministe 153

6.3.3 Conclusion 155
6.4 Conclusion 156

7 Conclusions et perspectives **157**
7.1 Contributions 157
7.2 Perspectives 160

Table des figures **163**

Liste des tableaux **165**

Bibliographie **167**

Introduction

Les techniques de tolérance aux fautes permettent de construire des systèmes robustes pouvant continuer à fonctionner malgré la possible occurrence d'inattendus. En général, même s'ils sont primordiaux pour le produit final, les besoins en tolérance aux fautes sont orthogonaux aux aspects "métiers" des applications. Pour des raisons d'efficacité et de réduction des coûts, il est fortement souhaitable de pouvoir concevoir et utiliser des solutions de tolérance aux fautes indépendamment des services fonctionnels que doivent fournir ces applications.

Le succès des intergiciels dans le cadre du développement de systèmes d'information "généralistes" comme les applications Web, encourage leur utilisation pour d'autres applications plus spécifiques, comme les applications temps réel ou même certaines applications critiques. Le développement et le déploiement de ces applications se base de plus en plus sur la réutilisation de composants (matériels et logiciels). On parle alors de composants pris sur étagère (COTS). Ces applications présentent des contraintes et des exigences en qualité de service plus difficiles à satisfaire. L'intergiciel doit maintenant répondre à de nouveaux défis et concilier de nouveaux besoins.

1 Position du problème

La sûreté de fonctionnement et la distribution sont deux domaines étroitement liés. D'une part, la distribution permet d'assurer la réplication d'entités sur plusieurs noeuds physiques (machines) ou logiques (partitions), et ainsi tolérer la défaillance de certaines de ces entités. D'autre part, la distribution nécessite des ressources matérielles et logicielles additionnelles, augmentant la probabilité de défaillances et donc le besoin d'appliquer des techniques de tolérance aux fautes appropriées. Grâce à ce lien étroit entre la tolérance aux fautes et la distribution, l'intergiciel est *de facto* l'un des meilleurs endroits pouvant héberger certaines fonctionnalités de tolérance aux fautes comme la gestion de la réplication. En outre, la consolidation de certaines fonctionnalités de l'intergiciel permet d'améliorer la sûreté de fonctionnement des applications développées et déployées à l'aide de cet intergiciel.

L'architecture schizophrène[100] donne à l'intergiciel des capacités de configuration, d'adaptation et d'interopérabilité très intéressantes, voire primordiales dans certains cas. Cette architecture a montré son efficacité même pour certaines applications présentant plusieurs contraintes en termes de consommation de ressources ou de déterminisme. Le manque de support de mécanismes généraux de tolérance aux fautes et de consensus constitue un obstacle et limite le champs des applications pouvant utiliser l'intergiciel. Nous nous proposons d'étudier la possibilité d'ajout de service de tolérance aux fautes et de consensus à l'architecture schizophrène sans compromettre ses propriétés.

Le problème du consensus peut se définir (informellement) sur un ensemble de processeurs comme suit : initialement, chaque processeur propose une valeur et tous les processeurs corrects

(par exemple ceux qui ne sont pas en panne) se mettent d'accord sur une valeur commune. Malgré sa formulation simple, le consensus est un problème fondamental sur lequel se base la majorité des algorithmes d'accord [114]. C'est le cas du multicast atomique (atomic multicast), de l'appartenance à un groupe (group membership), etc. La réduction des algorithmes d'accord au consensus a été sujet de plusieurs travaux de recherche. Par exemple [53] introduit la notion de filtres de consensus consensus filters qui permettent de résoudre plusieurs problèmes d'accord en adaptant un algorithme de consensus. L'abstraction du consensus peut également être utilisée d'une manière explicite par certaines applications. Par exemple, la réconciliation des valeurs renvoyées par plusieurs capteurs.

Pour être vraiment efficaces, les algorithmes de consensus doivent garantir un comportement temporel prévisible et fournir des garanties plus fortes que le critère de terminaison classique. En effet, Ce critère postule que chaque algorithme de consensus doit converger, sans donner aucune indication sur le temps de convergence. Or, en pratique, le temps de convergence est une donnée primordiale pour la conception et la mise en oeuvre des applications. Pour estimer les temps de convergence, des techniques telles que la simulation ou la modélisation formelle peuvent être utilisés.

La diversité des applications et la variabilité des environnements dans lesquels les applications (ou certaines briques logicielles) évoluent motivent la conception d'un service de consensus générique qui se base sur l'intergiciel à la fois comme support fonctionnel et comme garantie des hypothèses sur les modèles de calculs et ceux de défaillances. La généricité du service de consensus que nous proposons dépasse le cadre de la simple configuration aux choix transparents de l'algorithme de consensus souhaité par l'application. Ceci augmente la portabilité des applications et l'efficacité de l'intergiciel fournissant cette abstraction. L'intergiciel, en général, et les composants fournissant le service du consensus peuvent alors être considérés comme de vrais composants pouvant être pris sur étagère (COTS).

2 Objectifs et démarche

Dans cette thèse, nous nous intéressons à deux problèmes. D'abord, nous étudions les réponses offertes et les contraintes posées par l'architecture schizophrène lors du support de mécanismes généraux de tolérance aux fautes. Ensuite, nous nous intéressons à l'abstraction du consensus qui constitue une base fondamentale de plusieurs services tolérants aux fautes, et proposons une méthode de construction d'applications sûres de fonctionnement se basant sur l'ensemble des composants que nous avons défini.

Le standard `FT CORBA` propose plusieurs mécanismes généraux de tolérance aux fautes (redondances temporelles et spatiales, support de plusieurs styles de réplication classiques, etc.). En outre, il propose d'enrichir `CORBA`, un standard riche, répandu et ayant été utilisé dans le cadre d'applications critiques et temps réel. L'étude de ce standard nous permettra d'évaluer l'efficacité des réponses qu'offre l'architecture schizophrène et de proposer les améliorations nécessaires pour supporter les mécanismes de tolérance aux fautes les plus généraux. L'accent sera mis d'une part sur la détection et la notification des défaillances et d'autre part, sur le support des différents styles de réplication proposés par la norme.

Le second axe de notre travail consiste à étudier et à réaliser un service de consensus qui se veut le plus générique possible afin de se plier aux besoins du plus grand nombre d'applications. Nous nous basons sur l'expérience acquise lors de l'étude de `FT CORBA`, en particulier la gestion des groupes de répliques et la détection de défaillances pour proposer un service et une méthodologie de conception d'applications distribuées nécessitant l'abstraction du consensus. La

conception du service du consensus permettra de profiter pleinement des avantages de l'architecture schizophrène et de se plier à plusieurs contraintes imposées par ces systèmes (déterminisme, performances, exigences de modélisation comportementale, exigences de certification etc.).

Pendant notre étude de ces deux problématiques, nous définissons le rôle de l'intergiciel, les avantages de son architecture pour supporter les mécanismes de tolérance aux fautes. Plusieurs besoins et contraintes imposées par les systèmes critiques seront pris en compte tant sur le plan architectural qu'au niveau de l'implantation. Des évaluations qualitatives et quantitatives se basant sur des scénarios de tests minutieusement conçus compléteront notre étude.

3 Plan

Le document est structuré en trois parties : état de l'art, conception, et réalisation.

État de l'art

Dans la première partie, nous mettons l'accent sur les principes généraux de la tolérance aux fautes ainsi que la problématique du consensus. Nous nous intéressons également à la production des systèmes distribués critiques et aux propriétés permettant aux intergiciels de supporter les besoins de ces systèmes.

Dans un second chapitre, nous présentons les travaux sur lesquels repose cette thèse ainsi que les limitations de travaux de recherche existants. Cette étude met en valeur les contributions que nous proposons dans le domaine des intergiciels dédiés aux applications sûres de fonctionnement.

Conception

La seconde partie se décompose en deux chapitres. Dans le chapitre 3, nous présentons les principes de l'architecture intergicielle schizophrène ainsi que ces avantages. Nous nous intéressons ensuite au standard `FT CORBA`. Nous étudions ensuite les mécanismes les plus efficaces permettant d'implanter ce standard en profitant et en conservant les propriétés et les avantages de l'architecture schizophrène.

Dans le chapitre 4, nous concevons un ensemble de composants fournissant l'abstraction du consensus. Nous dégageons l'ensemble des abstractions que doit fournir l'intergiciel aux composants du consensus. En se basant sur les études théoriques autour du problème du consensus ainsi que sur certains partons de conception ; nous proposons un service de consensus dont nous maximisons dans un second temps les possibilités de configuration. Outre les cas d'utilisation basiques, nous traitons l'intégration de ce service au sein de l'architecture schizophrène et présentons les avantages de cette intégration.

Réalisation

Dans la troisième partie, nous expliquons les choix et les méthodes que nous avons utilisés pour réaliser les différents services et composants que nous avons décrits dans la deuxième partie. Pour notre implémentation et nos mesures, nous nous baserons sur une implémentation de l'architecture schizophrène : `PolyORB`[1].

Le chapitre 5 fournit une description détaillée des briques logicielles que nous avons conçus pour implanter `FT CORBA`. Dans le chapitre 6 nous détaillons l'implantation et l'ensemble des composants qui constituent le service du consensus que nous avons conçu. Afin de valider nos

[1] http ://polyorb.objectweb.org

propositions, nous avons mis en place des scénarios de test et effectué plusieurs mesures afin de monter l'efficacité des architectures que nous avons proposées et des choix d'implantation que nous avons faits. Nous en discutons les résultats et, quand c'est possible, nous comparons ces résultats avec les travaux similaires. Le dernier chapitre conclut ce mémoire en rappelant les contributions essentielles et les perspectives ouvertes par notre travail.

Première partie

État de l'art

Chapitre 1

Cadre général, notions de base et problématique

Contents

1.1 Cadre général **8**
1.2 Intergiciels **9**
1.2.1 Définition et généralités 9
1.2.2 Architecture de l'intergiciel 10
1.2.3 Apports des intergiciels 10
1.3 Tolérance aux fautes **11**
1.3.1 Définitions et notions de base 11
1.3.2 Besoins en tolérance aux fautes 13
1.3.3 Principes de la tolérance aux fautes 15
1.3.4 Conclusion 18
1.4 Consensus **18**
1.4.1 Définition 18
1.4.2 Besoins 19
1.4.3 Résultats théoriques 20
1.4.4 Conclusion 23
1.5 Production d'applications critiques distribuées **23**
1.5.1 Architecture 24
1.5.2 Utilisation des langages de programmation de haut niveau 25
1.5.3 Validation 26
1.5.4 Conclusion 27
1.6 Objectifs et axes de recherche **28**
1.6.1 Objectifs 28
1.6.2 Propositions et contributions 29

Les problèmes de tolérance aux fautes et de consensus sont posés par un nombre de plus en plus important d'applications. L'omniprésence de l'informatique dans la vie moderne et les diverses conséquences des défaillances motivent l'emploi des techniques assurant la continuité des services même en cas de défaillances. Certaines techniques de tolérance aux fautes nécessitent, selon le cas, plusieurs types d'accord dans un environnemt distribué. Ces accords peuvent par

exemple concerner les membres non fautifs d'un groupe, l'état global du système, le temps (synchronisation des horloges), l'élection d'un leader, etc. Le consensus est donc un problème général, trouvant plusieurs applications dans plusieurs domaines, y compris la tolérance aux fautes.

Les technologies intergicielles réduisent les efforts de conception et de développement d'applications distribuées. Le succès de ces technologies dans le cadre de la production d'applications généralistes motive leur utilisation pour d'autres applications plus exigeantes. L'étude de ces technologies et de leurs apports lors de la production d'applications devant présenter des garanties par rapport à leur sûreté de fonctionnement est également motivée par les pressions exercées sur les délais et les coûts de la mise sur le marché de ces applications.

Cette thèse s'intéresse au niveau de support que peuvent ou doivent offrir les intergiciels à la tolérance aux fautes et au consensus dans le cadre de la production d'applications sûres de fonctionnement.

Ce premier chapitre introductif présente les définitions et les concepts généraux relatifs aux intergiciels, à la tolérance aux fautes et au consensus. Nous nous intéressons ensuite à certaines pratiques de développement d'applications critiques. Ces applications présentent généralement des exigences fortes en sûreté de fonctionnement et utilisent souvent des services de tolérance aux fautes et/ou de consensus. Nous nous inspirons de ces pratiques et essayons de satisfaire au mieux les différentes exigences de ces applications, même si le champs de notre étude comprend également des applications moins exigeantes. La dernière section de ce chapitre présente les objectifs de notre thèse, les différents choix que nous avons effectué ainsi que nos contributions.

1.1 Cadre général

Définition 1 (Système critique) *Un système critique est un système dont le dysfonctionnement peut causer des pertes humaines ou des blessures graves, provoquer des pertes ou des dégâts matériels ou encore porter atteinte à l'environnement.*

Les systèmes informatiques deviennent de plus en plus complexes. Cette complexité se manifeste par la taille des applications (nombre de fonctionnalités, nombre de lignes de code, etc.). Elle peut aussi se manifester par le déploiement sous forme de plusieurs noeuds disposant chacun d'un espace d'adressage propre (distribution). Cette complexité peut également se manifester par l'utilisation de plusieurs fils d'exécution (concurrence). Cette complexité augmente le risque de défaillances des applications et invite à prendre les dispositions nécessaires pour garantir la continuité des services..

L'utilisation de l'informatique ne cesse de se répandre, à un point ou sa présence dans plusieurs systèmes ne se remarque plus. La qualité des applications est alors un objectif important et les conséquences des dysfonctionnements peuvent être dans certains cas dramatiques. Parallèlement, la réalité du marché et concurrence engendrent des pressions sur les coûts et les délais de mise sur le marché et invitent à optimiser les méthodes de production de ces applications. L'amélioration de la rentabilité de la production sans relaxer les objectifs de qualité est un défi dont la résolution intéresse plusieurs industriels.

Les technologies intergicielles fournissent plusieurs moyens permettant l'optimisation de la production. Ces technologies favorisent en effet la portabilité, l'interopérabilité et la réutilisation et réduisent donc les coûts du développement des systèmes distribués. D'autre part, la tolérance aux fautes, le consensus et la distribution sont étroitement liés (comme le montre 1.3.3). Le support de la tolérance aux fautes et du consensus au niveau de l'intergiciel constitue alors une contribution intéressante à la résolution de ce défi.

Le support de la tolérance aux fautes et du consensus par les intergiciels ne suffit pas à leur utilisation dans le cadre du développement d'applications sûres de fonctionnement, la qualité et l'efficacité des mises en oeuvre doivent également être garanties. Nous étudierons plusieurs aspects de la production de ces applications afin d'améliorer nos propositions et nos mises en oeuvre.

Les intergiciels ont également un autre rôle aussi important que le support de services permettant ou améliorant la sûreté de fonctionnement. Le niveau de qualité de service requis par les applications critiques et la variation des exigences et des contraintes entre les différentes applications cibles exige des capacités d'adaptation importantes au niveau de l'intergiciel. En général, il faut réussir le compromis entre la tolérance aux fautes, le temps et les fonctionnalités tout en respectant les caractéristiques de l'environnement dans lequel évolue le système. Nous étudierons les propriétés des intergiciels permettant la résolution de ces compromis.

Les prochaines sections présenteront les notions importantes et les définitions relatives aux domaines des intergiciels, de la tolérance aux fautes et du consensus. Nous présentons ensuite les pratiques de développement des applications critiques ainsi que nos propositions et contributions aux support de la tolérance aux fautes et du consensus par les intergiciels dans le cadre du développement d'applications sûres de fonctionnement.

1.2 Intergiciels

Cette section présente une vue générale des intergiciels et de leurs principes architecturaux. Elle montre également les avantages de l'utilisation de ces technologies.

1.2.1 Définition et généralités

Les définitions du terme intergiciel sont très nombreuses. Généralement, un intergiciel (en anglais middleware) est un logiciel servant d'intermédiaire de communication entre plusieurs applications, généralement complexes ou distribuées sur un réseau informatique[2]. Pour ce mémoire, nous proposons et utilisons la définition suivante :

Définition 2 (Intergiciel) *L'intergiciel est un terme désignant un ensemble de services et d'interfaces résidant entre l'application et le système opératoire. Il permet l'interaction entre plusieurs entités déployées sur une ou plusieurs machines et pouvant échanger des données grâce à un système de communications tel qu'un réseau.*

L'intergiciel doit son nom à sa position entre les entités applicatives et le système opératoire. Grâce à l'intergiciel, l'application est découplée du système opératoire sous jacent, ce qui permet aux développeurs et aux intégrateurs de faire abstraction de plusieurs détails de bas niveau liés à la distribution et aux caractéristiques des systèmes opératoires.

Les technologies intergicielles sont maintenant reconnues comme très bénéfiques voire indispensables à la conception et la réalisation de plusieurs applications distribuées. La principale fonctionnalité offerte par un intergiciel est en effet d'assurer ou de faciliter les échanges données entre entités physiquement ou logiquement séparées. Les intergiciels se basent sur des principes architecturaux intéressants tels que la séparation des préoccupations. Le paragraphe suivant fournit une vision générale sur l'architecture d'un intergiciel.

[2] `http://fr.wikipedia.org/wiki/Intergiciel`

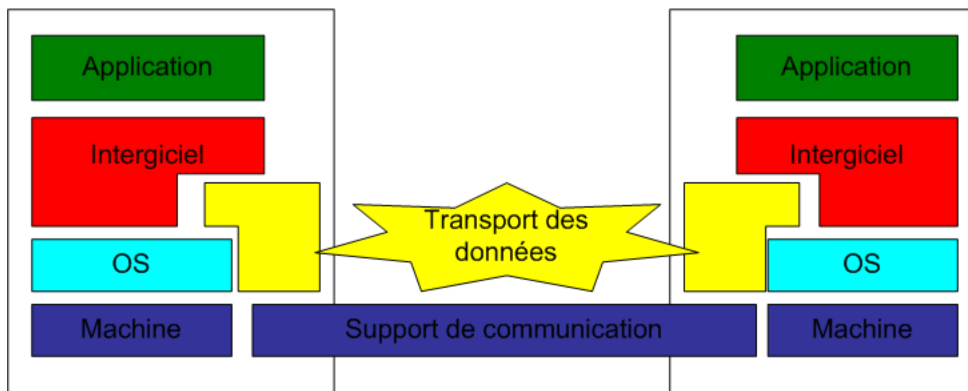

Fig. 1.1 – Application distribuée basée sur un intergiciel

1.2.2 Architecture de l'intergiciel

L'intergiciel est une *couche logicielle* fournissant un ensemble d'abstractions utiles pour la distribution et/ou pour la portabilité des applications. L'intergiciel fournit un ensemble de services généralement accessibles à travers une interface de programmation (en anglais, `API` pour Application Programming Interface). Ces services sont soit locaux, soit distribués. Les services locaux permettent d'abstraire les fonctions d'un OS. Il permettent d'assurer d'autres fonctionnalités comme la journalisation et la mise en place de caches. Les services distribués permettent d'assurer des fonctionnalités élémentaires comme les échanges de données entre les noeuds. Les services distribués les plus évolués peuvent se baser sur des protocoles complexes permettant par exemple la résolution des noms et la gestion des transactions.

La définition des `API` par les intergiciels permet donc d'abstraire les fonctionnalités de distribution et celles d'un OS. Elles présentent aux développeurs une vue uniforme des différents services locaux et distribués. Les développeurs peuvent alors se concentrer sur la logique métier des applications en faisant abstraction de plusieurs propriétés (non fonctionnelles) du produit final comme les paramètres de déploiement. La complexité liée à ces paramètres de déploiement doit maintenant être gérée au niveau de l'intergiciel.

Les intergiciels sont eux mêmes difficiles à concevoir et à réaliser. Leur mise en oeuvre doit en effet tenir compte des propriétés des environnements de déploiement des applications cibles mais aussi des exigences et contraintes imposées par ces applications. Les intergiciels doivent alors être soigneusement conçus et rigoureusement mis en oeuvre. L'architecture logicielle est un aspect fondamental de l'intergiciel et des applications réparties. Les propriétés des intergiciels sont souvent dérivées de leurs architectures. Ces architectures permettent également de faire plusieurs compromis afin de mieux satisfaire les besoins des applications cibles : intéropérabilité, portabilité, efficacité, facilité d'utilisation, adaptabilité, performances, etc. La section 2.2 fournira d'amples informations sur les architectures, et les propriétés des intergiciels. L'architecture de l'intergiciel doit donc être définie avec un maximum de rigueur et mise en oeuvre avec une très grande attention.

1.2.3 Apports des intergiciels

La position de l'intergiciel lui permet de découpler les entités applicatives du système opératoire et des systèmes de communication. Cette position permet de faire abstraction de la

distribution, des détails de mise en oeuvre et de l'hétérogénéité des langages de programmation et des systèmes opératoires.

L'intergiciel fournit un ensemble uniforme et généralement standardisé d'interfaces aux développeurs et aux intégrateurs d'applications distribuées. Ces applications peuvent alors être portées, réutilisées et composées. Il fournit également un ensemble d'interfaces, de services et d'abstractions permettant de résoudre certains problèmes récurrents indépendamment des applications. Ces différentes fonctionnalités réduisent dans plusieurs cas le besoin de développer des éléments spécifiques et encouragent la réutilisation.

Les technologies intergicielles permettent donc de réduire les coûts et les temps de production des applications distribuées. Leur utilisation a été très bénéfique dans le cadre d'un très grand nombre d'applications et surtout pour les systèmes d'informations généralistes comme par exemple les services Web.

Les succès des technologies intergicielles favorisent l'expansion de leur utilisation. Des applications de plus en plus exigeantes font partie d'ensemble des cibles de l'intergiciel. Ce dernier doit donc réconcilier des besoins de plus en plus contradictoires et présenter des garanties de plus en plus fortes quant à ses services et son comportement. La prochaine section s'intéresse à la tolérance aux fautes, l'une des techniques les plus importantes pour la sûreté de fonctionnement. La relation entre la tolérance aux fautes et la distribution et l'objectif de l'utilisation des intergiciels lors du développement d'applications sûres de fonctionnement motivent le support de la tolérance aux fautes au niveau de l'intergiciel.

1.3 Tolérance aux fautes

La tolérance aux fautes se présente comme une technique complémentaire aux actions permettant d'éviter les fautes (fault avoidance), de les prévoir (fault forecasting) et de les éliminer (fault removal). L'action d'éviter les fautes consiste à réduire autant que possible leur nombre en se basant par exemple sur un processus de développement rigoureux. L'élimination des fautes se fait pendant la phase de validation ou lors de l'utilisation du système (phases de maintenance). La prévision des fautes anticipe ces dernières afin de les éliminer ou de contourner leur effets.

L'importance des techniques de tolérance aux fautes réside surtout dans la possibilité de réagir, pendant l'exécution de l'application, à des situations imprévues. Cette section propose une introduction rapide à ce domaine. Nous montrons l'importance de la tolérance aux fautes et nous détaillons les principales avant d'en présenter les principes les plus importants.

1.3.1 Définitions et notions de base

Dans ce paragraphe, nous présentons les concepts et les définitions relatives au domaine de la tolérance aux fautes sur lesquels nous nous baserons pour la suite de ce document. Nous reprenons principalement les définitions de [74].

Définition 3 (Faute) *Une faute est définie comme une modification structurelle non voulue d'un produit. Cette modification entraîne le passage du système dans un état incorrect.*

Les fautes sont généralement invisibles depuis l'extérieur du système. Pour avoir des conséquences sur le fonctionnement correct des systèmes, les fautes doivent se transformer en erreurs (*Activation*). Dans les systèmes d'information, les fautes peuvent être classées selon plusieurs critères comme leur *activité* (actives ou latentes), leur *nature* (transitoires ou permanentes) ou leurs causes *causes* (aléatoires ou génériques). Généralement, les fautes matérielles se produisent

d'une manière isolée et sont généralement dues à l'environnement du système. Elles sont très souvent accidentelles et transitoires. Les fautes les plus fréquentes dans le monde du logiciel sont des fautes de conception. La complexité croissante des logiciels favorise l'apparition de fautes dans les logiciels. Même si plusieurs techniques de validation permettent l'élimination d'un grand nombre de fautes, il est difficile

Les techniques de validation ne permettent généralement l'élimination de toutes les fautes.

Définition 4 (Erreur) *L'erreur est la partie de l'état interne du système qui cause la défaillance de l'un ou de plusieurs services proposés.*

Dans la majorité des systèmes, il est important de détecter l'occurrence d'erreurs et de prendre des décisions permettant d'assurer le bon fonctionnement des systèmes. Cependant, la réparation des fautes à l'origine de ces erreurs, bien que très recherchée, n'est pas nécessairement essentielle pour assurer un fonctionnement correct des systèmes [103].

Définition 5 (Défaillance) *La défaillance est l'évènement qui a lieu quand le service délivré n'est plus conforme aux spécifications. C'est la panne proprement dite.*

Si un composant dépend d'un autre, la défaillance du premier est une faute qui peut à son tour provoquer une erreur interne puis une défaillance du second, c'est le mécanisme de *propagation*. Dans ce cas, il est très difficile d'identifier les vraies causes de la défaillance. La propagation peut en effet concerner plusieurs composants selon le schéma :
faute →erreur →défaillance →faute →erreur →défaillance →faute...

Définition 6 (Tolérance aux fautes et Sûreté de fonctionnement) *La tolérance aux fautes désigne la capacité d'un système à continuer à fournir un service acceptable malgré la présence de quelques fautes. La tolérance aux fautes s'intègre dans un domaine plus général qu'est la sûreté de fonctionnement (dependability). Cette dernière est une propriété (non fonctionnelle) d'un système informatique permettant à ses utilisateurs de placer une confiance justifiée dans le service qu'il délivre.*

La tolérance aux fautes intègre plusieurs attributs (fiabilité, disponibilité, sécurité, intégrité, facilité de la maintenance) [74]. Nous donnons ici les définitions des attributs qui serviront par la suite, notamment, pour exprimer les besoins en tolérance aux fautes.

Définition 7 (Fiabilité) *La fiabilité (Reliability) est la propriété exprimant l'aptitude d'un système à fournir ses services conformément aux spécifications pendant une certaine période. Généralement, on mesure la fiabilité d'un service par la probabilité de fonctionnent correct sur une période donnée.*

Définition 8 (Disponibilité) *La disponibilité (Availability) est la propriété exprimant que le système est en état de rendre son service à un instant donné. Généralement, la disponibilité d'un système se mesure en pourcentage de temps pendant lequel le système est capable de fournir (correctement) ses services.*

Définition 9 (Sécurité (ou sûreté)) *La sécurité (Safety) exprime qu'un dysfonctionnement du système n'a d'incidence catastrophique ni sur son environnement, ni sur l'utilisateur.*

La sécurité (au sens security) exprime la capacité d'un système à empêcher tout accès non autorisé aux données.

Les propriétés de fiabilité et de disponibilité ne sont pas équivalentes. Informellement, la première détermine le nombre de défaillances pendant une certaine durée, alors que la seconde détermine le temps total pendant lequel le système est opérationnel.

1.3.2 Besoins en tolérance aux fautes

Les besoins en tolérance aux fautes ne cessent d'augmenter du fait de l'utilisation croissante de l'informatique en général et de ces systèmes en particulier.

De nos jours, l'occurrence de fautes dans les systèmes informatique est inappréciable, dangereuse, ou même catastrophique : certains se rappellent encore de la chute d'Ariane 5 ou de l'échec des missiles "patriots" lors de la première guerre du Golf. Cependant, à cause de plusieurs facteurs, comme la complexité des logiciels, la dégradation physique des composants ou tout simplement parce que le risque existe[3] il est impossible d'éviter toutes les fautes, on tente alors de réduire leurs probabilités d'occurrence et d'activation.

La tolérance aux fautes est d'importance plus ou moins critique selon les applications. Dans le cas d'applications avioniques ou aéronautiques, il n'est supporté qu'un arrêt momentané (de durée très courte) des services critiques. Les exigences sont d'autant plus fortes que pour certaines de ces applications, il est impossible de procéder à des opérations de réparation ou de maintenance. En outre, les erreurs de valeurs ne sont généralement pas acceptables, car elles peuvent avoir des conséquences catastrophiques. Nous donnons ici deux cas d'échec de missions critiques à cause de l'absence ou de l'insuffisance de la tolérance aux fautes. Le premier est celui des missiles patriots en 1991 : à cause d'une faute passée inaperçue lors des tests, ces missiles n'ont pas réussi à intercepter les premiers missiles ennemis. Le cas d'Ariane 5 est plus intéressant : il motive l'utilisation des techniques de tolérance aux fautes logicielles et il montre, d'une part que la réutilisation doit se faire avec une très grande attention et d'autre part, que l'exigence en tolérance aux fautes dépasse la simple mise en oeuvre d'une redondance. La redondance doit en effet être appliquée d'une façon adéquate, en particulier, il faut que les composants redondants ne tombent pas en panne à cause d'une même faute.

L'introduction de la tolérance aux fautes dans un système n'est pas sans coût (car elle implique l'introduction d'une certaine redondance dans le système [51]). Ce coût est souvent loin d'être négligeable. La problématique est alors de déterminer le degré de la tolérance aux fautes adéquat à un système ou une application : il faut faire un compromis entre le coût et le risque [62]. Certaines techniques de tolérance aux fautes peuvent diminuer les performances globales des systèmes. Par conséquent, cette contrainte doit être prise en compte lors de la spécification des besoins de certaines applications qui présentent de telles exigences, par exemple les applications temps réel. Cristian parle alors d'un équilibre global entre coût, performances, et sûreté de fonctionnement [29].

1.3.2.1 Attributs de la tolérance aux fautes

Les besoins en tolérance aux fautes dépendent non seulement de la nature de l'application et des exigences des utilisateurs, mais aussi des conditions d'utilisation, des contraintes imposées par l'environnement, des paramètres de qualité de service requis et même les méthodes de conception. Ces besoins peuvent s'exprimer en fonction de certains attributs de la tolérance aux fautes : fiabilité, disponibilité et sécurité.

[3]La loi de Murphy est un principe empirique énonçant que s'il existe une possibilité de mauvaise manipulation d'un produit ou d'une méthode, cette mauvaise manipulation finit par avoir lieu

La fiabilité peut être utilisée pour caractériser les besoins des systèmes dont la réparation est très coûteuse voire impossible (par exemple les calculateurs embarqués sur un satellite). Généralement, il est très appréciable d'utiliser des systèmes fiables même si, pour certaines applications, on accepte certaines défaillances. La fiabilité permet par exemple de caractériser certains besoins d'applications spatiales. Par exemple, on peut trouver dans un cahier de charge qu'une station spatiale doit pouvoir fonctionner dans un mode particulier sans interruption pendant 30 jours avec une fiabilité de 80%.

La disponibilité est un critère significatif lors des spécifications des besoins. Certaines études comme [84] évaluent le coût horaire moyen de l'indisponibilité des services de certains secteurs de l'industrie américaine. Ces coûts s'évaluent à 2 500 000 $ dans le secteur des banques et atteignent 6 500 000 $ dans le courtage. Malgré la signification de ces chiffres, les exigences dans ces domaines ne sont pas les plus importantes. Pour les systèmes de contrôle du trafic aérien, l'indisponibilité ne doit pas dépasser 156 secondes par an pour certains services et 3 secondes par an pour d'autres.

La sécurité est peut être la propriété la plus importante attendue d'un système critique. En effet, cette propriété implique que le système ne présente pas d'incidence "catastrophique" sur son "environnement". Par exemple un mauvais fonctionnement du logiciel qui contrôle la température au coeur d'une centrale nucléaire ne doit en aucun cas entraîner une explosion. Les exemples illustrant l'importance de la sécurité sont très nombreux (contrôle du trafic aérien, guidage de missiles, etc.).

1.3.2.2 Domaines d'applications

L'expression des besoins en tolérance aux fautes en fonction de ses attributs présente plusieurs inconvénients. D'une part, elle peut souffrir d'imprécisions, car les attributs de la sûreté de fonctionnement dépendent de plusieurs paramètres comme l'environnement et les composants logiciels et matériels de l'application. D'autre part, ces attributs ne donnent pas d'indications sur les méthodes et les algorithmes qui doivent être utilisés.

Dans la littérature, nous trouvons d'autres méthodes plus précises pour l'expression des besoins. La première [118] propose une classification basée sur la coordination entre les "apports" des différentes techniques de la tolérance aux fautes d'une part et les besoins des applications d'une autre part. Cette classification ne prend pas en compte les fautes matérielles. La seconde [9], plus simple, classe les besoins selon la nature de l'application. Pour des raisons de clarté nous nous limitons à la seconde classification. Cette classification définit quatre classes (ou domaines) d'applications : applications critiques, applications à longue durée de vie, applications à haute disponibilité et applications commerciales.

Les applications critiques doivent justifier de plusieurs garanties au niveau de la sûreté et de l'efficacité des services qu'elles proposent. Ce besoin découle de la nécessité de protéger du matériel extrêmement cher ou de préserver des vies humaines. Le choix de modèles représentant le cas pire est généralement apprécié lors de la phase de l'expression des besoins. Il faut toutefois faire un choix judicieux de ces modèles. En effet, le choix systématique des modèles représentant les pires cas résulte généralement en un système très difficile voire impossible à mettre en oeuvre.

Les autres types d'applications présentent moins d'exigences par rapport à la sûreté de fonctionnement. Les applications à longue durée de vie (comme les satellites ou les robots récemment envoyés sur Mars) doivent pouvoir continuer à fonctionner sans maintenance durant des années, il faut alors anticiper les dégradations matérielles et les changements d'environnement. La fin des missions est généralement atteinte grâce à un usage intensif de la redondance. Les deux autres domaines d'applications présentent des contraintes moins fortes et sont moins exigeantes que les

deux premières. Elles tolèrent les interruptions de service de courte durée pour celles présentant des exigences en disponibilité. Les applications commerciales sont encore moins exigeantes.

L'expression des besoins en sûreté de fonctionnement est cruciale, elle doit être précise et complète. L'imprécision ou l'omission de certains besoins peuvent en effet causer des erreurs durant tout le cycle du développement. Le danger de la mauvaise expression des besoins est lié au fait que les problèmes qu'elle engendrent sont très difficilement détectables et qu'ils se ne manifestent dans la plupart des cas qu'à travers les défaillances. Ce paragraphe a montré l'importance de la tolérance aux fautes. Nous nous sommes intéressé aux besoins en tolérance aux fautes, présenté quelque méthodes permettant l'expression des besoins et montré l'importance de cette phase. Les paragraphes suivants présentent des éléments de la théorie de la tolérance aux fautes. Nous nous attarderons sur les techniques et résultats qui nous serviront pendant les prochains chapitres.

1.3.3 Principes de la tolérance aux fautes

La tolérance aux fautes et les systèmes distribués sont étroitement liés. D'une part, la tolérance aux fautes motive la distribution des données et des tâches parce qu'elle nécessite une certaine redondance. D'autre part, dans un environnement distribué, la multiplicité des ressources logicielles et matérielles augmente significativement les sources de fautes et les probabilités de dysfonctionnement. D'où le besoin d'accorder une attention particulière à la sûreté de fonctionnement du système et de mettre en oeuvre les solutions de tolérance aux fautes appropriées.

Cette section s'organise comme suit : d'abord, nous présentons les concepts de base de la tolérance aux fautes. Ensuite, nous présentons la réplication et donnons les raisons pour lesquelles la réplication est la solution la plus utilisée pour la tolérance aux fautes dans les environnements distribués.

1.3.3.1 Redondance : base de la tolérance aux fautes

Définition 10 (Redondance) *La redondance désigne la multiplicité et la répétition. C'est une pratique couramment utilisée lors de la conception de systèmes critiques et/ou tolérants aux fautes.*

On distingue trois forme de redondance [117], la réplication, la redondance fonctionnelle et la redondance analytique. Si la réplication permet d'exécuter exactement les mêmes traitements, la redondance fonctionnelle permet l'obtention des résultats mais en exécutant des traitements différents et la redondance analytique accepte des résultats différents s'ils restent cohérents.

Toute solution tolérante aux fautes se doit de mettre en oeuvre au moins une forme de redondance [51]. La redondance est une condition nécessaire mais pas suffisante. Le système doit en plus implémenter un mécanisme de détection d'erreurs (qui peut, lui-même, avoir besoin d'une certaine forme de redondance). Nous décrivons ci dessous les trois formes de redondance les plus connues : redondance d'information, redondance temporelle et redondance spatiale.

Redondance d'information La redondance d'information duplique les *données*. Elle est généralement utilisée pour maintenir l'intégrité des données dans un système. Les codes de Hamming illustrent ce type de redondance en définissant des bits additionnels dans une donnée pour permettre de la reconstituer en cas d'altération. La redondance d'information a plusieurs applications comme les liens de communication classiques et les supports de stockage (disques compacts, mémoire vive (RAM), etc.).

Redondance temporelle La redondance temporelle permet la tolérance au fautes en dupliquant les calculs et les traitements. Cette forme de redondance implique des coûts en temps pour corriger l'erreur, typiquement en relançant l'opération qui a produit l'erreur. Par exemple, un contrôleur de disque réessaye l'opération de lecture lorsqu'il détecte une erreur de parité. La redondance temporelle est très utile en cas de défaillances transitoires, mais sans utilité pour tolérer des défaillances permanentes.

Redondance spatiale La redondance spatiale duplique des composants matériels ou logiciels. Comme exemples se basant sur la redondance matérielle, nous trouvons, les disques RAID, les serveurs de backup. La redondance logicielle n'est digne de ce nom que si on met en oeuvre la diversité des conceptions (design diversity). Par exemple, le NVP (pour N Version Programming). Le choix de la redondance spatiale est déterminé par la nature des fautes et par le coût de la duplication des composants. Les coûts de la redondance des composants diffèrent selon la nature de logicielle ou matérielle des composants en question.

- Redondance matérielle : les défaillances matérielles sont dans la plupart des cas transitoires, la redondance matérielle peut alors s'appliquer en utilisant des copies identiques du même matériel.
- Redondance logicielle : la duplication des mêmes unités logicielles ne permettent pas de tolérer les fautes de conception et d'implémentation. La redondance logicielle ne doit pas se faire par simple copie. Les coûts de la redondance logicielle sont très élevés. Ce type de redondance n'est généralement utilisé que pour les applications extrêmement critiques.

L'application de la diversité des conceptions est un mécanisme très dépendant de l'aspect fonctionnel des applications ce qui rend la généralisation et la réutilisation impossibles. En plus, le coût de la redondance matérielle étant plus faible que celui de la redondance logicielle, on préfère généralement déployer les mêmes unités logicielles sur plusieurs unités matérielles pour mettre en oeuvre la redondance spatiale, on parle dans ce cas de réplication.

1.3.3.2 Réplication

Définition 11 (Réplication) *La réplication d'une entité est le processus permettant de créer une copie de cette entité. Dans ce manuscrit, il s'agit de la duplication cohérente des mêmes unités matérielles et/ou logicielles sur plusieurs noeuds physiques ou logiques pour mettre en oeuvre un aspect de la sûreté de fonctionnement.*

La réplication, comme forme particulière de la redondance spatiale, permet le masquage et le recouvrement des défaillances et évite les points uniques de défaillances (single point of failure). Si un noeud est inaccessible, soit parce qu'il est en panne, soit parce que les autres noeuds du groupe l'ont isolé, l'une de ces répliques permet d'assurer la continuité du service. Pour supporter les défaillances les plus graves (en particulier, les pannes byzantines), une redondance massive est généralement utilisée. Dans le cas où les pannes sont moins graves (pannes en omission ou pannes franches) une ou deux répliques peuvent suffire. Les styles de réplication définissent le comportement de l'ensemble des répliques en cas de fonctionnement normal et en cas de défaillances. Plusieurs styles de réplication existent. Nous décrivons ici les réplications active et passive.

Réplication active La réplication active (figure 1.2) consiste à mettre en oeuvre un ensemble de répliques jouant le même rôle. Dans un modèle client/serveur, on réplique les serveurs qui reçoivent tous les requêtes, les traitent (en mettant à jour, si nécessaire, leurs états internes)

et envoient une réponse au client. Ce dernier choisit une de ces réponses. Dans certaines cas, il est possible d'améliorer ce protocole en lui rajoutant un mécanisme de vote. On parle alors de réplication active avec vote. Notons que les mécanismes de vote peuvent être mis en oeuvre de plusieurs manières selon les besoins. Les mécanismes de vote peuvent également être répliqués comme pour les schémas `NMR` 1.4.2.3.

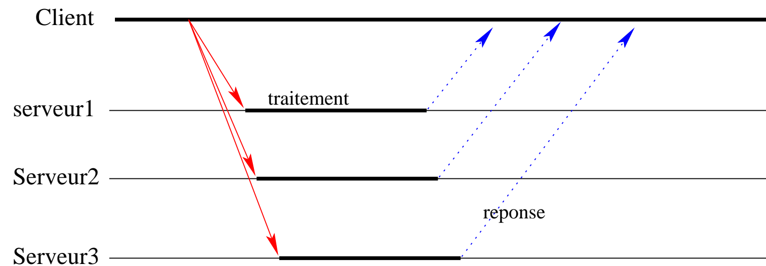

FIG. 1.2 – Réplication active

Réplication passive Comme le montre la figure 1.3, la réplication passive désigne un élément du groupe d'objets comme primaire, les autres sont appelés serveurs secondaires. Dans un modèle client/serveur, le client envoie sa requête uniquement au serveur primaire qui exécute le traitement. Une fois le calcul fini, le serveur primaire met à jour les secondaires et envoie la réponse au client. Si le serveur primaire n'est plus fonctionnel alors l'un des secondaires est promu et devient primaire.

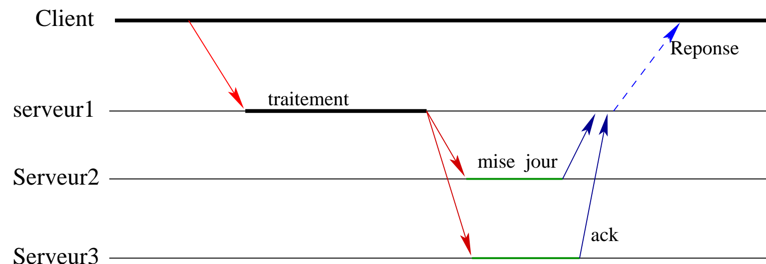

FIG. 1.3 – Réplication passive

Dans un groupe de serveurs, le consensus a une importance capitale. Par exemple, pour s'assurer que tous les processus se mettent d'accord sur la composition de ce groupe, un consensus doit être établi.

Choix du style de réplication Le choix d'un style de réplication est guidé par les besoins de l'application, par les ressources disponibles et par la nature de l'environnement de l'application. La réplication active est en général plus rapide en terme de temps de réponse. Le client peut en effet se contenter de la première réponse qu'il reçoit. En même temps, il faut supposer que tous les serveurs ont effectué le même calcul et qu'ils ont des états internes équivalents. C'est l'hypothèse du déterminisme. Malheureusement, cette hypothèse n'est pas toujours vérifiée. La réplication passive ne nécessite pas l'hypothèse du déterminisme mais elle est plus exigeante en terme du nombre de messages envoyés. Un autre inconvénient de ce style est le temps nécessaire

au recouvrement en cas de défaillance. Notons que pour ce style, il est possible de définir une synchronisation périodique des états des répliques et d'optimiser la période de synchronisation pour faire le meilleur compromis entre la consommation des ressources et la cohérence des états des répliques. Ces deux styles ne répondent pas à tous les besoins en réplication. D'autres styles intermédiaires existent comme par exemple la réplication semi-active [31] et la réplication semi-passive [37].

1.3.4 Conclusion

Pour maximiser la sûreté de fonctionnement, les différents types de redondance peuvent être appliqués simultanément. L'expression des besoins en tolérance aux fautes est généralement difficile, même si certaines méthodes peuvent avoir un effet positif. Concernant les principes des mises en oeuvre, la réplication est généralement préférée aux autres techniques de tolérance aux fautes. Le choix de la réplication est généralement argumenté par son coût relativement faible et par les résultats qu'elle offre.

La tolérance aux faute a une relation très forte avec la distribution. La réplication est un bon exemple qui montre cette relation. Cette forte relation et l'apport des intergiciels lors de la gestion des problèmes de distribution motivent l'emploi de ces technologies.

Le consensus a une relation très étroite avec la tolérance aux fautes et avec la réplication. Cependant, les applications du consensus dépassent le cadre de la tolérance aux fautes à la résolution de problèmes généraux de synchronisation et d'accord dans les environnements distribués. La prochaine section traite ce problème fondamental.

1.4 Consensus

L'abstraction du consensus est une base fondamentale permettant la résolution de la majorité des problèmes d'accord dans les environnements distribués. Nous commençons par fournir une définition précise de ce problème. Ensuite, nous montrons l'intérêt de cette abstraction ainsi que sa relation avec plusieurs techniques de tolérance aux fautes. Enfin, nous reprenons quelques résultats théoriques liés à ce problème et utiles pour la suite du manuscrit.

1.4.1 Définition

Soit un ensemble de processus $\Pi = \{p_1, p_2, \ldots, p_n\}$ reliés par des canaux de communication. Initialement, chaque processus p_i propose une valeur v_i. À la fin (si l'algorithme se termine) : chaque processus p_i décide une valeur d_i. Le problème du consensus est défini par les trois propriétés suivantes :

- Validité : toute valeur décidée est l'une des valeurs proposées
- Terminaison : si au moins un processus correct lance le consensus, tout processus correct décide au bout d'un temps fini
- Accord : la valeur décidée est la même pour tous les processus corrects

Un processus est dit correct s'il n'est pas en panne. Dans cette définition, le critère d'accord ne concerne que les processus *corrects*. Dans un système réel, ce critère est insuffisant car les processus incorrects sont autorisés à prendre des décisions même fausses et à exécuter des actions pouvant être irréversibles au risque de “contaminer” le reste du système. On définit alors le *consensus uniforme*.

- Accord uniforme : la valeur décidée est la même pour tous les processus (corrects ou pas).

1.4.2 Besoins

Malgré sa formulation simple, le consensus est un problème fondamental qui abstrait la majorité des problèmes d'accord même en la présence de défaillances. L'abstraction du consensus peut être utilisée d'une manière explicitement ou implicitement par d'autres protocoles d'accord tolérants aux fautes. Le consensus est aussi très utile pour la tolérance aux fautes dans les systèmes distribués en supportant la réplication.

1.4.2.1 Utilisation explicite

Le consensus peut être utilisé par les entités applicatives devant construire une vision cohérente de l'état du système. Par exemple, un service de consensus est requis par service répliqué effectuant des opérations de lecture de capteurs. Ce besoin est toujours valable même si les différentes entités utilisent le même capteur puisque les valeurs de sortie qu'il mesure peuvent varier avec le temps. Le consensus peut également être utilisé explicitement pour synchroniser les résultats de calculs distribués et pour prendre des décisions communes comme l'élection d'un leader ou l'exclusion un processus défaillant.

1.4.2.2 Utilisation implicite

L'abstraction du consensus peut être utilisée implicitement lors de la mise en oeuvre de plusieurs algorithmes d'accord. C'est le cas avec le problème de choix de leader (leader election) où un ensemble de processus doivent s'accorder sur l'identité d'un leader. La relation entre le consensus et certains problèmes d'accord comme l'appartenance à un groupe (group membership) et la diffusion atomique (atomic broadcast) a fait le sujet de plusieurs études théoriques. Par exemple [53] introduit la notion de filtres de consensus (consensus filter) qui résolvent plusieurs problèmes d'accord en adaptant un algorithme de consensus.

1.4.2.3 Support de la tolérance aux fautes

Ce paragraphe illustre l'importance du consensus pour la tolérance aux fautes. Comme nous l'avons précisé ci dessus, le consensus est la base de plusieurs protocoles d'accords tolérants aux fautes. Nous illustrons dans ce paragraphe la relation entre le consensus d'une part et les différents styles de réplication et la NMR (pour N Modular Redundancy) d'autre part.

Pour la réplication active, et dans un modèle client/serveur, il est nécessaire pour les répliques de recevoir les invocations dans le même ordre. Les primitives de communications doivent alors avoir des propriétés d'ordre total. Il s'agit du problème du multicast atomique. Or ce problème est connu comme équivalent au problème du consensus [22]. La relation entre la réplication active et le consensus est donc très forte. La relation entre la réplication passive et le consensus a été également étudiée. Par exemple [37] clarifie cette relation à travers le concept de la réplication semi passive. Les schémas de type `NMR` sont des cas particuliers de redondance spatiale illustrant l'importance du consensus. Une installation possible et très utilisée du `NMR` est présentée dans la figure 1.4. En haut, le schéma non répliqué utilise trois entités A,B et C. Les résultats de calcul de A et de B sont les données d'entrée pour B et C (respectivement). Si l'un des composants est défaillant, le résultat des trois traitements est presque sûrement faux. Le schéma Triple Modular Redundancy réplique A, B et C et introduit une phase de vote à la fin de chaque étape des traitements. Les mécanismes de vote doivent également être répliqués car eux aussi peuvent souffrir de défaillances. Cette étape de synchronisation des entrées et des résultats au niveau

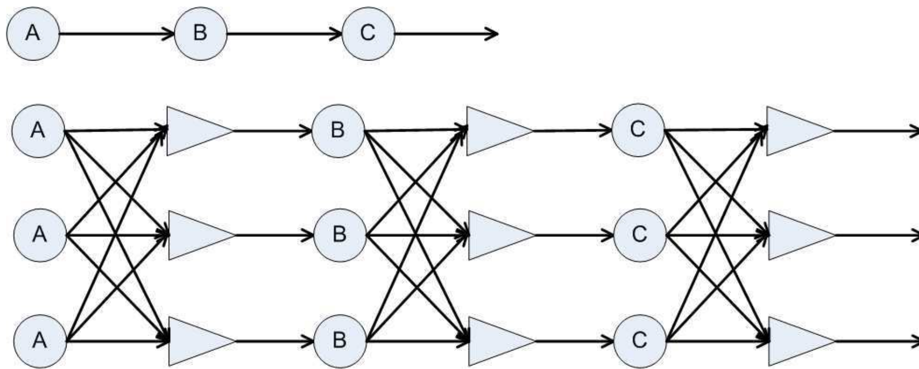

FIG. 1.4 – Triple Modular Redundancy

de chaque étape correspond au consensus. Les architectures NMR sont très résistantes et peuvent tolérer tous les types de défaillances.

Ce paragraphe montre l'importance du consensus à travers quelques exemples d'utilisation. Nous nous sommes également intéressés à la relation entre le consensus et la tolérance aux fautes, la réplication et le schéma NMR sont les meilleurs exemples illustrant ce lien. Le prochain paragraphe présente plusieurs éléments de la théorie du consensus.

1.4.3 Résultats théoriques

La nature du système défini par l'ensemble des processus et celui des canaux de communications est d'une importance capitale lors de la résolution du consensus. Par exemple, en l'absence de défaillances, le problème du consensus admet des solutions simple. Si on suppose un modèle de défaillances moins trivial (panne franche, etc.) le problème devient plus compliqué.

Dans les deux paragraphes suivants, nous nous intéressons aux modèles de défaillances et aux modèles de calcul. Ces modèles servent à exprimer les propriétés vérifiées par les unités de calcul et liens de communications et permettent l'établissement de théorèmes et de preuves autour de ces algorithmes.

1.4.3.1 Modèles de défaillances

Dans les systèmes réels comme dans les modèles abstraits, les algorithmes distribués dépendent de la nature des canaux de communications. Il est par exemple utile de savoir si un message envoyé par un processus correct sera reçu par le processus correct destinataire (critère d'absence de pertes). Il est également utile de pouvoir faire des hypothèses d'absence de création et de duplication (un message reçu par un processus a été envoyé par un autre et ne résulte pas d'un dysfonctionnement du canal). Généralement on utilise les modèles de canaux *Quasi Fiables*, vérifiant les hypothèses d'absence de pertes, de duplication et de création. Ce modèle décrit un comportement proche de celui du protocole TCP. Il existe également d'autres modèles comme le modèle de canaux à pertes autorisant la perte des messages mais pas leur création.

Concernant le comportement des processus, on trouve dans la littérature plusieurs modèles que nous détaillons dans le paragraphe suivant. Ces modèles sont très importants en particulier pour définir la force de l'"adversaire".

Arrêt sur défaillances A l'occurrence d'une défaillance, le processus arrête son activité et prévient les autres processus de son état.

Pannes franches Ce modèle suppose que les fautes ont lieu lorsqu'un processus perd son état interne et s'arrête. En général il est très utile de se ramener au cas où le composant s'arrête immédiatement après la détection d'une erreur, en effet plus le temps entre apparition de l'erreur et la défaillance (délai de latence) est long, plus la recherche des fautes et des erreurs est difficile.

Pannes par omission Dans le cas d'un système modélisé par une boîte noire avec des messages entrants et sortants, la seule déviation par rapport aux spécifications est la perte de messages entrants (omission en réception) ou sortants (omission en émission).

Pannes de temporisation Une panne de temporisation se manifeste lorsque l'exécution d'une tâche a lieu en dehors de la fenêtre du temps qui lui est dédiée.

Pannes byzantines Dans le modèle de pannes byzantines, le système peut avoir toutes sortes de comportements (y compris les comportements malveillants). Du point de vue théorique, ce modèle est très intéressant : il représente le cas "pire", c'est dire les hypothèses les plus défavorables. En général, ce modèle est utile pour les systèmes critiques placés dans un environnement hostile (par exemple les fusées, les réacteurs nucléaires, etc.).

1.4.3.2 Modèles de calcul et Synchronisme

Le modèle de calcul consiste en un ensemble d'hypothèses sur le comportement temporel des processus et des canaux de communications qui les relient. On s'intéresse généralement à deux propriétés principales :

- *Rapidité* relative des processus.
- *Temps de transmissions des messages* entre les processus.

En utilisant ces deux propriétés on définit les systèmes synchrones et asynchrones.

Définition 12 (Système synchrone) *Dans un système synchrone, il existe une borne supérieure sur les délais de transmission des messages et sur les vitesses des processus.*

Définition 13 (Système asynchrone) *Dans un système asynchrone, il n'y a pas d'hypothèse sur les vitesses relatives des processus ni sur les délais de transmissions des messages.*

La mise en oeuvre d'un synchronisme total dans un système est très difficile. La résolution du problème de consensus dans un système totalement asynchrone est généralement difficile et n'est pas systématiquement possible. Dans la littérature, nous trouvons plusieurs modèles intermédiaires comme le modèle asynchrone temporisé (Timed Asynchronous) [46], et le modèle partiellement synchrone (partially synchronous) [36].

Malgré l'existence de ces systèmes, la résolution du problème du consensus dans les systèmes asynchrones reste très intéressante de points de vues théorique et pratique. En effet, le modèle asynchrone est suffisamment générique pour pouvoir modéliser plusieurs systèmes y compris ceux ayant des contraintes temps réel. Le modèle asynchrone peut en effet être adapté à la modélisation de ce genre de systèmes[76].

1.4.3.3 Problème du consensus dans les systèmes asynchrones

Les travaux de Fisher, Lynch et Patterson [45] ont prouvé l'impossibilité de résoudre le problème du consensus dans un modèle asynchrone où les processus sont sujets aux défaillances. Pour pallier ce résultat d'impossibilité, plusieurs types de solutions ont été proposées [5].

La première solution propose l'addition d'hypothèses supplémentaires sur le comportement temporel du système. Le système n'est donc plus totalement asynchrone. Ces hypothèses supplémentaires peuvent être établies en supposant l'existence de bornes supérieures sur délais de transmissions des messages. Ces bornes peuvent être connues ou inconnues, elles peuvent également être atteintes après un certain temps du fonctionnement du système. Parmi ces modèles intermédiaires, nous trouvons les modèles partiellement synchrone[36] et asynchrone temporisé [46].

La seconde solution consiste à assumer des probabilités pour certaines transitions du système. Le résultat d'impossibilité de [45] n'est plus valable si le critère de terminaison est affaibli et qu'on ne nécessite plus qu'une probabilité de terminaison égale à un. Le principe des algorithmes randomisés est de résoudre le problème de l'existence de problèmes de terminaison dans certaines exécutions déterministes. L'existence d'une seule exécution pouvant ne pas se terminer implique que l'algorithme déterministe est faux. On évite ces situations en faisant en sorte que les processus continuent a essayer de trouver un accord en lançant des tours (rounds) supplémentaires pour augmenter la probabilité de convergence qui doit atteindre 1 au bout d'un certain nombre d'étapes qui dépend de l'algorithme. Parmi les protocoles résolvant le consensus en utilisant cette technique nous notons [4] et [41].

Une autre solution, proposée dans [22], ne fait pas d'hypothèses supplémentaire sur le synchronisme du système et ne relaxe pas le critère de terminaison. Elle enrichit le modèle asynchrone avec des détecteurs de défaillances, le prochain paragraphe présente la théorie de la détection des défaillances dans les systèmes asynchrones.

1.4.3.4 Détection des défaillances dans les systèmes asynchrones

Définition 14 (Détecteur de défaillances) *Un détecteur de fautes est un oracle distribué fournissant à l'ensemble des processus des indications sur les processus défaillants.*

Les détecteurs de défaillances ont été formalisés dans [22]. Généralement, chaque processus dispose d'un module de détection lui fournissant la liste des processus qu'il suspecte. Ils ne fournissent que des indications sur les processus qu'ils suspectent de ne pas fonctionner correctement et sont autorisés à faire des erreurs. Pour caractériser la qualité d'un détecteur de défaillances on introduit deux propriétés : la *complétude* (completeness) et la *précision* (accuracy). La complétude caractérise la détection de tous les processus défaillants, alors que la précision limite le nombre de "faux positifs" que fait le détecteur. [22] définit deux types de complétude (faible et forte) selon qu'un seul ou tous les processus corrects détectent tous les processus incorrects. Il définit aussi quatre types de précision (faible, forte, inévitablement faible, inévitablement forte), selon qu'un ou plusieurs processus ne suspectent pas ou finissent par ne plus suspecter les processus corrects. En se basant sur ces propriétés, huit classes de détecteurs de défaillances sont définies, elle sont décrites dans le tableau 1.1.

Chandra et Toueg [22] ont établi que la propriété de complétude forte peut être obtenue à partie de la complétude faible dans un système avec des canaux fiables où les processus peuvent tomber en panne franche. Ceci implique que tout problème pouvant être résolu grâce à détecteur de défaillances de classe P (respectivement S, $\diamond P$, $\diamond S$) peut également être résolu en remplaçant ce détecteur avec un autre de classe Q (respectivement W, $\diamond Q$, $\diamond W$).

Complétude	Précision			
	Forte	Faible	Inévitablement forte	Inévitablement faible
Forte	Parfait P	Fort S	Inévitablement parfait $\diamond P$	Inévitablement fort $\diamond S$
Faible	Q	Faible W	$\diamond Q$	Inévitablement faible $\diamond W$

Tab. 1.1 – Classes des détecteurs de défaillances

Notons que l'étude de Chandra et Toueg définit un ensemble *abstrait* de détecteurs de défaillances. Les aspects de mise en oeuvre n'ont pas été considérés. Des études ultérieures on montré que la mise en oeuvre de détecteurs de défaillances est plus ou moins facile selon leurs propriétés, et même que certains (comme le détecteur de défaillances parfait) ne sont pas implémentables dans certains environnements. Même si les aspects pratiques n'ont pas été traités par le travail original, la définition des classes de détecteurs de défaillances constitue un pas important vers la résolution systématique des problèmes d'accord dans les environnements distribués.

1.4.4 Conclusion

Le consensus est un problème fondamental posé par plusieurs systèmes distribués et tolérants aux fautes. Dans cette section, nous avons commencé par présenter une définition formelle de ce problème. Ensuite, nous avons montré l'importance de cette abstraction pour les systèmes distribués tolérants aux fautes. La relation entre le consensus et certaines techniques de tolérance aux fautes a également été explicitée. La dernière partie de cette section a montré certains résultats théoriques qui nous seront utiles dans la suite. La prochaine section s'intéresse aux différents aspects pratiques du développement des applications critiques. Cette section nous permettra de fixer les objectifs et les axes de recherche de cette thèse.

1.5 Production d'applications critiques distribuées

Durant les dernières décennies, les applications critiques ont subi une très grande évolution. Les fonctionnalités de ces applications sont maintenant implantées sous forme de modules logiciels plutôt qu'au niveau de cartes spécialisées. En outre, comme dans tous les autres domaines de l'informatique, la complexité de ces applications ne cesse de croître. Les conséquences des dysfonctionnements exigent une attention très minutieuse à la qualité de ces applications et encourage chaque fournisseur à proposer et à maximiser les éléments permettant de placer une confiance justifiée dans les systèmes qu'il produit.

La production des applications distribuées critiques doit résoudre plusieurs compromis. Outre les exigences de simplicité et de traçabilité et de garanties de sûreté de fonctionnement, il faut réduire les coûts de développement et limiter les délais de mise sur le marché. Ces applications ont en effet des coûts de développement très élevés pouvant être décourageants. L'optimisation de la production est également un très bon moyen pour faire face à la concurrence.

Dans cette thèse, nous partons d'une architecture intergicielle temps réel que nous améliorons afin de répondre aux besoins d'applications exigeantes en qualité. Les applications critiques présentent des exigences extrêmes en termes de qualité. L'étude des pratiques de développement de ces applications facilite la compréhension du point de vue de l'utilisateur de l'intergiciel c'est à dire l'architecte ou le chef de projet qui doit choisir entre la réutilisation de composants sur

étagère ou d'un intergiciel ou de développer une nouvelle solution spécifique à son problème. Ce choix est l'un des plus difficiles. D'une part, la première alternative peut impacter la qualité du produit final (comportement temporel instable, surconsommation des ressources, difficulté de certification, etc.). D'autre part, le choix de concevoir et de développer une solution ad-hoc est généralement très coûteuse, surtout si l'application finale est assez complexe (à cause de la distribution, de la réplication, etc.).

Nos propositions sont inspirées par les différentes pratiques de la production des applications critiques. Cette section décrit plusieurs de ces pratiques. D'abord, nous nous intéressons aux aspects architecturaux des applications critiques. Nous mettons l'accent sur l'aspect de réutilisation déjà présent dans des pratiques comme la définition de niveaux de criticité, de partitions et les techniques de Wrapping. Nous présentons ensuite les caractéristiques du développement et du codage des applications critiques. Ces pratiques peuvent être adoptées lors du développement de modules intergiciels. Enfin, nous décrivons les techniques les plus utilisées lors de la validation des applications critiques. Certaines de ces techniques peuvent être favorisées par l'architecture de l'intergiciel.

1.5.1 Architecture

Tolérance aux fautes La réplication est la forme la plus simple et la plus utilisée pour la tolérance aux fautes. Plusieurs schémas de réplication peuvent être envisagés selon la nature de l'application. En fonction de la criticité du module, il peut être nécessaire de le dupliquer (généralement trois ou quatre fois) afin de tolérer ses probables défaillances. D'autres techniques peuvent également être appliquées pour garantir le niveau de sûreté de fonctionnement requis. Pour les fonctions extrêmement critiques, on utilise des schémas `NVP` (N Version Programming). Cette technique consiste à faire développer les mêmes modules logiciels par plusieurs équipes différentes en supposant l'absence de corrélation entre les défaillances des différents modules ainsi produits. Les conditions d'activation des défaillances sont alors dé-corrélées et le système est plus robuste. Du point de vue de l'intergiciel, le support correct de la réplication ou des schémas `NMR` que nous avons présentés successivement dans les paragraphes 1.3.3.2 et 1.4.2 peut suffire au support de la majorité de techniques de tolérances aux fautes, y compris les schémas `NVP`.

Décomposition fonctionnelle et partitionnement Les fonctionnalités de l'application peuvent être cataloguées selon leurs criticités, c'est à dire en fonction des conséquences de leurs défaillances. La norme `DO-178B` définit par exemple les niveaux de criticité suivants : catastrophique, dangereux, majeur, mineur et sans effet. Le tableau 1.2 présente un exemple de définition des besoins se basant sur la notion de criticité et sur les probabilités d'occurrence pour chaque niveau. Ces différents niveaux peuvent être utilisés à des fins de certification du logiciel. Selon le niveau de criticité, le code doit répondre à des exigences de plus en plus strictes. La difficulté de la certification ainsi que son coût augmentent également avec les niveaux de criticité.

Les systèmes se basant sur les partitions sont implémentés en se basant sur une architecture assurant une très faible dépendance entre les différents modules. On assigne à chaque partition un ensemble de ressources ainsi qu'un niveau de criticité. Les niveaux de criticité servent à distinguer les fonctions critiques des autres. Cette séparation permet par exemple de rajouter des fonctions utiles mais pas nécessaires à un coût raisonnable.

Les interactions entre les modules se font à travers des *filtres* permettant les échanges de données. Ces filtres sont généralement capables de détecter les données provenant de modules défaillants. La définition de partitions empêche la propagation des erreurs et chaque partition peut alors être considérée comme région de confinement des erreurs.

Niveau	Conséquences	Définition/Risques	Probabilité d'occurrence
A	Catastrophique	Pertes de vies humaines	10^{-12}
B	Dangereux	Destructions, blessures graves	10^{-9}
C	Majeures	Perte ou indisponibilité du système	10^{-6}
D	Mineures	Dégradation acceptable du système	10^{-3}
E	Sans effet		–

TAB. 1.2 – Niveaux de criticité définis par `DO-178B`

Le déploiement des partitions peut se faire en donnant à chaque partition un accès exclusif aux ressources dont elle a besoin. Ce schéma peut engendrer des coûts supplémentaires (énergie, poids, taille, maintenance, etc.). Pour pallier ce problème, le modèle architectural `IMA` (pour Integrated Modular Avionics) définit des directives et des règles permettant l'utilisation de ressources de calcul communes par plusieurs partitions. Parmi les standards supportant l'`IMA` nous trouvons `ARINC 653`, `DO-255` et `EUROCAE WG-60`.

A titre d'exemple, `ARINC 653` définit un exécutif supportant le partitionnement dans le temps (chaque partition dispose d'une fenêtre de temps pendant laquelle elle peut accéder au processeur) et dans l'espace (chaque partition dispose de son propre espace mémoire).

Wrapping Le Wrapping [107] consiste à ajouter une interface supplémentaire par laquelle il est possible d'étendre ou de restreindre des fonctionnalités. Cette technique permet de restreindre l'ensemble des fonctionnalités d'un composant à celles nécessitées ou à celles validées. Les wrappers peuvent également être utilisées pour filtrer les données et assurer qu'ils vérifient un ensemble de conditions, par exemple l'appartenance des paramètres à un intervalle donné. Enfin, ils peuvent être appliqués pour implémenter des assertions permettant la détection, le confinement et accessoirement le recouvrement d'erreurs.

1.5.2 Utilisation des langages de programmation de haut niveau

Le code des fonctions les plus critiques doit généralement passer une longue procédure de certification ainsi qu'un ensemble de tests de toute sorte. Les standards de certification imposent, outre la conformance stricte aux spécifications, un ensemble de contraintes sur l'architecture et sur le code implémentant les fonctionnalités. Ces contraintes visent à profiter de la puissance des langages de programmation de haut niveau, tout en minimisant l'introduction de fautes lors du développement. Les contraintes favorisent en effet les analyses statiques et maximisent les qualités des exécutables générés (empreintes mémoire, comportement temporel, fiabilité, etc.). Parmi ces contraintes, nous nous attardons sur deux ensembles : les constructions autorisées et les restrictions sur les modèles de concurrence. L'importance de ces deux ensembles de contraintes découle de leurs impacts immédiats sur les pratiques courantes de programmation mais aussi sur la définition de bibliothèques réutilisables pour les applications critiques.

Règles de codage L'utilisation des langages de programmation de haut niveau comme `Ada`, et `C++` facilite la programmation d'applications critiques. Pour limiter les risques de dysfonctionnement dus aux erreurs de conception et de mise en oeuvre du logiciel, il existe aujourd'hui des standards et des directives aidant les architectes et les ingénieurs à développer des applications critiques. Parmi les standards les plus connus nous trouvons `DO-178B` et `MISRA C`.

MISRA C fournit des directives permettant d'éviter les inconvénients du langage /C++ en posant des restrictions sur l'utilisation de constructions pouvant s'avérer dangereuses. DO-178B est indépendant du langage mais il propose des règles similaires. Les règles imposées par ces standards concernent plusieurs aspects des langages de programmation : transtypage, égalité entre les types, flots de contrôle, fonctions, structures, tableaux, pointeurs, etc. Par exemple, l'utilisation des constructions goto et break et des pointeurs sur fonctions non constants sont interdites par MISRA C.

L'utilisation de la programmation orientée objet est un moyen puissant pour la mise en oeuvre d'applications critiques. Outre les problèmes classiques de fautes de programmation, le niveau d'abstraction de l'orienté objet étant plus élevé, les analyses *à priori* et le comportement temporel du produit final deviennent plus difficiles. D'autre part, l'utilisation de certaines constructions de de l'orienté objet comme le polymorphisme peut poser certains problèmes. Même si ce dernier n'est pas formellement interdit par DO-178B, son utilisation est fortement découragée. En effet, il peut introduire des difficultés d'analyses. [27] fournit des explications détaillées sur l'origine du problème et propose une solution élégante pour le résoudre.

Exécutif et modèles de concurrence Les différents processus de certification peuvent nécessiter, selon la criticité de l'application, une traçabilité durant les différentes étapes du développement, entre les besoins exprimés et les différents composants du produit final. L'exécutif généré par la chaîne de compilation, faisant partie du système final, doit lui aussi passer l'épreuve de la certification. Lorsque c'est possible, on préfère utiliser une chaîne de compilation qui ne se base pas sur un exécutif. L'édition des liens est alors simplifiée et la traçabilité est garantie (les seuls objets dans le produit final sont ceux définis par l'utilisateur). Les utilisateurs et les fournisseurs de chaînes de compilation ont intérêt à supprimer ou à restreindre les exécutifs liés à la compilation. Plus l'exécutif est compact, plus il est approprié aux applications critiques : la certification est plus facile, la consommation des ressources est plus faible et les risques de dysfonctionnement sont limités.

Dans le cas d'applications critiques concurrentes, l'intérêt de réduire la taille de l'exécutif se conjugue avec le besoin d'effectuer des analyses statiques d'ordonnaçabilité. [18] définit cinq profils de concurrence pour le langage Ada et associe à chaque niveau les analyses d'ordonnancement adéquates. Le profil le plus restrictif (niveau 0), s'apparente au profil Ravenscar [32]. Le profil Ravenscar est maintenant assez populaire, sa mise en oeuvre est très facile dans le langage Ada, ce profil commence à être supporté par d'autres langages comme JAVA.

1.5.3 Validation

La validation des applications critiques se base principalement sur les tests, les simulations et la certification. L'emploi des méthodes formelles pour la vérification se répand petit à petit. Des idées théoriques comme le correct par construction commencent à se se répandre dans les environnement de développement réalistes [55].

Définition 15 (certification) *La certification est le processus permettant d'assurer, par des tiers de confiance (certification dite tierce partie), qu'un produit est conforme aux exigences d'un cahier des charges ou de spécifications techniques.*

La certification d'un logiciel consiste à faire correspondre chacune de ses lignes à un besoin spécifié préalablement exprimé.

Parmi les standards de certification les plus connus, nous trouvons DO-178B [108], le standard le plus utilisé pour l'avionique. Il fournit un ensemble de recommandations et de règles à suivre

pour déterminer d'une manière précise et cohérente si les aspects logiciels des applications aéronautiques leurs permettent d'être utilisées sans danger. Ces recommandations s'intéressent aux différentes étapes de la production du système (depuis l'expression des besoins, à la validation en passant par les différents étapes du développement des composants logiciels). La traçabilité des documents assure que les règles et les objectifs de chaque étape ont été respectés.

Les méthodes formelles se basent sur un formalisme permettant de représenter et d'analyser certains de ses aspects. Nous trouvons deux méthodes permettant les analyses de modèles : les preuves et le model checking. Les preuves se basent sur une description rigoureuse du modèle, les propriétés sont traitées comme des théorèmes et sont vérifiées grâce à un prouveur. Parmi les formalismes couramment utilisés nous pouvons noter B [24], Z [121] et `PVS` [4]. Parmi les techniques basées sur le model checking nous notons par exemple les réseaux de Petri colorés et les automates temporisés.

La vérification formelle est généralement orthogonale à la certification. Cependant, pour limiter l'effort de modélisation et rendre ce type de vérification possible, il peut être nécessaire de réduire les fonctionnalités utilisées. Dans le cas d'`Ada`, le langage SPARK[20] est un exemple de méthodes pour l'analyse et la vérification formelle. Cette méthode se base sur un ensemble de restrictions simplifiant les analyses. Ces analyses se basent sur l'addition sous forme de commentaires d'instructions permettant la prévision du flot d'exécution.

Les pratiques de validation courantes sont très coûteuses car généralement basées sur les tests, les simulations et la certification. En effet, les tests doivent avoir lieu à plusieurs niveaux : il faut faire des tests unitaires, des tests d'intégration et ensuite il faut tester le système en entier. La simulation permet de réduire certains efforts mais nécessite des efforts supplémentaires pour développer des environnements de simulation fidèles et réalistes. La certification est également utilisée pour valider les modules les plus sensibles (dans les cas où le dysfonctionnement du logiciel peut avoir des conséquences graves).

Les pratiques de validation courantes peuvent souffrir de problèmes d'efficacité, en particulier les tests et les simulations prouvent l'existence des erreurs mais pas leurs absence. La certification impose des règles de codage strictes, et dissuade l'inclusion de toute fonctionnalité non utilisée et décourage la réutilisation. En effet, le processus de certification consistant généralement à associer chaque ligne de code à un besoin spécifique dérivé du cahier de charge initial, décourage l'inclusion de toute fonctionnalité superflue. La réutilisation de composants prouvés peut en revanche être favorisée par certaines architectures basées sur le concept du correct par construction [55] et sur les méthodes formelles [64]. Même si certaines de ces architectures ne sont pas explicitement dédiées aux applications critiques, le besoin de réduction des coûts de validation pour ces applications encourage l'adaptation de ces architectures et la réutilisation de leurs méthodes de validation dans le contexte de production des applications critiques.

1.5.4 Conclusion

Les applications critiques ont subi une grande évolution. Leur complexité tant au niveau des fonctionnalités que des architectures et du déploiement ne cesse d'augmenter. En particulier de plus en plus d'applications critiques sont maintenant distribuées. Le besoin de réduire les coûts et le temps de développement de ces applications encourage l'adaptation de techniques favorisant la réutilisation au contexte des applications critiques. La réutilisation de composants sur étagère comme ceux fournis par les intergiciels lors du développement de ces applications est un choix difficile à faire résolvant des problèmes fréquents d'optimisation du processus du développement

[4] `http://pvs.csl.sri.com/index.shtml`

mais posant des problèmes de qualité (certification, performances, etc.).

Nous avons présenté plusieurs aspects du développement des applications critiques comme les aspects architecturaux, les aspects du développement logiciel et les aspects de validation. Même si nous ne prétendons pas proposer des architectures et des composants directement utilisables ou adaptables aux applications critiques, nous nous alignons sur plusieurs des pratiques et des choix faits lors du développement de ces applications. La prochaine section détaille les objectifs et les axes de recherche de cette thèse.

1.6 Objectifs et axes de recherche

Comme pour la tolérance aux fautes, le consensus est intimement lié aux systèmes critiques d'une part, et à la distribution d'autre part. Les résultats théoriques relatifs au consensus et à la tolérance aux fautes ainsi que les exigences en qualité de plusieurs applications pouvant faire partie des cibles de l'intergiciel comme la gestion de la distribution et le respect des contraintes temporelles motivent l'utilisation des intergiciels lors du développements des applications.

L'optimisation de la production de ces applications sûres de fonctionnement est une problématique qui intéresse les industriels et les académiques. Cette thèse propose des contributions dans ce sens. À travers l'étude et la proposition d'architectures et de services de tolérance aux fautes et de consensus, nous réconcilions des objectifs assez contradictoires comme la configurabilité et les performances ou les aspects temporels et les aspects de tolérance aux fautes. Le paragraphe suivant détaille les objectifs de cette thèse et présente les axes de recherche que nous avons empruntés.

1.6.1 Objectifs

La conception et le développement d'applications sûres de fonctionnement doit relever plusieurs défis difficiles à réconcilier. L'exemple le plus significatif de ces défis est la réconciliation des besoins en tolérance aux fautes et les contraintes sur le temps. Par exemple, les temps d'exécution des services de tolérance aux fautes peuvent être imprédictibles et non bornés, donc incompatibles avec toute application temps réel. Parmi les défis à relever et à réconcilier nous notons la tolérance aux fautes, la distribution, l'optimisation du comportement temporel et la réduction des ressources utilisées ainsi que le besoin de validation et de certification. L'impossibilité de réconcilier (d'une manière raisonnable) tous ces objectifs justifient l'intérêt de maximiser leurs intersections et surtout l'intérêt de proposer une solution facilitant les compromis entre ces objectifs en fonction des besoins réels de chaque application.

Les technologies intergicielles commencent à avoir un poids de plus en plus fort lors de la conception des applications. Les intergiciels sont en effet les meilleurs éléments architecturaux permettant plusieurs types de transparence favorisant la portabilité, l'interopérabilité et la réutilisation dans le cadre du développement des applications distribuées. Le principe de la séparation des préoccupations fréquemment appliqué dans les intergiciels favorise l'adaptation des services aux besoins spécifiques de chaque application. La complexité croissante des applications, les exigences de qualité, la nécessité de satisfaire, simultanément, un ensemble de besoins pouvant être globalement incompatibles, les besoins de réutilisation et les pressions sur les cycles de développement, ainsi que le succès des intergiciels dans le cadre d'applications généralistes motivent leurs études dans le cadre d'applications distribuées exigeantes en qualité comme les applications sûres de fonctionnement et critiques. L'objectif de cette thèse est de proposer des architectures de services intergiciels permettant de supporter les besoins en tolérance aux fautes et en consensus tout en faisant attention aux aspects de configurabilité et aux exigences des applications

cibles (contraintes temporelles, consommation des ressources, besoin de validation, etc.). Pour ce faire, nous partons d'une architecture intergicielle dont nous montrons les qualités et nous l'enrichissons avec deux services de tolérance aux fautes et de consensus. Notre objectif est de proposer des architectures adaptables sans influencer le comportement temporel ni les aspects de validation. Sans aller jusqu'aux preuves complètes d'un comportement du type temps réel dur ou jusqu'à valider formellement nos implémentations, nous maximisons les optimisations architecturales pour supporter ces contraintes.

1.6.2 Propositions et contributions

L'étude du rôle de l'architecture et des services des intergiciels dans le cadre du développement d'applications exigeantes en qualité permet d'optimiser l'efficacité des processus de développement de ces applications.

Pour notre étude nous nous basons sur `PolyORB`[5] [100], un intergiciel schizophrène. `PolyORB` présente un énorme potentiel de configurabilité tout en supportant plusieurs standards comme `CORBA` [91], `DSA` [67] et partiellement `DDS` [90]. Certaines instances de `PolyORB` ont pu être vérifiées grâce à une modélisation formelle se basant sur les réseaux de Petri colorés. `PolyORB` peut également supporter des exigences très strictes ; il supporte par exemple l'allocation déterministe de mémoire, le profil `Ravenscar`. Notons que `PolyORB` permet de définir des configurations minimalistes ayant des empreintes mémoire ne dépassant pas les 300 KB. `PolyORB` doit plusieurs de ses propriétés à son architecture schizophrène. Nous fournissons une description détaillée de `PolyORB` et de son architecture ultérieurement dans ce manuscrit.

Au début de cette thèse, le support de la tolérance aux fautes dans `PolyORB` était très limité. Le premier axe de travail de cette thèse était de faire évoluer l'architecture schizophrène en proposant une architecture et une mise en oeuvre d'un service de tolérance aux fautes. Pour notre étude, nous avons choisi `FT CORBA`. Ce choix est motivé par plusieurs raisons que nous détaillerons ultérieurement dans ce mémoire. L'architecture et la mise en oeuvre proposées s'inspirent des pratiques utilisées pour la mise en oeuvre d'applications critiques et préservent les différentes propriétés de `PolyORB`. Le service proposé est compatible avec le profil `Ravenscar` et offre plusieurs possibilités de configuration (profils de concurrence, choix du style de réplication, etc.). Des mesures de performances montrent la validité de nos choix de conception et d'implémentation.

En se basant sur l'expérience acquise lors de la mise en place de `FT CORBA` dans `PolyORB`, nous avons proposé un service générique de consensus dédié aux applications critiques. Conscients du rôle de l'architecture pour la séparation des préoccupations, l'adaptation et la réutilisation, nous nous sommes particulièrement intéressés aux interactions entre les différents composants du service du consensus. Nous nous sommes également intéressés au rôle de l'intergiciel et avons isolé les services intergiciels les plus utiles pour la mise en oeuvre de l'abstraction du consensus. L'architecture de ce service a été conçue en isolant les services intergiciels de base nécessaires au bon fonctionnement de ce service, ce qui a facilité son intégration dans l'architecture schizophrène. Nous pensons également que ce service pourra être facilement adapté à d'autres architectures intergicielles. La mise en oeuvre des composants de ce service a été faite en respectant plusieurs restrictions couramment pratiquées dans le domaine comme la compatibilité avec le profil `Ravenscar` et l'absence de l'orienté objet.

Même si la gestion des aspects temps réel sort du cadre de cette thèse, nous avons pris le soin, lorsque c'était possible, de réduire le non déterminisme et de maximiser les performances. Pour chacun des composants spécifiés et mis en ouvre, nous proposons des mesures de performances

[5] http ://polyorb.objectweb.org

et des analyses complémentaires prouvant la généricité des approches et l'efficacité des mises en oeuvre. Avant de décrire en détail les différents services proposés, le chapitre suivant présente un état de l'art des intergiciels et des services de tolérance et de consensus proposés par d'autres technologies que `PolyORB`. Cet état de l'art montre les avantages de l'architecture de `PolyORB` et la validité du choix des axes de travail de cette thèse.

Chapitre 2

Intergiciels, tolérance aux fautes et consensus

Contents

2.1 Introduction 31
2.2 Intergiciels 32
2.2.1 Modèles de distribution 32
2.2.2 Propriétés des intergiciels 37
2.2.3 Synthèse 41
2.3 Tolérance aux fautes dans les systèmes distribués 41
2.3.1 Architectures et intergiciels tolérants aux fautes 41
2.3.2 Tolérance aux fautes et réplication dans `CORBA` 47
2.3.3 Synthèse 50
2.4 Support de l'abstraction du consensus par les intergiciels 50
2.4.1 Consensus, détection de défaillances : de la théorie à la pratique . . 51
2.4.2 Patrons et architectures pour le consensus 53
2.4.3 Conclusion 55
2.5 Synthèse 56
2.5.1 Technologies intergicielles 56
2.5.2 Tolérance aux fautes 56
2.5.3 Consensus 56

2.1 Introduction

Dans le chapitre précédent, nous nous sommes fixés pour objectif de partir de l'architecture schizophrène de `PolyORB` et de l'améliorer en proposant des composants et services pour la tolérance aux fautes et le consensus. Ce chapitre présente les différents concepts et motive les différents choix que nous avons effectués dans cette thèse. D'une part, il présente les travaux sur lesquels cette thèse est construite. D'autre part, il montre les limitations des travaux de recherche existants et la contribution de cette thèse aux domaine des intergiciels dédiés aux applications sûres de fonctionnement.

Ce chapitre se divise en trois parties traitant les intergiciels, la tolérance aux fautes et le consensus. La première partie décrit les intergiciels existants en les classant selon leurs modèles de distribution. Elle s'intéresse ensuite aux propriétés permettant aux intergiciels de répondre

efficacement aux exigences des applications sûres de fonctionnement et/ou critiques. Cette partie nous expliquons également les motivations du choix de `PolyORB` comme intergiciel de départ pour notre étude.

La seconde partie s'intéresse aux services de tolérance aux fautes dans les systèmes distribués. Il montre que l'architecture proposée par `FT CORBA` réalise un bon compromis entre les objectifs de configurabilité et de qualité. Les stratégies de mise en oeuvre de `FT CORBA` sont également discutées.

La troisième partie de ce chapitre décrit les différents travaux autour de la problématique du consensus. Nous nous intéressons ici à deux ensembles de travaux : les travaux faisant le lien entre la théorie et la pratique du consensus et les architectures intergicielles proposant ou se basant sur un service de consensus. Ces travaux présentent d'une part les concepts et les idées qui ont influencé nos choix et d'autre part les travaux similaires aux notres.

2.2 Intergiciels

Le succès des technologies intergicielles favorise leur prolifération et rend leur classification difficile. Nous commençons par présenter les modèles de distribution les plus utilisés ainsi que les concepts architecturaux sur lesquels ils se basent. Nous classerons ensuite les différents intergiciels selon ces modèles. La seconde partie de cette section s'intéresse au propriétés que doivent avoir intergiciels pour résoudre le double objectif de réduction des coûts et de sûreté de fonctionnement.

2.2.1 Modèles de distribution

L'ensemble des fonctions de répartition fournies par les intergiciels a un impact sur les architectures de ces derniers, nous parlons dans ce cas de modèles de distribution. Ce paragraphe donne une vision générale des technologies intergicielles les plus répandues.

Définition 16 (Modèle de distribution (ou de répartition)) *Un modèle de distribution est un ensemble cohérent de fonctions permettant aux noeuds d'un système distribué d'échanger des données.*

Les modèles de distribution les plus connus peuvent se classer en deux classes selon le synchronisme : synchrones ou asynchrones. Pour un modèle de distribution, la définition de synchronisme n'est pas liée aux garanties temporelles au niveau du système de communication de base, mais plutôt au modèle d'interactions. Dans un modèle synchrone, l'envoi d'une donnée par un noeud émetteur cause le blocage de ce dernier jusqu'à la réception de la donnée au niveau du récepteur et parfois jusqu'à la réception d'une réponse au niveau de l'émetteur. Dans un schéma asynchrone, l'envoi de données n'est pas bloquant. Nous présentons les intergiciels orientés messages (`MOM` pour Message Oriented Middleware) qui sont la forme la plus répandue des intergiciels asynchrones. Nous décrivons ensuite deux modèles de distributions synchrones : les appels de procédures distants (Remote Procedure Call) et les objets distribués (Distributed Object Computing).

2.2.1.1 Intergiciels orientés messages

Le modèle de distribution mis en ouvre par les intergiciels orientés messages permet un faible couplage entre les entités de l'application distribuée (cf figure 2.1). Ce faible couplage permet de supporter plusieurs sortes de modèles pour la propagation des données (site central, architectures maillées et/ou partiellement maillées).

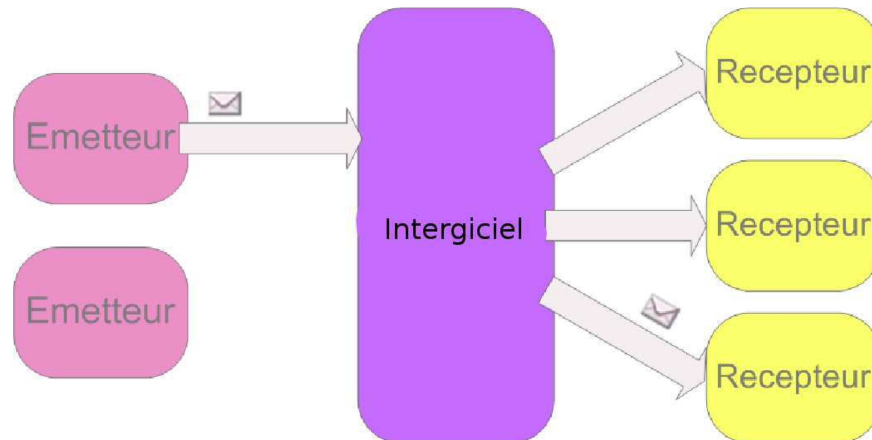

FIG. 2.1 – Intergiciel orienté messages

Le transfert effectif des données peut se faire par envoi de messages (passage de messages), en utilisant le paradigme des files de messages (message queues) [14], en se basant sur la notion de tuples ou selon le modèle publication/souscription (publish/subscribe).

Parmi ces modèles, le plus abouti est le modèle publish/subscribe. En effet, il permet de découpler les émetteurs des récepteurs dans le temps, dans l'espace et au niveau des synchronisations. Le découplage temporel assure que les entités participantes à l'interaction n'ont pas besoin de participer à l'interaction en même temps, en particulier, la déconnexion de certains récepteurs n'empêche pas l'accomplissement de l'interaction à leur re-connexion. Le découplage dans l'espace garantit que les émetteurs n'ont pas besoin de connaître les récepteurs ni d'avoir leur références et inversement. Le découplage au niveau la synchronisation assure que les interactions ne bloquent pas les participants, contrairement aux intergiciels synchrones. Dans ce modèle, la notification des données peut se baser sur trois critères principaux [39] : sur le sujet (topic-based), sur le contenu (content-based) ou sur le type de l'évènement (type-based).

Parmi les standards basés sur le paradigme publication/souscription, DDS [90] se distingue par son support de l'approche MDA et par son indépendance des plates-formes. DDS est une norme émergente pour les applications distribuées temps réel. DDS permet la gestion de la qualité de service et des cycles de vie des données. Il peut également fournir des garanties au niveau du comportement temporel et au niveau de la fiabilité de l'acheminement des données.

2.2.1.2 Appels de procédures distants

Les appels de procédures distants (RPC [13]) proposent les concepts de base des intergiciels synchrones. Dans un appel synchrone, le contrôle est transféré à l'entité appelée et l'entité appelante est bloquée jusqu'à la fin de l'appel et la réception d'une réponse (ou l'occurrence d'une exception, si ces dernières sont supportées). Les appels de procédures distants facilitent la programmation d'applications distribuées basées sur les services. Dans ce modèle l'accès à un service est effectué grâce à un appel distant. Les RPC généralisent les appels de sous-programmes au cas distribué. Pour ce faire, de nouveaux modules appelés souches et squelettes sont générés automatiquement à partir des signatures des sous programmes. L'introduction de ces modules permet d'assurer les transferts de données et assure une forme très recherchée de portabilité et de réutilisation. Les interfaces de programmation offerts par l'intergiciel sont réutilisées même si l'environnement de déploiement change, ce qui garantit la portabilité des applications se basant

sur les schémas RPC.

Au niveau du client, la souche présente un sous programme ayant la même signature que le sous programme réel. La souche s'occupe de l'emballage des paramètres, elle crée ensuite un message représentant l'appel (comprenant le nom du sous programme et les paramètres emballés). Ce message est ensuite envoyé en utilisant une bibliothèque de communication bas niveau. Au niveau du serveur, le squelette attend l'arrivée des messages depuis la souche, elle déballe les paramètres et retrouve le nom du sous programme à appeler. Elle appelle le sous programme réel et crée, en cas de besoin un nouveau message comprenant les paramètres de retour. Ce message est ensuite envoyé à la souche, qui à son tour relaye le résultat au client après déballage des paramètres de retour. Les différentes étapes du schéma d'invocation sont explicitées dans la figure 2.2.

L'intérêt de ce schéma aux développement d'applications distribuées est multiple. D'une part, tous les appels se font d'une manière synchrone comme les appels locaux. D'autre part, les souches et les squelettes sont générées automatiquement à partir des seules signatures des programmes distants ; le schéma de représentation pour l'emballage et le déballage des paramètres ainsi que les primitives utilisées pour le transfert des données n'impactent pas les besoins fonctionnels de l'application. Enfin, les détails de mise en oeuvre de la distribution sont pris en charge par la couche intergicielle et sont générés automatiquement, réduisant le risque d'erreurs de programmation et améliorant la fiabilité de l'application.

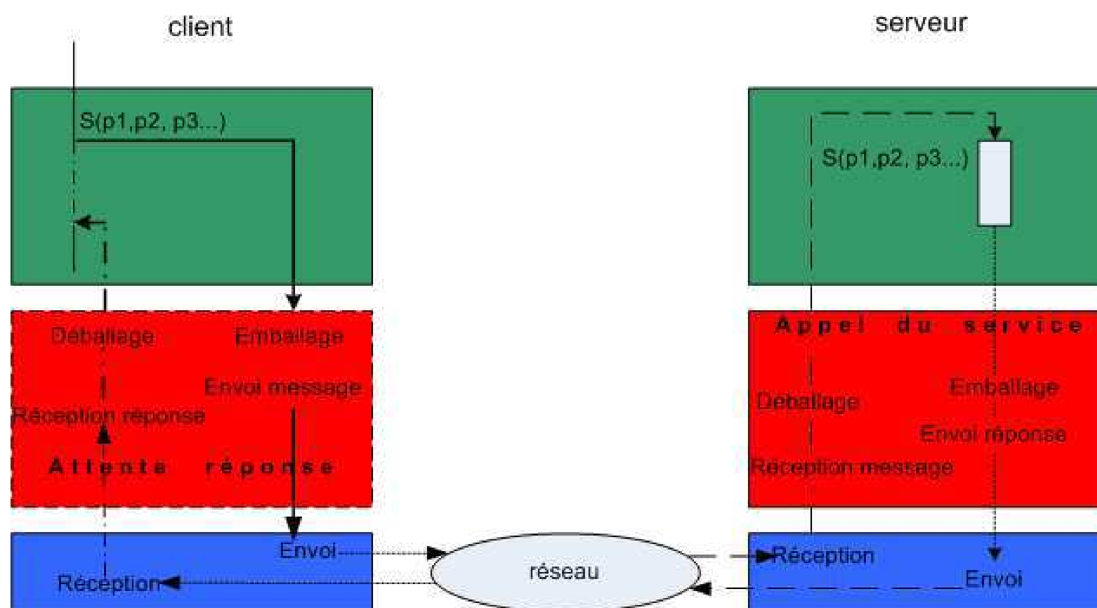

FIG. 2.2 – De l'appel local à l'appel distant

2.2.1.3 Objets distribués

Les objets distribués représentent une adaptation des RPC pour l'orienté objet. Comme le montre la figure 2.3, les principes des interactions dans ce modèle sont similaires à celles du modèle RPC.

Le modèle des objets distribués profite des nombreux avantages offerts par la technologie orientée objet. Cette technologie s'intéresse en effet à la capture des patrons les plus utilisés et

à la conception d'éléments réutilisables en profitant de concepts évolués comme l'encapsulation, l'abstraction des données, la notion d'interfaces, et le polymorphisme.

La majorité des intergiciels mettant en oeuvre ce modèle de distribution se basent sur un patron de distribution commun : *le courtier d'objets* (ORB pour Object Request Broker). Le courtier est responsable d'un ensemble de fonctions de distribution. C'est l'élément central de l'architecture et peut être sujet à plusieurs optimisations . Par exemple, il est possible d'éviter la sollicitation des couches de transport si le client et le serveur sont dans le même espace d'adressage. Le

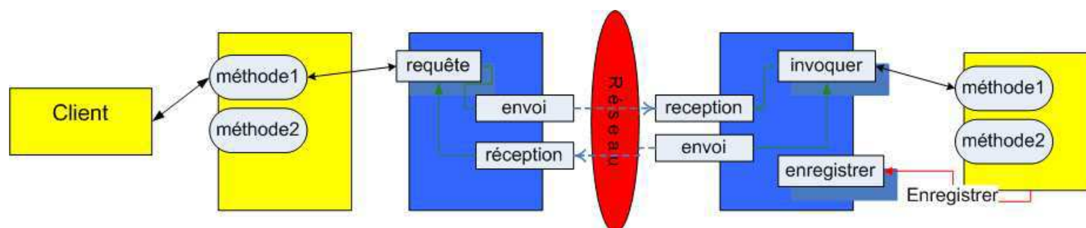

Fig. 2.3 – Interactions entre les objets distribués

patron de conception Broker définit six classes de participants : clients, serveurs, courtier, pont ainsi que deux mandataires clients et serveurs (proxies). Dans ce patron, les mandataires clients et serveurs permettent un accès uniforme aux fonctions du courtier. Ils peuvent également assurer des fonctionnalités supplémentaires comme la localisation des services (mandataire client), l'enregistrement et la gestion du cycle de vie des servants (les implémentations d'objets distants) , la construction de références et l'aiguillage des requêtes (mandataire serveur). Le courtier définit une interface générique permettant de localiser les différents objets de l'application répartie et d'assurer les échanges de données sous forme de requêtes (une requête comporte généralement la méthode distante ainsi que les paramètres d'appels). Le pont permet enfin de gérer les détails pratiques de bas niveau permettant l'échange de messages. Tous les détails de bas niveau sont masqués par l'interface du pont, favorisant l'interopérabilité.

Le modèle des objets distribués se base sur le patron Broker et permet le développement d'applications distribuées en faisant abstraction de l'architecture globale du déploiement et des propriétés locales de chaque noeud (langages de programmation, systèmes opératoires, matériel, etc.). Les applications ainsi développées sont alors indépendantes des localisations et présentent un fort potentiel de portabilité. Ce patron favorise également la séparation des préoccupations. La logique applicative et les solutions permettant les échanges des données peuvent être développées séparément. La gestion des ressources et les optimisations de performances sont également possibles à plusieurs niveaux.

Parmi les standards et intergiciels se basant sur le paradigme des objets distribués nous fournissons ici une brève présentation de RMI [73], CORBA [91] et Ice [59, 60].

RMI (Remote Method Invocations) est proposé par SUN, il permet l'invocation de méthodes sur des objets distants. Même si de récents développement de RMI lui permettent de supporter le protocole IIOP (RMI sur IIOP) et favorisent donc partiellement son intéropérabilité avec CORBA, RMI reste dépendant du langage de programmation JAVA, ce qui limite son utilisation.

CORBACommon Object Request Broker Architecture est un standard de l'OMG. CORBA est le résultat de la contribution de plusieurs centaines d'organisations, son utilisation est très répandue chez les industriels. CORBA est une famille de spécifications pouvant être utilisés conjointement. Parmi les plus importantes, nous notons la spécification de l'ORB, les service des évènements et de notification ainsi que l'interface des messages asynchrones (AMI pour Asynchronous Message Inter-

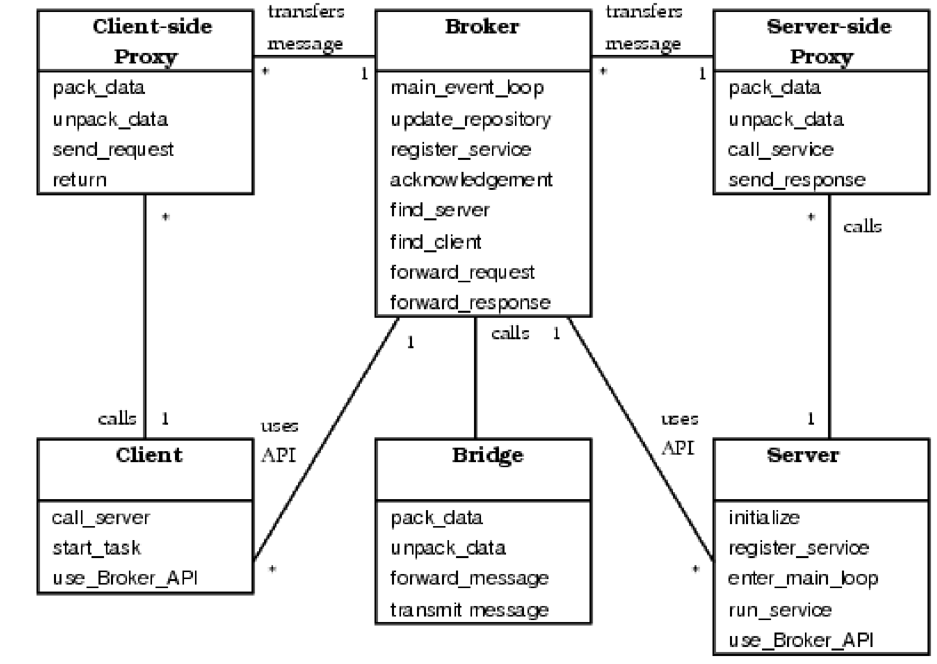

Fig. 2.4 – Le patron Broker [47]

face). Des spécifications pour le temps réel et la tolérance aux fautes (respectivement RT CORBA et FT CORBA) sont également proposées par l'OMG et sont supportées par plusieurs intergiciels.

Ice est un intergiciel orienté objets ayant les mêmes objectifs et supportant les mêmes concepts de CORBA. Ice se propose de résoudre certaines limitations de CORBA en proposant un modèle d'objets plus simple mais offrant des fonctionnalités équivalentes. Ice propose par exemple le support de UDP, l'agrégation des interfaces, ainsi que plusieurs services de sécurité.

2.2.1.4 Conclusion

Les modèles de distribution et par conséquent les intergiciels qui en sont dérivés se divisent en deux catégories selon leur synchronisme. Les intergiciels synchrones de type RPC ou objets distribués sont généralement faciles d'utilisation. Cependant, le modèle et les structures qu'il exporte peuvent poser plusieurs problèmes, par exemple en cas de défaillance des composants automatiquement générés ou des primitives de communications. Pour résoudre ces problèmes certains intergiciels introduisent un synchronisme de haut niveau. Par exemple comme CORBA propose les AMI et les services d'évènement et de notification. Cet asynchronisme “simulé” donne lieu à une inversion d'abstraction puisque le modèle des interactions reste synchrone. Nous reviendrons sur un problème posé par le synchronisme lors de la détection de défaillances plus loin dans ce manuscrit (chapitre 3).

Les intergiciels asynchrones partent d'un modèle d'interactions plus simple (envoi de messages), ce modèle ne pose pas de contraintes et permet une grande flexibilité pour l'acheminement des données et pour les réglages des paramètres de qualité de service. Néanmoins, les structures n'atteignent pas les niveaux d'abstraction des modèles synchrones. Le développement des applications est donc moins rapide. Pour résoudre ce problème, les modèles publication/souscription introduisent un niveau d'abstraction supérieur en supportant des structures de données com-

plexes et en proposant des schémas de notification introduisant une sorte de synchronisme entre les producteurs et les consommateurs. Par exemple, les modèles en `push` permettent de prévoir une ou plusieurs réactions dès l'arrivée d'un message. Les applications peuvent alors être conçues sous forme de services au prix d'un travail supplémentaire qui dépend de sa logique applicative. Nous revenons sur les modèles de notification en `push` et en `pull` dans le paragraphe 4.1.3.2. Le choix entre ces deux types d'intergiciels doit alors se faire en fonction de plusieurs compromis incluant la facilité de la gestion des aspects métiers mais également des aspects non fonctionnels tels que les performances et la sûreté de fonctionnement. Le prochain paragraphe discute les propriétés des intergiciels les plus significatives dans le contexte de notre thèse.

2.2.2 Propriétés des intergiciels

Grâce à la réutilisation, à la réduction des coûts, à la clarté des architectures et à la séparation des préoccupations, le concept des intergiciels a contribué, depuis son apparition, au développement des applications distribuées. Le spectre des applications cibles des intergiciels ne cesse de s'élargir grâce notamment à une maturation des technologies et des standards [111]. De nouvelles applications appartenant aux domaines de l'embarqué et temps réel se basent maintenant sur des intergiciels.

Dans [129] S. Vinoski prédit plus de succès aux technologies intergicielles [6]. Ce succès passe par le support des classes d'applications à fortes demandes en qualité de service et devant présenter un comportement temporel irréprochable (temps réel, très hautes performances, etc.). Ce paragraphe présente l'impact des besoins de ces applications sur l'intergiciel et sur ses propriétés architecturales et comportementales. Nous avons isolé les propriétés que doivent avoir les intergiciels pour pouvoir satisfaire, même partiellement, les besoins des applications sûres de fonctionnement et/ou critiques. Nous montrons l'importance de chaque propriété et en décrivons les intergiciels les plus représentatifs. Cette étude montre que `PolyORB` et son architecture satisfont déjà à plusieurs propriétés requises dans le contexte du développement des applications sûres de fonctionnement et/ou critiques.

2.2.2.1 Adaptation et réduction des coûts

Ce paragraphe s'intéresse à l'ensemble des moyens permettant aux intergiciels de s'adapter aux différents besoins de chaque application. L'adaptation est très importante, elle permet de réutiliser les composants et les architecture mais également de trouver le compromis entre les différents besoins et de faciliter leur réconciliation.

Pour pouvoir satisfaire les besoins d'un large éventail d'applications, il est généralement appréciable de concevoir l'intergiciel comme une collection de services ou de fonctionnalités. La maximisation des services augmente les capacités de l'intergiciel à satisfaire les différents besoins des applications. Cependant, l'instance de l'intergiciel que doit utiliser l'application ne doit pas contenir des services inutiles ou superflus. Ces services ont en général un coût très important pouvant limiter les performances et sur-consommer les ressources. Il faut que l'intergiciel puisse s'adapter aux besoins des applications en :

- ne sélectionnant que les services requis par l'application cible
- paramétrant les services sélectionnés en fonction des besoins effectifs de l'application

[6] "Due to its separation of concerns, middleware will continue to provide performance, scalability, portability, and reliability for applications while insulating them from many accidental and inherent complexities of underlying operating systems, networks, and hardware"

Ces deux objectifs sont généralement atteints par les intergiciels *configurables*. Ces intergiciels mettent en oeuvre cette propriété de configurabilité à travers plusieurs moyens que nous détaillons dans les paragraphes suivants.

L'approche GLADE GLADE [94] est une mise en oeuvre de DSA, l'annexe des systèmes distribués d'Ada. Cette mise en oeuvre propose, outre le compilateur permettant de générer les souches et les squelettes et une bibliothèque d'exécution fournissant les services nécessaires aux échanges de données, un outil de partitionnement appelé GNATDIST [70].

Cet outil supporte la configuration globale et le déploiement de l'application distribuée. Pour ce faire, il propose un langage de configuration permettant la définition des noeuds ainsi que la configuration de certaines propriétés favorisant un meilleur comportement de l'application distribuée (parallélisme, utilisation des filtres, politiques de re-connexion après une défaillance, etc.). Cette forme même limitée de configuration, couple la génération de l'application distribuée à sa configuration. Elle peut être vue comme l'ancêtre de certaines autres approches génératives que nous présentons ultérieurement.

Patrons de conception TAO(The ACE ORB) est un intergiciel se basant sur l'utilisation intensive des patrons de conception pour répondre aux besoins d'applications distribuées temps réel. Cet intergiciel se base sur un ensemble de patrons de conception pour fournir ses différents services et pour les optimiser dans certains cas[110].

Des patrons comme Strategy, Abstract Factory et Service configurator fournissent des solutions architecturales élégantes favorisant la configurabilité. Par exemple, le patron Strategy permet de changer la politique d'exécution d'un service même dynamiquement pendant l'exécution de l'application. Le patron de conception Abstract Factory peut être utilisé pour créer les stratégies et permettre d'exprimer leurs dépendances de manière cohérente. TAO n'est pas le seul intergiciel à utiliser ces patrons de conception, c'est également le cas de Quarterware [119] et de PolyORB [126].

Composants et réflexivité Il est communément admis que la configuration des intergiciels est facilitée par la programmation à base de composants. Ces composants mettent en oeuvre le principe de la séparation des préoccupations en séparant les interfaces des implémentations. Il est alors possible de fournir l'implémentation la plus adéquate aux besoins des applications.

La réflexivité permet à un programme d'accéder, raisonner et altérer sa propre représentation. Cette approche propose une solution élégante à la configuration des intergiciels. Par exemple, Quarterware [120] propose une architecture basée sur la réflection permettant la construction d'un intergiciel configurable supportant des exigences avancées comme la répartition des charges et le fonctionnement temps réel.

Configuration basée sur les modèles L'utilisation des langages de description d'architectures (ADL pour Architecture Description Language) permet d'automatiser la configuration des intergiciels. L'ADL permet la description de l'application distribuée. A partir de cette description, les composants de l'intergiciel requis sont choisis, configurés et instanciés au niveau de chaque noeud. Il est alors possible d'automatiser les phases de configuration et de déploiement [101]. Il est également possible de coupler cette approche générative avec d'autres étapes intermédiaires se basant sur outils permettant de vérifier les propriétés de l'application. Par exemple Ocarina supporte la génération de code applicatif et d'un modèle formel (se basant sur les réseaux de Petri) à partir d'une même description [125].

CoSMIC [38] est une chaîne d'outils permettant de résoudre la complexité du choix automatique de configurations des différents composants intergiciels dédiés aux applications embarquées temps réel. Ce projet se base sur un langage de configuration spécifique permettant la modélisation et l'analyse des configurations. CADENA [56] est un environnement permettant la modélisation et la vérification d'applications distribuées à base de composants. CADENA permet en particulier de modéliser les propriétés de canaux d'évènements temps réel afin d'appliquer des techniques de model checking. Ces deux projets ont été utilisés dans le cadre du développement d'applications appartenant au domaine de l'avionique.

2.2.2.2 Propriétés Comportementales

Outre les possibilités d'adaptation et de réutilisation, l'intergiciel doit satisfaire plusieurs exigences par rapport à son comportement. L'intergiciel, faisant partie de l'application finale ne doit pas compromettre l'exécution des composants applicatifs. Au contraire, il doit idéalement fournir les garanties nécessaires permettant de l'intégrer dans le produit final. Par exemple, dans le cadre d'une validation par certification, l'intergiciel doit être certifié à un niveau de criticité supérieur ou égal au niveau de son application cible. En élargissant le spectre de ses applications cibles, l'intergiciel doit pouvoir se plier à un ensemble plus large de contraintes. Cette constatation s'applique à toutes les propriétés comportementales de l'intergiciel, en particulier à son comportement temporel et à sa consommation des ressources.

Intergiciels temps réel Le comportement temporel de l'intergiciel peut être un critère de (non) sélection important, surtout dans le cadre d'applications distribuées critiques. Même si les propriétés temporelles des services intergiciels dépendent de l'infrastructure de déploiement (canaux physiques de transmission, architecture matérielle des unités de calcul), l'intergiciel ne doit pas causer de dégradation du comportement temporel.

La maturité des standards comme CORBA et DDS permet aujourd'hui de fournir des solutions utilisables par les applications distribuées temps réel. Des problèmes tels que l'inversion de priorité et le non déterminisme ont été résolus dans plusieurs contextes. Parmi les implémentations compatibles avec CORBA, TAO[7], ORBExpress[8] et microORB[9] sont la base de plusieurs applications industrielles. Parmi les implémentations de DDS, NDDS[10] se distingue par son support de plusieurs domaines d'applications (télécommunications, systèmes médicaux, avionique, etc.) mais surtout par la facilité de la spécification des différents paramètres de qualité de service et son support de la pré-allocation des ressources.

Certains intergiciels non standards sont également adaptés aux applications distribuées temps réel. Parmi ces intergiciels nous notons, ARMADA voir 2.3.1.1 et OSA+ [113] Même si ces intergiciels sont généralement très efficaces, ils sont généralement dédiés à un domaine d'applications particulier. Le potentiel de réutilisation et d'interopérabilité qu'ils offrent est généralement limité.

Consommation des ressources Les composants intergiciels et les entités applicatives sont en concurrence sur les différentes ressources du système. Les composants de l'intergiciel, ne doivent pas sur-consommer les ressources du système. La minimisation des ressources consommées est donc souhaitable, voire nécessaire si ces ressources sont déjà limitées. C'est par exemple le cas des systèmes embarqués dans lesquels de fortes contraintes sur le poids, la taille et sur l'énergie

[7] www.cs.wustl.edu/~schmidt/TAO.html
[8] http://www.ois.com/
[9] www.scisys.co.uk/
[10] http://www.rti.com

peuvent être imposées. Il est possible de limiter la consommation des ressources par les composants de l'intergiciel par exemple en appliquant certains patrons architecturaux. Le patron composant virtuel (Virtual Component) [28] permet d'optimiser la consommation de la mémoire.

Il est généralement très appréciable d'obtenir un très haut taux d'utilisation des ressources d'un système comme le processeur, la mémoire et la bande passante. La gestion des ressources est un domaine de recherche très intéressant surtout dans le cadre d'applications mobiles ou multimédia. Les problèmes de gestion de ressources sont généralement très complexes et reprennent des concepts des systèmes re-configurables (adaptation à la surconsommation des ressources), de la tolérance aux fautes (perte d'une ressource) et des systèmes embarqués et mobiles (consommation de énergie, compression pour l'optimisation des bandes passantes, etc.)

Les profils minimalistes comme Minimum CORBA permettent d'alléger l'intergiciel des fonctionnalités gourmandes en ressources, comme nous avons vu plus haut, le profil High Assurance propose un support de restrictions plus fortes. Parmi les intergiciels optimisant les ressources consommées nous notons D-RAJE [54], `nORB` [123] et RT-ZEN [102].

Support des optimisations Pour optimiser la consommation des ressources et le comportement temporel, il est très souhaitable de n'utiliser que les services requis par chaque application. Les services de l'intergiciel peuvent être séparés en deux catégories. Les services génériques devant être instanciés quelque soit l'application et les autres services dépendants des domaines d'applications et des modèles de distribution.

Dans [119], l'approche RISC (Reduced Instruction Set Computer) couramment adoptée lors de la conception des microprocesseurs est appliquée aux fonctions des intergiciels. Cette approche consiste en la définition d'un ensemble de fonctions élémentaires à partir desquelles des instructions plus évoluées peuvent être construites. `Quarterware` définit six fonctionnalités fondamentales de distribution : représentation, adressage, transport, protocole, aiguillage et invocation (Data Marshalling and Unmarshaling, Object References, Data Transport, Dispatching, Invocation Policy, Wire Protocol). Cette décomposition a permis de mettre en place des mécanismes génériques permettant plusieurs optimisations de performances et d'empreintes mémoires. Les différents autres services ne sont pas obligatoires et ne sont sélectionnés qu'en cas de besoin explicite exprimé par l'application cible. L'architecture schizophrène de `PolyORB` [126] propose une décomposition semblable se basant sur sept services canoniques. Nous revenons sur ce point avec plus de détails dans la section 3.2 .

Notons que la définition de profils minimalistes facilement extensibles est en train d'être appliquée pour le profil `High Assurance CORBA` qui se propose de partir de `Minimum CORBA` et de lui rajouter les fonctionnalités nécessaires provenant des domaines de tolérance aux fautes et du temps réel. Ce profil propose également plusieurs restrictions permettant d'augmenter le déterminisme et les performances. Les restrictions sont inspirées des pratiques courantes appliquées pour le développement d'applications critiques (restrictions sur les types d'`IDL` utilisables dans les profils et les services, restrictions des fonctionnalités, allocation statique des ressources, etc.).

Validation La validation des intergiciels est généralement empirique. Les propriétés des intergiciels sont généralement en utilisant les mêmes techniques que celles utilisées pour valider les applications qui les utilisent. Les techniques actuelles sont majoritairement basées sur les tests et la certification (par exemple, `microORB`). L'utilisation de la vérification formelle lors de la vérification des propriétés de l'intergiciel est une solution intéressante vu le rapport entre le degré de confiance qu'elle procure et son coût. `PolyORB` permet la construction d'instances d'intergiciels vérifiées contre certaines propriétés comportementales (vivacité, équité, etc.)[64].

2.2.3 Synthèse

Dans cette section, nous avons présenté les principes architecturaux les plus importants dans les intergiciels ainsi que plusieurs modèles de distribution supportés par les standards et les intergiciels. Nous nous sommes ensuite intéressés aux propriétés que doit satisfaire un intergiciel pour pouvoir prétendre de cibler des applications distribuées sûres de fonctionnement. L'intergiciel doit, d'une part, satisfaire aux propriétés d'adaptation et de réutilisation facilitant les cycles de développement et d'autre part, présenter un ensemble de propriétés comportementales durant l'exécution de l'application. Parmi les modèles de distributions synchrones, le modèle des objets distribués et la famille des spécifications `CORBA` présentent des propriétés intéressantes et ont déjà été utilisées dans des applications industrielles embarquées, sûres de fonctionnement ou temps réel. [57] traite les défis à relever par les intergiciels pour satisfaire les besoins des applications distribuées critiques et montre que `CORBA` admet déjà plusieurs propriétés recherchées dans ce contexte. `DDS` implémente un modèle de distribution asynchrone et est également utile au développement d'applications ayant des exigences similaires. Lors des différents analyses des propriétés utiles, nous remarquons qu'en général, chaque intergiciel résout un ou deux problèmes en relaxant les autres besoins. Par exemple `TAO` est bien adapté aux applications temps réel, cependant certaines instances de cet intergiciel présentent une empreinte mémoire supérieure à 2MB alors que certains autres intergiciels ne dépassent pas les 50KB mais n'offrent pas autant de services que `TAO`.

`PolyORB` figure parmi les technologies les plus citées lors de notre étude des propriétés des intergiciels. En effet, `PolyORB` se caractérise par son support *simultané* de l'adaptation de la configuration et de la vérification formelle. Il propose une architecture basée sur un ensemble de services de distribution canoniques permettant d'abstraire, d'optimiser et de vérifier les fonctionnalités de ses instances. Grâce à son architecture, `PolyORB` supporte également la cohabitation des modèles de distribution augmentant ainsi son potentiel d'interopérabilité et de réutilisation. L'état de l'art que nous avons dressé dans cette section montre que `PolyORB` est un bon choix comme intergiciel de départ pour supporter et réconcilier les besoins d'applications distribuées sûres de fonctionnement. Dans cet état de l'art, nous n'avons pas étudié les propriétés de tolérance aux fautes et de consensus. Leur importance pour notre thèse nous a poussé à leur consacrer les deux prochaines sections. Le but de ces deux sections est de montrer les travaux qui ont influencé nos propositions pour le support de la tolérance aux fautes et du consensus dans `PolyORB`.

2.3 Tolérance aux fautes dans les systèmes distribués

Cette section s'intéresse à la tolérance aux fautes dans les systèmes distribués. Nous nous sommes concentrés tout d'abord sur les fonctions de tolérance aux fautes supportés par différentes architectures. Nous montrons que `FT CORBA` fournit un ensemble équilibré et très complet de fonctionnalités. La seconde partie donc les mécanismes permettant le support de la tolérance aux fautes dans `CORBA`, elle inclut les différentes stratégies d'implémentation de `FT CORBA`.

2.3.1 Architectures et intergiciels tolérants aux fautes

La tolérance aux fautes est un domaine assez vaste. Pour évaluer l'efficacité des différentes solutions, nous nous basons sur les critères suivants : le support de la redondance spatiale en général et de la réplication en particulier, le support de la redondance temporelle et la détection des défaillances. Nous évaluons également les propriétés de transparence et d'interopérabilité ainsi que le potentiel de réutilisation (COTS). Les architectures étudiées ont été classés selon

trois catégories : les intergiciels propriétaires, les intergiciels standardisés et les architectures génériques supportant plusieurs standards ou définissant des cycles de développement entiers d'applications sûres de fonctionnement.

2.3.1.1 Intergiciels propriétaires

Les intergiciels que nous définissons comme propriétaires n'implémentent pas un standard particulier. Certains sont développés spécifiquement pour répondre aux besoins en tolérance aux fautes d'un domaine d'applications particulier. Les solutions proposées sont généralement très efficaces mais également coûteuses et peu réutilisables.

`Maruti`[11] a pour objectif de supporter le développement d'applications distribuées temps réel tolérantes aux fautes. `Maruti` permet de réconcilier les objectifs de tolérance aux fautes et de temps réel en combinant la réplication (pour la tolérance aux fautes) et la réservation des ressources (y compris l'ordonnancement) pour garantir le comportement temps réel. La redondance mise en oeuvre par `Maruti` permet la détection et la réaction temps réel aux défaillances. La défaillance d'une réplique ne peut pas influencer les propriétés temps réel des autres répliques. En cas d'impossibilité de satisfaire la qualité de service spécifiée, le système peut annuler les garanties préalablement établies et passer à un mode dégradé.

Les problèmes de portabilité sont partiellement résolus. En effet, `Maruti` définit clairement un ensemble minimal de propriétés devant être garanties par l'architecture matérielle sous-jacente. Cependant, l'architecture est peu extensible. Notons que la réutilisation et l'interopérabilité ne sont pas traitées par ce projet.

ARMADA [1] est un ensemble de services intergiciels développés pour répondre aux besoins en distribution et en tolérance aux fautes des applications embarquées et temps réel. ARMADA fournit un service de communications permettant l'échange de messages selon des besoins prédéfinis en qualité de service. Ce projet dispose également d'un ensemble de protocoles de communications de groupe (RTCAST). Ce service permet le transport de messages en respectant les contraintes temporelles, les propriétés d'ordre total, et les retransmissions. Le service de diffusion atomique temporisée permet divers types de synchronisation entre les répliques. ARMADA supporte les applications temps réel tolérant un léger taux d'incohérence entre les états des répliques. A cet effet, il met en oeuvre un schéma réplication passive à chaud. Ce schéma permet de faire le compromis entre les contraintes temporelles et la cohérence de la réplication.

Malgré les propriétés temps réel, les mécanismes de tolérance aux fautes offerts par ARMADA sont très limités. En particulier, la détection des défaillances est déléguée au protocole de communication de groupe. En outre, cet intergiciel ne supporte qu'un seul style de réplication, et il n'y a pas de support pour les ré-invocations.

ROAFTS (Real-Time Object-Oriented Adaptive Fault Tolerant Support) [72] est une architecture intergicielle supportant plusieurs systèmes opératoires grand public tels que UNIX et `Windows NT`. ROAFTS met en oeuvre certaines stratégies de tolérance aux fautes comme les blocs de recouvrement et les blocs de recouvrement distribués. Il permet également une adaptation dynamique du mode opérationnel en réponse aux changements de certains paramètres comme l'utilisation des ressources.

Cet intergiciel se base sur le modèle TMO (Time-triggered Message-triggered Object) dans lequel les services se déclenchent en fonction du temps ou à l'arrivée de messages. Lorsqu'un service

[11] `http://www.cs.umd.edu/projects/maruti`

est appelé, l'ordonnanceur local crée une tâche qu'il ordonnance en fonction de son échéance. Le système se base sur une unité de supervision permettant la détection des défaillances et nécessite une synchronisation des horloges des différents noeuds. ROAFTS permet la définition de délais sur les temps de recouvrement et peut ainsi garantir un comportement temps réel même en la présence de défaillances.

Bien qu'il présente plusieurs avantages au niveau de l'architecture, ce système est destiné aux applications à large échelle et ne satisfait pas les besoins des applications critiques embarquées. Ses services souffrent de complexité et de sur-utilisation des ressources.

Conclusion D'une manière générale, les architectures propriétaires se basent sur les propriétés des composants matériels et des systèmes opératoires. Ces composants sont généralement développés en fonction d'une seule application cible à très haut niveau de criticité et sont donc peu réutilisables et très coûteux. Pour les architectures propriétaires, l'intergiciel fournit un ensemble minimal de fonctionnalités basiques comme le transport de données. Les services génériques et les différentes abstractions généralement offertes par les intergiciels sont peu ou pas utilisées. Les procédures de validation de ces applications se basent généralement sur la certification et sont très coûteuses. La prochaine section s'intéresse aux architectures génériques pour la tolérance aux fautes.

2.3.1.2 Architectures génériques

Les architectures génériques peuvent supporter plusieurs intergiciels et modèles de distribution. Elles permettent l'utilisation de composants pris sur étagère (COTS) grâce à des techniques de confinement d'erreurs et de réplication. Certaines de ces architectures ont été instanciées dans le cadre de projets industriels.

DeDiSys définit des modèles, des métriques ainsi qu'une architecture permettant de faire le compromis entre la cohérence des données et le niveau de disponibilité requis par l'application cible. DeDiSys a pour objectif la satisfaction des besoins de plusieurs applications allant des systèmes nécessitant une très haute intégrité de données comme les systèmes de transactions bancaires aux systèmes nécessitant une grande disponibilité comme certains systèmes critiques. Cette architecture dispose de plusieurs atouts de généricité, en effet DeDiSys ne dépend pas d'un intergiciel ni d'un standard donné. En plus cette architecture permet la réutilisation de COTS. Plusieurs prototypes se basant sur les EJB, .NET et CORBA ont été réalisés.

DeDiSys présente plusieurs éléments architecturaux semblables à ceux de FT CORBA avec un gestionnaire de réplication, un service de gestion des groupes, des protocoles de réplication, etc. Le nombre et la lourdeur des services qu'elle propose en plus (gestion des transactions, gestion de la cohérence, etc.), le manque de documentation sur les capacités de configuration et le manque de concrétisations de cette architecture de haut niveau dans des cas pratiques semblent être des obstacles avant l'adoption de cette architecture dans des applications temps réel ou critiques.

FRIENDS(Flexible and Reusable Implementation Environment for your Next Dependable System) [42] propose des mécanismes permettant la constructions d'applications tolérantes aux fautes. La flexibilité est l'un des atouts de FRIENDS. Son architecture se base en effet sur un ensemble de bibliothèques de *méta objets*. L'architecture de FRIENDS se décompose en trois niveaux : le niveau noyau assurant la portabilité, le niveau système fournissant les fonctionnalités essentielles de tolérance aux fautes et le niveau utilisateur permettant la mise en oeuvre des applications.

Pour atteindre l'objectif d'indépendance du système opératoire en prévoyant un sous système fonctionnant au dessus de noyaux minimalistes. Grâce à une architecture réflexive, ces sous-systèmes peuvent être réutilisés ou portés lors de la migration vers une nouvelle plate-forme. Le niveau système définit un ensemble de sous systèmes implémentés sous forme de méta objets et responsables chacune d'une fonctionnalité (détection d'erreurs, communications de groupe, gestion des domaines de réplication, recouvrement, etc). FRIENDS fournit également une implémentation de gestion de stockage stable, ainsi qu'une implémentation des protocoles de réplication passive et semi-active (leader-followers). Le niveau applicatif utilise et configure les méta-objets. Le comportement des méta-objets dépend justement des informations de configuration fournies par les objets applicatifs.

L'architecture de FRIENDS est certes novatrice. Elle se base sur une architecture adaptée à l'intégration des besoins de validation dès les premières phases de conception. Cependant, elle souffre de problèmes d'interopérabilité, de performances et d'absence de garanties sur le comportement temporel.

GUARDS est une architecture générique pour la sûreté de fonctionnement de systèmes temps réel et d'applications critiques. GUARDS définit un environnement de développement supportant et et favorisant réutilisation de COTS comme les OS grand public. Les systèmes opératoires pris sur étagère présentent généralement plusieurs failles de sûreté de fonctionnement, GUARDS se base sur la définition des régions de contenance des défaillances (Failure Containment Regions) et sur la redondance massive selon trois axes : l'axe des canaux (duplication des canaux), l'axe intra-canaux (duplication des ressources) et l'axe de l'intégrité (définition de pare feux pour protéger les composants critiques des défaillances d'autres composants du système). GUARDS se fixe des objectifs de compatibilité avec les besoins des applications temps réel, de généricité et de supporter la validation et les certifications. Afin de réduire les coûts de la validation, GUARDS propose de prendre en compte ces besoins dès la phase de conception, de réutiliser des composants déjà validés et de supporter des composants ayant des niveaux de criticité différentes dans la même instance de l'architecture.

GUARDS propose une architecture puissante et réaliste avec des applications critiques appartenant à des domaines exigeants comme l'espace et le nucléaire. Cependant, les problèmes de portabilité, de flexibilité et d'interopérabilité ne sont que très peu supportés par ce projet.

Chameleon [69] est une infrastructure adaptable se basant sur les agents mobiles pour contrôler les aspects de tolérance aux fautes des applications distribuées. Il supporte simultanément plusieurs niveaux de *disponibilité*. Chameleon a été utilisé dans un système de contrôle de trains. Ce système utilise le schéma NMR (voir paragraphe 1.4.2.3). Chameleon permet d'isoler les unités défaillantes et en cas de besoin, de réintégrer les unités défaillantes relancées. L'architecture de Chameleon se base sur un gestionnaire temps réel responsable de l'exécution du système sous les contraintes du temps réel. Ce gestionnaire permet de calculer des temps d'exécution au pire cas. Pour cela, il suppose l'existence d'un sous système de communications dédié et d'un OS compatible avec POSIX.

La principale caractéristique de Chameleon est son architecture basée sur les agents. Cette architecture supporte facilement l'intégration de nouvelles fonctionnalités. Les besoins en tolérance aux fautes et en comportement temporel déterministe sont pris en compte en configurant ces agents. Chameleon supporte également la distribution et l'hétérogénéité des noeuds.

Conclusion Les architectures génériques présentent un grand potentiel d'adaptation et de réutilisation. Il se basent sur des paradigmes de programmation innovants comme la réflexivité et la programmation par aspects pour fournir un support de la tolérance aux fautes. Ces systèmes présentent certaines formes de transparence et permettent la réutilisation de composants sur étagère. Cependant, toutes les architectures étudiées souffrent de problèmes d'efficacité et échouent à réconcilier les besoins d'adaptation, de portabilité et de réutilisabilité d'une part avec le support des contraintes temporelles et de mécanismes avancés de tolérance aux fautes d'autre part.

2.3.1.3 Intergiciels standards

La majorité des standards a été initialement conçue pour les systèmes d'informations "généralistes". Ils répondent en premier lieu aux problèmes d'intégration et d'interopérabilité. La prise en compte des besoins comme le temps réel ou la tolérance aux fautes est généralement effectuée plusieurs années après la première publication des différents standards. Ce support est généralement limité, inefficace, et parfois inexistant. C'est le cas de `DCOM`, `RMI` et `JMS`. `DCOM` supporte quelques mécanismes de tolérance aux fautes au niveau protocolaire. `RMI` dispose de quelques mécanismes de protection contre les défaillances du réseau. Cependant ces deux intergiciels ne fournissent qu'un support moyen voire faible. Par exemple, il ne supportent pas la réplication d'objets ou de services. Bien que `JMS` soit devenu maintenant un standard bien établi, il ne supporte pas encore la tolérance aux fautes. De nouveaux travaux comme JMS-Groups proposent d'étendre cette spécification pour supporter les communications de groupe. Cette extension peut être la base de travaux ultérieurs pour la proposition d'un service de tolérance aux fautes.

Certains standards commencent à supporter certains mécanismes généraux pour la tolérance aux fautes. C'est le cas de `J2EE`, d'`Ice` et de `CORBA`.

`J2EE` Même si les mécanismes de tolérance aux fautes ne sont pas supportés au niveau du standard, il existe certains intergiciels compatibles avec `J2EE` et tolérants aux fautes. Nous étudions ici `ADAPT` et `JSR`.

`ADAPT` [6] est un canevas intégré au serveur d'applications JBOSS. Il permet l'application de protocoles de réplication. Il est en effet possible de supporter la réplication d'`EJB` et de certains services Web. `JSR` (pour Java Server Recovery) [16] propose des mécanismes plus avancés pour la tolérance aux fautes. Il masque l'occurrence de fautes aux clients de l'application, traite les fautes qui surviennent lors de l'exécution d'une requête sur un serveur en tentant de rejouer la même requête sur d'autres serveurs de la grappe. Le système `JSR` se base sur les techniques de programmation par aspects et peut être intégré à toute application `J2EE`.

Les solutions existantes pour `J2EE` sont principalement basées sur le masquage de défaillances franches ou le redémarrage de composants logiciels fautifs. La première technique est généralement réalisée en introduisant une indirection simple côté client (par exemple grâce à un proxy) permettant d'invoquer les requêtes sur un ensemble de serveurs répliqués.

`Ice` permet la réplication d'adaptateurs d'objets, supporte certains mécanismes de ré-invocation de requêtes ainsi que la notion de groupe de répliques. L'unité de la réplication dans `Ice` est l'adaptateur d'objets. Ceci est peut être une réponse à certains industriels qui trouvent que la réplication des objets est trop "grain fin". `Ice` utilise un mécanisme de proxy permettant d'essayer des adresses alternatives en cas de défaillances. `Ice` permet également de définir et de référencer un groupe de répliques. Les services offerts par `Ice` pour la réplication restent cependant très limités. Il n'y a pas de détecteurs de défaillances, ni de synchronisation d'états, ni la possibilité

de mettre en oeuvre les différents algorithmes de réplication présents dans la littérature. A part la notion de la réplication d'adaptateurs d'objets, les mécanismes proposés par Ice ont des équivalents dans la spécification de FT CORBA.

Implémentations de FT CORBA Nous décrivons dans ce paragraphe les grandes lignes architecturales des implémentations de FT CORBA. Ces grandes lignes sont naturellement définies par le standard. Les impléntations de FT CORBA seront détaillées un peu plus loin dans ce chapitre.

CORBA définit une spécification complète d'un service de tolérance aux fautes : FT CORBA. La spécification CORBA a été étendue pour supporter la tolérance aux fautes. En avril 2000, la première version de FT CORBA a été adoptée par l'OMG. Depuis décembre 2001, cette spécification fait partie du standard. FT CORBA supporte la majorité des mécanismes pour la tolérance aux fautes. En effet, FT CORBA supporte efficacement la redondance spatiale et temporelle, offre des mécanismes génériques pour la journalisation, la synchronisation des états, la détection des défaillances et la gestion des groupes d'objets.

La configuration et la génération automatique de services pour FT CORBA en fonction des besoins en sûreté de fonctionnement est possible depuis quelques années. Même si ces fonctions s'effectuent au niveau de l'intergiciel, il y a une tendance par l'OMG à faciliter ces processus. En effet, l'OMG a dernièrement standardisé des profils UML permettant la modélisation de la qualité de service et la sûreté de fonctionnement des applications. Par exemple, [92] permet l'analyse des risques et la configuration des services permettant de pallier ces risques. Même si ce profil s'intéresse principalement aux aspects de la qualité de service, il peut être utilisé comme point de départ pour la standardisation et l'établissement de modèles de référence pour l'expression des besoins en tolérance aux fautes d'une part et pour le choix et la configuration des mécanismes permettant la satisfaction de ces besoins d'autre part. Notons que les métriques et les modèles qu'il présente sont en relation avec la spécification de FT CORBA et en particulier avec les propriétés standards de groupes d'objets. Le paragraphe 2.3.2 fournit plus de détails sur l'état de l'art de la tolérance aux fautes pour les applications CORBA et notamment celles basées sur FT CORBA.

Conclusion La classe des intergiciels standards, généralement destinée aux systèmes d'informations de type serveur Web n'inclut qu'un nombre limité de mécanismes de tolérance aux fautes. L'intégration de la tolérance aux fautes dans les standards est généralement tardive et limitée. Elle est tardive car la tolérance aux fautes n'est pas le premier objectif de ces intergiciels et standards. Elle est limitée par le nombre de mécanismes supportés. Dans les modèles client/serveur l'intelligence de la réplication est fournie principalement au niveau du client et la synchronisation des états des répliques n'est pas supportée par des standards très répandus comme J2EE. Néanmoins, la maturité de CORBA et son adoption par plusieurs organisations touchant à la sûreté de fonctionnement et au temps réel fait de lui une exception chez les standards du moins au niveau des fonctions de tolérance aux fautes qu'il propose.

2.3.1.4 Synthèse

Dans le domaine de la sûreté de fonctionnement pour les applications temps réel et critiques, l'efficacité et la réduction des coûts sont des critères de plus en plus importants pour le choix des architectures. L'étude que nous avons menée montre deux extrêmes. Nous trouvons d'une part des solutions propriétaires très coûteuses et peu réutilisables qui servent au développement

d'applications avec des exigences très strictes. D'autre part, il existe des solutions beaucoup moins chères et même libres destinées aux applications présentant beaucoup moins d'exigences.

Les architectures génériques représentent le juste milieu entre ces deux classes. Ces systèmes présentent certaines formes de transparence et permettent la réutilisation de composants sur étagère. Cependant, toutes les architectures étudiées souffrent de problèmes d'efficacité et échouent à réconcilier les besoins d'adaptation, de portabilité et de réutilisabilité d'une part et les aspects de performance et les mécanismes avancés de tolérance aux fautes d'autre part.

FT CORBA est le standard fournissant le support le plus complet de la tolérance aux fautes. D'autre part, comme le montre notre conclusion sur l'état de l'art des intergiciels (paragraphe 2.2.3) CORBA supporte maintenant certaines applications temps réels et critiques. En outre, les propriétés d'interopérabilité et de flexibilité offertes par CORBA trouvent difficilement des équivalents parmi les autres standards.

Les travaux au sein de l'OMG autour du MDA et des profils UML pour les systèmes temps réel et tolérants aux fautes fournissent des éléments de solution à certains problèmes résolus par les architectures génériques comme la configuration et l'adaptation de l'intergiciel en fonction des besoins en tolérance aux fautes. En outre, les propriétés d'interopérabilité et de flexibilité offertes par CORBA trouvent difficilement des équivalents parmi les autres standards. Cette section justifie le choix de FT CORBA comme cas d'étude pour la tolérance aux fautes. Le paragraphe suivant est dédié à l'étude des différentes stratégies pour mettre en oeuvre FT CORBA et de manière plus générale, pour intégrer un support de la tolérance aux fautes dans les architectures CORBA.

2.3.2 Tolérance aux fautes et réplication dans CORBA

Dans ce paragraphe, nous nous intéressons aux différentes approches et pour l'intégration de mécanismes de tolérance aux fautes dans les architectures CORBA. D'abord, nous présentons les différentes stratégies d'intégration de la tolérance aux fautes en général et de FT CORBA en particulier. Nous nous intéressons ensuite aux travaux dépassant la simple intégration de la tolérance aux fautes pour intégrer d'autres besoins comme le comportement temps réel et les performances d'une part et d'autre part l'expression des besoins et la configuration de la tolérance aux fautes.

2.3.2.1 Architectures

Les tentatives d'implantation de services de tolérance aux fautes pour les applications CORBA ont commencé longtemps avant la publication de FT CORBA. En 1996, Maffeis propose déjà un patron de conception appelé Object Group [80] définissant un schéma de réplication active sur un ensemble d'objets CORBA. Depuis, les tentatives d'intégration de mécanismes de tolérance aux fautes dans l'architecture CORBA se sont multipliées aboutissant à l'adoption de FT CORBA.

Felber [44] classe les stratégies d'intégration de la tolérance aux fautes en trois approches : service, intégration et interception. Ce classement se base sur la position des mécanismes de tolérance aux fautes dans l'architecture de l'intergiciel. L'approche par intégration, parfois appelée approche explicite consiste à modifier l'ORB. L'approche par interception consiste à définir des intercepteurs au niveau des couches basses de l'intergiciel ou même au niveau du système opératoire. L'approche basée sur les services consiste à implanter les services de tolérance aux fautes sous forme de service CORBA.

L'approche par intégration consiste à agir au niveau de l'ORB afin d'implanter les fonctionnalités requises pour la tolérance aux fautes (par exemple mettre en oeuvre un support direct d'un protocole de communications de groupe au lieu du protocole IIOP). Electra [79] consiste

en un ORB modifié afin de supporter les mécanismes de communications fournis par Horus [106]. Orbix [68] agit similairement sur l'ORB afin de d'utiliser le protocole de multicast fiable fourni par Isis [12]. Des travaux plus récents continuent à utiliser cette approche comme par exemple [130]. Les principales modifications visent le support de protocoles de communications de groupe avec certaines propriétés.

L'approche service consiste à implanter l'infrastructure de tolérance aux fautes sous forme d'un service CORBA (pouvant se résumer à un ensemble d'interfaces IDL et leur concrétisations). Cette approche permet de mettre en oeuvre la tolérance aux fautes sans modifier l'ORB et tout en restant conforme au standard. En contre partie, cette approche présente deux inconvénients. Elle peut nécessiter des modifications dans le code des applications afin d'utiliser le service de tolérance aux fautes. Elle peut également engendrer des problèmes de performance à cause du multi-layering [48]. IRL [8] et OGS [43] implémentent FT CORBA selon cette approche.

La stratégie par interception se base sur la notion d'intercepteurs et favorise les propriétés de transparence. Cette approche ne nécessite de modification ni au niveau de l'ORB ni au niveau de l'application. Les intercepteurs peuvent faire partie de l'intergiciel, comme par exemple les Portable Interceptors de CORBA. Il peuvent aussi intervenir au niveau du système opératoire, c'est le cas de l'intercepteur d'Eternal [87]. L'intercepteur d'Eternal qu'intercepte les appels système (passant par l'API socket, provoqués par les messages IIOP et crées par l'ORB). L'interception au niveau du système opératoire cause un problème évident d'interopérabilité et de portabilité. L'utilisation d'intercepteurs portables permet d'éviter ce problème mais certaines limitations([83]) imposent l'utilisation d'objets intermédiaires (passerelles) pour re-diriger les invocations des clients aux différentes répliques, engendrant des problèmes de performances.

La classification de Felber ne prend pas en compte une autre stratégie proche de l'approche par interception et en constitue une généralisation, il s'agit de l'approche réflexive. définit des méta objets capables non seulement d'intercepter les requêtes et les appels systèmes, mais aussi de contrôler le comportement de ces entités. La stratégie réflexive peut être considérée comme une extension de l'approche par interception. Parmi les implémentations basées sur cette approche nous notons DAISY[10].

Dans cette classification, on distingue les approches intrusives (par intégration) des approches non intrusives (approches réflexives, par interception et basées sur les services). Les approches intrusives ne répondent pas aux besoins de transparence, de configuration et d'aptation. Les approches basées sur les services sont généralement pas performantes à cause des nombreuses indirections qu'il présente (multi-layering). Les approches définissant des intercepteurs bas-niveau posent des problèmes de portabilité et d'interopérabilité. Les nouvelles implémentations se basent en grande partie sur les intercepteurs de haut niveau (comme les intercepteurs portables de CORBA) ou alors sur le concept de la réflexion.

2.3.2.2 Comportement temporel

Les performances et la stabilité du comportement temporel des intergiciels tolérants aux fautes sont primordiaux. Nous présentons ici DOORS et MEAD, deux intergiciels compatibles avec CORBA ayant pour objectifs, respectivement, les performances et le comportement temps réel.

DOORS [88] (Distributed Object-Oriented Reliable Service) est un service de tolérance aux fautes compatible avec FT CORBA. Les auteurs de DOORS appliquent plusieurs techniques de génie logiciel et proposent des patrons de conception pour améliorer les performances des applications basées sur leur infrastructure. Une implémentation utilisant l'ORBTAO, et une évaluation des performances mettent en valeur l'efficacité des optimisations appliquées [89]. Ce projet montre la possibilité de réconcilier des objectifs de tolérance aux fautes et de performances. Malheureuse-

ment les aspects temps réel ne sont pas traités par DOORS.

Dans la littérature, le travail le plus intéressant concernant la réconciliation et l'établissement de compromis entre la tolérance aux fautes et les aspects temps réel est celui du projet MEAD [86]. Narasimhan part des deux standards FT CORBA et RT CORBA et analyse les conflits entre ces deux standards. Les défis à relever sont nombreux, en particulier l'ordonnancement des tâches pour satisfaire simultanément les échéances et la cohérence des états des répliques est très difficile. En plus, il est assez problématique d'estimer les temps d'exécution au pire cas ainsi que mes temps de détection et de recouvrement. MEAD adresse justement ces problèmes et permet la configuration de propriétés de tolérance aux fautes selon plusieurs critères comme la définition d'échéances sur l'exécution de services répliqués. MEAD définit également un gestionnaire de ressources distribué. En se basant sur les ressources disponibles, le système peut négocier certains paramètres de qualité de service surtout en cas de défaillances. Le travail de MEAD est intéressant car c'est le premier projet qui s'intéresse à la réconciliation des spécifications RT CORBA et FT CORBA. Cependant, cette réconciliation n'est pas encore complète. Le support de la configuration des propriétés de tolérance aux fautes est également limité. En outre, le support des besoins spécifiques d'applications critiques comme les restrictions sur les types de données et les profils de concurrence ne sont pas supportés.

2.3.2.3 Adaptation, configuration et modélisation

Parmi les projets s'intéressant à l'adaptation des services de tolérance aux fautes au conteste d'exécution nous notons AQuA et AFT-CORBA. L'architecture d'AQuA (Adaptive Quality of Service Availability) [105] répond à plusieurs besoins d'adaptation des mécanismes de tolérance aux fautes. Cette architecture permet aux développeurs de définir des besoins comme la disponibilité. AQuA se base sur QuO pour la spécification des besoins de l'application, de la boite à outils Ensemble pour les communications de groupe ainsi que sur un gestionnaire de qualité de service (Proteus) pour la configuration du système en fonction des besoins et des défaillances. AQuA ne supportent qu'une configuration statique, pendant la phase de la compilation. D'autres systèmes comme AFT-CORBA[30] permettent de modifier dynamiquement certaines propriétés de tolérance aux fautes comme par exemple le style de réplication.

Pour les aspects de configuration, Bondavalli propose une technique de transformation de modèles permettant l'automatisation des analyses de sûreté de fonctionnement en partant de modèles UML [15]. Cette approche fait partie des premières permettant des analyses haut niveau de la sûreté de fonctionnement. Dans la continuité de ce travail, [81] présente un travail particulièrement intéressant permettant de configurer les applications se basant sur FT CORBA en fonction de besoins de l'application. Cette approche se base également sur des modèles UML, et définit un ensemble de règles de transformation permettant la construction de réseaux de Petri stochastiques à partir desquels il est possible d'estimer plusieurs attributs de sûreté de fonctionnement.

Polze définit un canevas et une boite à outils permettant la génération de services de tolérance aux fautes en fonction des réels besoins en sûreté de fonctionnement [98] . Les aspects non-fonctionnels de l'application sont spécifiés en utilisant une interface graphique. Il est ensuite possible de définir la technique de tolérance aux fautes utilisée, le comportement temporel d'un composant (temps d'exécution moyens et temps d'exécution au pire cas) ainsi que les aspects de vote. Même si les aspects de vote sont laissés à la charge du fournisseur de l'intergiciel. Cette approche a l'avantage de supporter la configuration automatique de services

2.3.3 Synthèse

La problématique du support de mécanismes génériques tolérants aux fautes dans les architectures CORBA trouve plus d'une solution. Si les premières tentatives visent un support simple de la réplication, plusieurs autres besoins plus spécifiques comme l'adaptation, la configuration et le support de contraintes temporelles ont été progressivement adressés. FT CORBA est un standard laissant plusieurs détails d'implémentation à la charge du fournisseur de l'intergiciel et ne fournit donc pas suffisamment de détails pour la mise en oeuvre de la plupart des mécanismes de tolérance aux fautes qu'il offre. En particulier, les détails de la mise en oeuvre de la détection de défaillances et de la gestion de la cohérence des états des répliques font défaut. Les travaux de recherche autour de FT CORBA sont nombreux. Ils visent, d'une part, la satisfaction de plusieurs propriétés comportementales (comportement temporel, adaptation aux changement du contexte d'exécution de l'application cible, etc.) et d'autre part, la configuration automatique ou semi-automatique des services de ce standard en fonction des réels besoins de l'application.

Les différentes études autour de la tolérance aux fautes pour les applications CORBA et autour de FT CORBA montrent un manque de support simultané des besoins d'adaptation et de qualité de service. Les différents projets s'inscrivent dans l'une des catégories mais pas dans les deux. Par exemple, les études sur la modélisation des besoins et la génération automatique d'applications tolérantes aux fautes sont rarement et souvent pas accompagnées par des études sur le comportement temporel des applications résultantes. En outre, les applications développées chez les industriels servent à satisfaire des objectifs fonctionnels et non-fonctionnels à la fois immédiats et difficiles à satisfaire à cause des différentes contraintes (environnement de développement, exigence de certification, comportement temps réel, etc.). Ce genre de projets s'intéresse rarement aux aspects de génération (semi) automatique.

Les implantations actuelles de FT CORBA confirment cette constatation. Nous trouvons, soit des architectures configurables mais supportant la tolérance aux fautes en sur-couche (sous forme de service), ce qui pose un grand handicap lorsqu'il faut répondre aux exigences de qualité de service. Nous trouvons encore intergiciels pouvant offrir des garanties sur la qualité de service mais qui sont très peu adaptables et configurables, à cause d'architectures trop intrusives mélangeant les fonctions essentielles de l'intergiciel et le support de la tolérance aux fautes, ce qui limite la réutilisation.

Nous nous proposons de réconcilier les exigences de qualité de service et de réutilisation. Nous partons de l'architecture schizophrène offrant des capacités de configuration importantes et nous introduisons le support de la tolérance aux fautes tout en garantissant plusieurs propriétés de réutilisation et de comportement temporel stable. Nous étudions l'apport de l'architecture schizophrène pour la mise en oeuvre de FT CORBA dans le prochain chapitre.

2.4 Support de l'abstraction du consensus par les intergiciels

Le consensus est un problème générique pouvant être utilisé pour le développement de systèmes distribués tolérants aux fautes en général, et en particulier pour la mise en oeuvre de certains mécanismes spécifiés par FT CORBA comme les différents styles de réplication. Dans ce paragraphe, nous nous intéressons, d'une part à l'utilisation pratique des algorithmes de consensus et d'autre part aux approches d'intégration du consensus dans les architectures logicielles. Dans la littérature nous trouvons plusieurs travaux concernant la théorie du consensus. Même si la contribution de ces travaux à la compréhension et à l'évolution de ce domaine est indéniable et même essentielle (résultats d'impossibilités, algorithmes de réduction, etc), les aspects pratiques de l'utilisation du consensus semblent être moins traités par les projets de recherche.

Dans un premier temps, nous étudierons les travaux faisant le lien entre la théorie et la pratique du consensus. Les travaux les plus intéressants dans ce registre sont ceux qui font le lien entre les modèles abstraits de défaillances et de calculs (voir paragraphe 1.4.3) et les systèmes réels d'une part, et ceux qui s'intéressent au comportement temporel de ces algorithmes (mesures de performances, simulations, etc.) d'autre part. La première partie de cette section s'intéresse à ces travaux. Nous nous intéressons ensuite aux architectures logicielles et aux intergiciels supportant l'abstraction du consensus.

2.4.1 Consensus, détection de défaillances : de la théorie à la pratique

Les études faisant le lien entre la pratique et la théorie du consensus, bien que peu nombreux, sont très importants. Ce sont ces travaux qui permettent aux industriels de profiter des résultats autour de la théorie du consensus.

Les analyses des comportements temporels d'algorithmes de consensus restent très demandés. Dans ce sens, nous trouvons [2] qui étudie, d'un point de vue théorique et pratique, l'impact de la qualité des détecteurs de défaillances sur la terminaison du consensus. Les études pratiques sont généralement faites grâce à des mesures de performance. Les aspects temporels du consensus ont également été analysés grâce à une modélisation se basant sur les réseaux de Petri stochastiques [116].

Les choix des modèles de calcul et de défaillances ont également fait l'objet de plusieurs études. A travers la notion de la couverture des hypothèses (assumption coverage), [99], explique par exemple l'importance du choix du modèle de défaillances et son impact sur la sûreté de fonctionnement globale de l'application finale.

La couverture des hypothèses est un concept reliant les modèles de défaillances théoriques aux défaillances réelles que peut observer le système pendant son fonctionnement. Cette notion établit un lien entre le mode de défaillance choisi pour un composant et la sûreté de fonctionnement des systèmes utilisant ce composant. Il est important de noter que le système tombe en panne si les hypothèses faites pendant la phase de conception ne sont pas vérifiées à l'exécution. Comme noté dans [99] cette hypothèse incite à choisir le bon modèle de défaillances. En effet, l'adoption d'hypothèse trop pissimistes comme le modèle byzantin n'est pas forcément le meilleur choix. Par exemple, la redondance doit être augmentée selon la gravité des défaillances supposées. Or la mise en oeuvre de la redondance implique des coûts supplémentaires et augmente les sources de défaillances pouvant causer une régression de la sûreté de fonctionnement globale du système. Powell et al. montre en particulier que le choix du modèle de défaillances byzantines n'est pas forcément le meilleur moyen de garantir la sûreté de fonctionnement.

Pour les modèles de calcul, [77] montre l'intérêt du modèle asynchrone aux systèmes temps réel. Il sépare entre les modèles de "conception" ou de production des algorithmes et le modèle de l'"implémentation" ou la mise en oeuvre effective de l'algorithme. Il montre que le choix d'un modèle asynchrone est judicieux pour concevoir les algorithmes dédiés aux systèmes temps réel et qu'il vaut mieux retarder l'immersion de l'algorithme dans le système réel (principe de late binding). Des études comme [61] montrent l'intérêt du modèle asynchrone avec détecteurs de défaillances à la résolution pratique du consensus dans les environnements temps réels.

2.4.1.1 Consensus

Le choix pratique d'un algorithme de consensus pour un système réel ne dépend pas seulement du modèle de défaillances et de calcul que doit vérifier le système. Le comportement temporel et les ressources consommés sont aussi importants. En particulier, la propriété de terminaison

assurant que les processus décident inévitablement doit être raffinée pour les systèmes réels. Pour ces systèmes, il est souhaitable de définir une borne supérieure limitant le temps de toute décision du consensus. Par exemple, dans [61] on parle de "temps de terminaison au pire cas" (worst case termination time) pour les algorithmes distribués en général et pour le consensus en particulier.

L'étude des performances des algorithmes de consensus est généralement faite à travers des mesures de complexité. D'une manière générale on se base sur le nombre de tours ou sur le nombre de messages échangés pour évaluer le comportement des algorithmes de consensus. Ces mesures ne sont pas suffisantes surtout pour les applications dont le fonctionnement dépend du comportement temporel du consensus.

Pour pallier cette insuffisance [115] propose d'utiliser une modélisation en réseaux de Petri stochastiques et sur une simulation sur ces modèles afin d'avoir une idée précise sur le comportement temporel des algorithmes de consensus. Les implémentations pratiques des algorithmes est elle aussi un moyen efficace pour évaluer le comportement temporel des algorithmes. Par exemple [58] compare les implémentations de deux algorithmes de consensus. La difficulté de la mise en oeuvre de la distribution, et l'impossibilité de résoudre certaines problèmes pratiques comme la synchronisation d'horloges découragent l'implémentation effective des algorithmes si le seul objectif est l'évaluation du comportement temporel de ces algorithmes.

L'intérêt du modèle asynchrone et les résultats d'impossibilité dans ce modèle motivent l'utilisation du modèle de défaillances avec détecteurs de défaillances. [116] montre l'utilité de ce modèle même lorsque les systèmes imposent des contraintes temporelles sur la terminaison du consensus. Cette étude montre que le choix du modèle asynchrone avec détection des défaillances est compatible avec une prise en compte des contraintes sur le temps de convergence et que les caractéristiques des détecteurs de défaillances permettent d'abstraire les problèmes liés au temps dans ce modèle. Le paragraphe suivant montre l'importance des détecteurs de défaillances et traite les aspects pratiques de leurs mises en oeuvre.

2.4.1.2 Détection de défaillances

Dans les systèmes asynchrones avec détection de défaillances, le comportement temporel des algorithmes de consensus dépend de celui de des détecteurs de défaillances. [116] montre l'impact du comportement du détecteur de défaillances sur le temps de terminaison du consensus pour les exécutions avec et sans défaillances. Elle montre également la difficulté de la tâche de choisir le meilleur détecteur de défaillances à travers les compromis à faire. Les auteurs de ce papier proposent une conclusion très intéressante dans le contexte de notre thèse : il est possible de découpler les aspects logiques (conception de l'algorithme, preuves) des aspects temporels (implémentation effective des algorithmes et surtout mise en oeuvre des détecteurs de défaillances). [2] se base sur un modèle synchrone et montre qu'un service de tolérance aux fautes efficace permet d'accélérer la terminaison du consensus et peut être utilisé pour mise en ouvre de systèmes critiques nécessitant un comportement temps réel.

Pour les applications présentant des exigences temporelles, les détecteurs de défaillances garantissant (uniquement) la détection inévitable ne sont pas suffisants. Ces applications ont besoin de détecteurs de défaillances fournissant des garanties sur les temps de détection. [25] définit un ensemble de métriques permettant de spécifier la qualité d'un détecteur de défaillances dans les systèmes ou les délais et les pertes de messages suivent des distributions probabilistes. Ces métriques définissent par exemple la vitesse avec laquelle les processus défaillants sont détectés et de quelle manière les faux positifs sont évités. Ce papier propose ensuite un algorithme de détection dont les paramètres peuvent être ajustés en fonction des exigences sur le comportement des détecteurs de défaillances. Ce travail est peut être le premier qui étudie la qualité de service

des détecteurs de défaillances et se basant sur la théorie des probabilités.

L'étude du comportement temporel des algorithmes de détection de défaillances se fait généralement grâce à des mesures de performances ad-hoc ou à des simulations [115, 25].

2.4.1.3 Conclusion

L'importance du consensus dans les environnements distribués motive la multiplicité des travaux permettant sa formalisation et la définition de modèles et d'abstractions permettant de le résoudre. Les travaux mettant en pratique les résultats théoriques sont moins nombreux. Ceci peut être expliqué par plusieurs facteurs : d'une part, la problématique du consensus semble mal comprise par les non spécialistes [52]. D'autre part, les problèmes posés par les environnements distribués comme la difficulté d'accéder à un temps global et la complexité de la programmation de ces environnements ne favorisent pas la réalisation de solutions mettant en pratique la théorie du consensus.

Cependant, depuis les années 2000, les travaux étudiant le comportement temporel et proposant des architectures se basant sur le consensus ou permettant sa mise en oeuvre commencent à voir le jour. Dans le prochain paragraphe, nous nous intéressons aux architectures intergicielles proposant un service de consensus ou à celles se basant sur cette abstraction pour résoudre d'autres problèmes de la distribution.

2.4.2 Patrons et architectures pour le consensus

La mise en oeuvre de l'abstraction du consensus a été l'objectif plusieurs architectures. Cette abstraction a été utilisée pour permettre l'atteinte d'un accord malgré les défaillances dans des environnements distribués. Le but d'étudier les architectures se basant sur cette abstraction.

2.4.2.1 MAFTIA

MAFTIA (Malicious and Accidental Fault Tolerance for Internet Applications) [122] est un projet européen dont le but est de répondre au besoin de tolérer les fautes malicieuses et les défaillances accidentelles dans les systèmes distribués à large échelle. Ce projet présente des discussions intéressantes sur les modèles de calculs, de défaillances et les politiques de gestion des groupes à adopter pour résoudre des problèmes pratiques. Il adopte un modèle asynchrone et tolère les défaillances byzantines.

L'architecture de MAFTIA propose une architecture claire permettant, d'une manière progressive, de construire des couches de plus en plus fiables. L'architecture de MAFTIA se base sur le consensus pour assurer un ensemble de fonctionnalités comme l'élection d'un leader, la réplication, la gestion des transactions et des bases de données distribuées, etc. Cependant les aspects de configuration dans MAFTIA sont limités. En plus, cet intergiciel étant destiné pour le support des transactions sur Internet ne répond pas aux exigences de comportement temporel et de consommation de ressources que nous nous fixons pour notre étude.

2.4.2.2 Thema

Thema est un intergiciel permettant de développer des services Web tolérants aux fautes selon le modèle trois tiers. Son architecture consiste en trois bibliothèques, une utilisable par le client (Thema-C2RS), une seconde permettant la tolérance aux fautes byzantines (Thema-RS) ainsi qu'une troisième pour les appels de services externes (Thema-US). Thema se base sur SOAP et UDP comme protocoles de communication.

Thema se base sur la réplication pour la tolérance aux fautes aux fautes byzantines. Les concepts architecturaux de Thema sont trop liés au modèle trois tiers et donc difficilement adaptables aux contraintes des systèmes critiques temps réel.

2.4.2.3 Bast

Bast [50] est une librairie fournissant un ensemble de protocoles pour le développement de systèmes tolérants aux fautes. Bast définit un ensemble de composants fournissant chacun une solution a un problème d'accord spécifique comme le consensus et la diffusion atomique. Bast se base sur le patron STRATEGY pour la définition et la composition des différents protocoles. Les composants définis par Bast fournissent des services de communication, d'invocation à distance, de consensus, de détection de fautes, ainsi que pour certains autres problèmes d'accord comme les transactions atomiques et l'ordre total.

Bast permet la résolution de problèmes d'accord courants et très utiles dans un contexte réel. Bien qu'il se base sur une idée originale permettant d'abstraire les protocoles de les composer d'une part et facilitant l'extension de la libraire d'autre part, Bast souffre de plusieurs limitations. Par exemple, Bast suit une approche boite blanche permettant la réutilisation des implémentations mais elle est inadaptée pour la flexibilité et la transparence. Bast ne supporte pas l'hétérogénéité, par conséquent, ses capacités à supporter des intergiciels comme CORBA sont nulles. En outre, les auteurs ne fournissent pas d'indications sur le comportement temporel et sur la consommation des ressources des différents composants.

2.4.2.4 OGS

OGS (Object Group Service) [43] propose une architecture permettant la réplication d'objets CORBA en se basant sur l'abstraction du consensus. OGS se définit par un ensemble d'interfaces IDL spécifiant plusieurs services de consensus, de détection de défaillances et d'accord distribués tolérants aux fautes. Ces services sont définis sous forme de composants indépendants les uns des autres. Les services définis par OGS s'inspirent de ceux proposés par Bast. Plusieurs concepts introduits par OGS ont été repris par la norme FT CORBA (groupes d'objets, styles de réplication, etc.).

OGS dépend fortement du protocole IIOP et de la norme CORBA. Même s'il permet le développement d'applications tolérantes aux fautes, les besoins des applications critiques ne semblent pas être prises en compte. Par exemple, OGS ne prend pas en compte les problèmes de comportement temporel et de consommation des ressources.

2.4.2.5 CORE/SONiC

CORECOnsensus for REsponsiveness fournit un ensemble de protocoles, de services et de stratégies d'ordonnancement au niveau du micro noyau. Ces services se basent sur le consensus et sont dédiés au calcul parallèle. Le consensus est utilisé pour la détection de défaillances et pour l'établissement de vues globales du système. CORE met en oeuvre une couche de communication fiable au dessus de SONiC [97] et utilisant les services du micro-noyau Mach. Le modèle CORE/SONiC permet l'exécution d'un ensemble de tâches parallèles et déployées sur des unités de calculs inter-connectées. Plusieurs politiques d'ordonnancement comme RMS et EDF peuvent être utilisés. Ce modèle a été étendu aux environnement CORBA [96].L'unité basique de réplication et d'ordonnancement dans le modèle proposé est la requête CORBA.

Même si ce modèle ne supporte pas certains mécanismes fondamentaux comme la réplication, il a plusieurs points forts comme le support de différentes politiques d'ordonnancement et la couche d'adaptation pour permettant l'utilisation du consensus par les applications `CORBA`.

2.4.2.6 `AspectIX`

`AspectIX` est un intergiciel basé sur les aspects, il dispose d'une interface générique permettant le contrôle de certaines propriétés non fonctionnelles de services distribués. Le concept des aspects permet à cet intergiciel de répondre à des besoins de généricité, d'isolation et de composition. Dans `AspectIX` il est possible de contrôler la cohérence, la réplication et la mobilité [104]. `AspectIX` propose un service de communication de groupe et se base sur un service de consensus pour obtenir des propriétés d'ordre total. Le service de communications de groupe dispose de quelques dimensions de généricité, il permet par exemple le choix de la variante de l'algorithme du consensus, il supporte plusieurs politiques de gestion de groupe.

L'architecture d'`AspectIX` prévoit un module pour le consensus et permet le choix entre les différents algorithmes. La détection de défaillance ne semble pas être correctement prise en compte. Les contraintes temporelles et la gestion des ressources ne figurent pas dans les objectifs d'`AspectIX`.

2.4.3 Conclusion

Le problème du consensus a été le sujet de plusieurs études théoriques et pratiques. Dans l'ensemble des intergiciels et architectures que nous avons présentés ci dessus, on distingue certains concepts architecturaux importants pour notre problématique.

`Bast` et `OGS` définissent des architectures génériques isolant un ensemble de composants très utiles lors du développement d'applications nécessitant des algorithmes d'accord tolérants aux fautes. `OGS` se base justement sur cette architecture afin de proposer un service de tolérance aux fautes pour les applications `CORBA`. Certains des principes de `OGS` ont d'ailleurs été retenus par `FT CORBA`.

Nous avons également présenté des projets utilisant le consensus pour tolérer les défaillances au sein d'intergiciels dédiés comme `Thema`, et `MAFTIA`. Même si la réutilisation de ces services dans le cadre de systèmes temps réels semble inadaptée, ces projets proposent des réflexions intéressantes sur les services de l'intergiciel et le choix des modèles de calcul et des politiques de gestion des groupes de participants.

`AspectIX` et `CORE` et présentent des architectures répondant partiellement à nos besoins. L'utilisation des aspects dans `AspectIX` lui permet d'avoir certaines dimensions de configurabilité. Néanmoins, il semble ne pas pouvoir supporter les applications critiques temps réel. En outre, on n'a que très peu d'éléments sur l'efficacité de son architecture ni de certains détails d'implémentation importants comme le support et la gestion des tâches concurrentes. `CORE` répond aux besoins des applications temps réel en proposant des algorithmes d'ordonnancement adaptés. Il se base également sur une couche d'adaptation à `CORBA`. La généricité et la portabilité sont les points faibles de cet intergiciel qui dépend fortement du système opératoire.

Dans tous les intergiciels et les architectures que nous avons étudiés, la réconciliation du comportement temporel, de la gestion des ressources et de la réutilisation ne sont que très peu étudiés. Concernant la problématique du consensus, notre objectif est double. D'une part nous nous proposons de prendre en charge les besoins de systèmes critiques et temps réel. D'autre part, nous élargissons le spectre des applications cible de notre architecture en considérant aussi les problématiques de transparence, d'interopérabilité, et de configuration.

2.5 Synthèse

Dans le chapitre 1 nous, avons conclu que la réduction des coûts des applications sûres de fonctionnement nécessite l'intégration de plusieurs aspects possiblement incompatibles comme les contraintes de qualité et de validation, la tolérance aux fautes, les contraintes temporelles, l'adaptation et la réutilisation. La première partie de notre état de l'art s'est intéressée aux architectures et aux propriétés maximisant les possibilités de réconciliation de ces objectifs. Nous nous sommes ensuite intéressés aux architectures logicielles et intergicielles permettant un support efficace de la tolérance aux fautes et du consensus.

2.5.1 Technologies intergicielles

Nous avons exploré les technologies intergicielles les plus connues et montré que `PolyORB` fournit une bonne architecture permettant l'interopérabilité, la configuration et supportant la vérification formelle. Notre objectif est de profiter de son architecture pour mettre en oeuvre des services de tolérance et de consensus utilisables par une large gamme d'applications cibles. Ces services sont en effet indispensables lors de la conception et du développement de plusieurs applications sûres de fonctionnement et/ou critiques.

2.5.2 Tolérance aux fautes

Plusieurs types d'architectures ont été étudiées. Nous remarquons l'existence de deux extrêmes : les architectures assurant un niveau très élevé de sûreté de fonctionnement mais très couteuses et peu réutilisables. Le second extrême n'atteint pas le niveau de sûreté de fonctionnement requis par les applications critiques. `FT CORBA` constitue un juste milieu entre ces extrêmes. D'une part, la réutilisation et l'interopérabilité sont assurées par `CORBA` qui commence à être utilisé efficacement dans certaines applications a fortes demandes en qualité de service. D'autre part, ce standard propose un ensemble de techniques de tolérance aux fautes assez complet. En plus, `FT CORBA` est compatible avec des méthodes efficaces permettant l'expression des besoins et la configuration ou la génération des services de tolérance aux fautes en fonction de ces besoins. Nous avons ensuite étudié les stratégies de mise en oeuvre de `FT CORBA`. Notre implémentation se base sur l'interception, son intégration dans `PolyORB` permet d'avoir des propriétés importantes pour notre objectif comme la compatibilité avec le profil `Ravenscar`, les performances, et la configurabilité des stratégies de tolérance aux fautes.

2.5.3 Consensus

Nous avons étudié les travaux reliant la théorie à la pratique du consensus ainsi que les architectures supportant cette abstraction. Les différentes études montrent qu'il n'existe pas une seule solution répondant à tous nos objectifs de qualité et de réutilisation. Le service de consensus que nous proposons dans le chapitre 4 associe la généricité de l'architecture qui est indépendante des paramètres de déploiement, des protocoles de transport et de détection de défaillances à l'efficacité et au comportement temporel déterministe. Ce service est également compatible avec des profils de restrictions tels que `Ravenscar`.

Deuxième partie

Conception

Chapitre 3

Conception et intégration d'un service de tolérance aux fautes dans une architecture schizophrène

Contents

3.1 Introduction **59**
3.2 Architecture schizophrène **60**
3.2.1 Définitions, objectifs et principes 60
3.2.2 Personnalités applicatives et protocolaires 61
3.2.3 Couche neutre et services canoniques de distribution 62
3.2.4 Avantages de l'architecture 64
3.2.5 Conclusion 65
3.3 Conception et intégration d'un service de tolérance aux fautes dans l'architecture schizophrène **66**
3.3.1 Problématique 66
3.3.2 Mécanismes d'interception dans `CORBA` 70
3.3.3 Mise en oeuvre de la réplication à l'aide des intercepteurs 72
3.3.4 Détection et notification des défaillances 75
3.3.5 Avantages de l'architecture 78
3.4 Conclusion **79**

3.1 Introduction

L'utilisation de l'intergiciel pour le développement d'applications sûres de fonctionnement exige que ce dernier réponde simultanément à divers besoins architecturaux et comportementaux. Nous partons de l'architecture schizophrène et nous étudions l'impact des services de tolérance aux fautes de `FT CORBA` sur les propriétés de cette dernière. Notre objectif est de minimiser cet impact sur cette architecture et de préserver ses différents avantages.

Le choix de l'architecture schizophrène est motivé par ses capacités de configuration, d'adaptation et d'interopérabilité et par son support de la vérification formelle et de certains profils de restrictions. Le choix de `FT CORBA` est justifié par la maturité de ce standard, par les différentes réponses qu'offre `CORBA` lors du support des applications critiques temps réel mais également par

les riches fonctionnalités de tolérance aux fautes proposées (redondances temporelles et spatiales, support de tous les styles de réplication classiques, etc.).

Dans ce chapitre, nous commençons par décrire l'architecture schizophrène ainsi que les avantages qu'elle offre. Ensuite, nous étudions les mécanismes fondamentaux permettant la mise en oeuvre de solutions intergicielles pour la tolérance aux fautes. L'architecture de `FT CORBA` sera analysée afin d'en extraire les principes. Nous isolons deux problématiques : mise en oeuvre des styles de réplication et détection et notification des défaillances. Nous étudions les mécanismes les plus efficaces permettant de mettre en ouvre ces fonctionnalités et nous réduisons l'impact de l'ajout de ces mécanismes sur l'architecture schizophrène.

3.2 Architecture schizophrène

L'architecture de l'intergiciel lui permet de satisfaire simultanément et efficacement plusieurs besoins des applications distribuées qu'il supporte. Ces besoins peuvent être fonctionnels, tels que l'échange de données et l'abstraction des fonctionnalités d'un système opératoire. L'intergiciel permet aussi de satisfaire des besoins non-fonctionnels tels que la mise en oeuvre de plusieurs types de transparence, la tolérance aux fautes, le déterminisme, etc.

L'utilisation de l'intergiciel pour le développement d'applications sûres de fonctionnement définit deux problèmes à résoudre. D'une part, l'intergiciel doit faire le compromis entre plusieurs contraintes pour satisfaire les besoins des applications qu'il supporte [63]. D'autre part, il doit fournir les garanties de sûreté de fonctionnement lors des premières phases de conception mais aussi pendant l'exécution de l'application distribuée.

Comme précisé dans le chapitre 2, l'architecture de l'intergiciel lui permet de répondre à ces deux besoins primordiaux. Dans cette section, nous faisons un tour d'horizon de l'architecture schizophrène. Nous mettons l'accent sur l'apport de cette architecture lors du développement d'applications distribuées sûres de fonctionnement.

3.2.1 Définitions, objectifs et principes

Initialement introduite pour résoudre les problèmes d'interopérabilité entre les modèles de distribution [7], l'architecture schizophrène permet une adaptation rapide aux besoins en distribution d'un large éventail d'applications distribuées. L'architecture schizophrène réutilise le concept de personnalités défini par les intergiciels génériques. Elle découple les aspects applicatifs des aspects protocolaires d'un modèle de distribution. Ce découplage permet une meilleure modularité et facilite la réalisation de passerelles dynamiques entre modèles de distribution [93].

Définition 17 (Personnalité) *Une personnalité se définit par le résultat de la personnalisation d'un intergiciel générique. Par exemple, dans le cas de `Jonhatan`, elle consiste en un systèmes de types, un système de liaisons, une interface de programmation (API) ainsi qu'un ensemble de règles de programmation[34]. En utilisant `Jonhatan`, il est possible d'instancier deux personnalités correspondant à `CORBA` et `RMI`.*

Définition 18 (Architecture schizophrène) *La schizophrénie caractérise chez un intergiciel sa capacité à disposer, simultanément, de plusieurs personnalités afin de les faire interagir efficacement [93].*

L'architecture schizophrène réutilise la notion de personnalités définis par les intergiciels génériques. Plusieurs personnalités peuvent cohabiter et coopérer au sein de la même instance

de l'intergiciel. Cette architecture définit donc un comportement schizophrène dans le sens [124]. Ce comportement favorise la réutilisation et l'interopérabilité entre les modèles de distribution.

L'architecture schizophrène reprend et étend les principes des intergiciels configurables et génériques. Premièrement, comme celle des intergiciels configurables, l'architecture schizophrène permet à l'application de choisir les services et les politiques d'exécution qui correspondent le mieux à ses besoins. Deuxièmement, même si elle reprend le concept de personnalités défini par les intergiciels génériques, elle l'étend en distinguant les personnalités applicatives des protocolaires et en introduisant une couche neutre de point de vue de la distribution. Cette couche permet la coopération et d'interopérabilité des personnalités. Elle se distingue par un taux de factorisation de code supérieur à celui proposé par certains intergiciels génériques tels que `Quarterware` [119] ou `Jonhatan` [34]. Enfin, cette architecture raffine la composition de la couche neutre en distinguant, des services fondamentaux permettant d'abstraire correctement les besoins fonctionnels en distribution que doivent satisfaire les intergiciels.

3.2.2 Personnalités applicatives et protocolaires

L'architecture schizophrène définit deux classes de personnalités : les personnalités *applicatives* et les personnalités *protocolaires.*

3.2.2.1 Personnalités applicatives

Définition 19 (Personnalité applicative) *Une personnalité applicative est la spécification d'une interface entre des objets applicatifs et un intergiciel. Cette interface permet l'accès aux services intergiciels et l'interaction avec d'autres objets applicatifs[100].*

Les personnalités applicatives constituent une couche d'adaptation entre les composants de l'application et l'intergiciel. Une personnalité applicative peut consister en une interface de programmation spécialisée, éventuellement assistée par un générateur de code.

Une personnalité applicative permet aux objets de l'application de solliciter les services de l'intergiciel afin d'interagir avec d'autres objets applicatifs indépendamment des localisations, des langages de programmation et surtout, indépendamment des personnalités. Les fonctionnalités fournies par les personnalités applicatives peuvent être utilisés d'une manière explicite par l'application (cas de `DSA`) ou implicitement par du code automatiquement généré (par exemple les souches et squelettes `CORBA`). Dans un modèle client/serveur, on peut distinguer les fonctionnalités d'une personnalité applicative selon le rôle que joue l'intergiciel :

- Coté client : la personnalité applicative supporte l'émission des requêtes vers l'intergiciel hébergeant l'objet distant.
- Coté serveur : la personnalité applicative permet aux applications d'enregistrer des objets afin de leurs transmettre des données. Généralement, une personnalité applicative fournit un adaptateur d'objets permettant d'associer des identifiants à des objets. Cet adaptateur d'objets est sollicité afin d'acheminer correctement les requêtes à la bonne destination.

3.2.2.2 Personnalités protocolaires

Définition 20 (Personnalité protocolaire) *Une personnalité protocolaire est la spécification d'une interface entre deux intergiciels destinée à l'échange de messages représentant les interactions entre les objets hébergés par ces intergiciels[100].*

Les personnalités protocolaires permettent l'échange de données à travers un canal de communication. Elles fournissent les mécanismes d'établissement et de maintien de connections. Les

personnalités protocolaires permettent la création d'objets de liaison [100]. En particulier, d'instancier une pile de protocoles à partir d'une référence (coté client) ou à partir d'un point d'accès (coté serveur).

3.2.3 Couche neutre et services canoniques de distribution

La couche neutre agit comme une couche d'adaptation entre les personnalités applicatives et protocolaires. Cette couche est indépendante des modèles de distribution et des différentes personnalités. Par conséquent, elle permet la cohabitation et la coopération entre les personnalités de différentes classes.

La figure 3.1 montre une instance de l'architecture schizophrène contenant deux personnalités applicatives (`CORBA` et `DDS`), deux personnalités protocolaires (`GIOP` et `SOAP`) ainsi que la couche neutre (qui est présente dans chaque instance de l'architecture schizophrène).

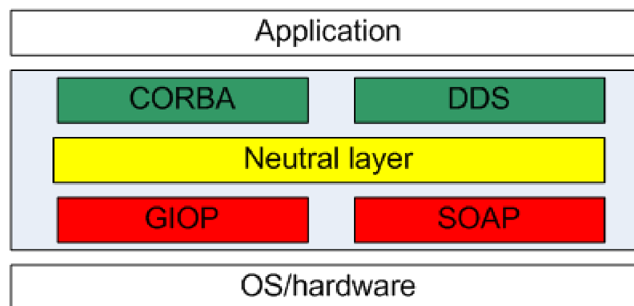

FIG. 3.1 – Architecture schizophrène

Parmi les services offerts par la couche neutre, on distingue sept services fondamentaux pour la distribution. Ces services sont coordonnés par un composant unique appelé μ`Broker`. Le μ`Broker` est un composant configurable fournissant l'abstraction de la boucle de contrôle principale d'un intergiciel.

La figure 3.2 détaille cette vision raffinée de la composition de la couche neutre. Il est important de noter que le μ`Broker` est sollicité à chaque invocation/réception de requête et que ces services fondamentaux sont sollicités pratiquement à chaque invocation de requête. Ils sont dans le chemin d'exécution de toutes les requêtes que ce soit coté client ou coté serveur. Du point de vue validation, l'apport de cette décomposition fonctionnelle est grand : le μ`Broker` puis les services fondamentaux sont les premiers composants à étudier, à consolider et à valider.

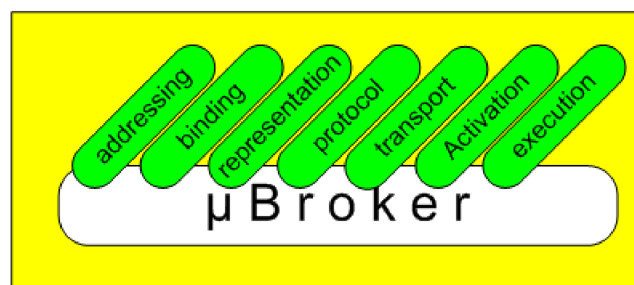

FIG. 3.2 – Services fondamentaux de distribution

Adressage

Le service d'adressage associe à chaque objet une référence qui l'identifie d'une manière unique. Une référence peu designer une entité physique par exemple un objet enregistré auprès d'un ORB (Object Request Broker) mais aussi des entités plus complexes comme les groupes d'objets de FT CORBA.

Une référence peut comporter plusieurs profils. Chaque profil contient des informations pratiques sur l'intergiciel qu'héberge l'objet, l'identité de l'objet au sein de cet intergiciel ainsi que les informations de localisation propres au protocole du transport utilisé (par exemple : adresse IP, numéro de port). Bien entendu, les références sont normalisées par les différents standards.

Transport

Le service de transport permet aux différents noeuds d'échanger des données. Indépendamment du protocole de transport effectif utilisé pour l'échange de données, la couche neutre contient deux abstractions : les points d'accès (transport access points) et les points de terminaison (transport endpoints). Les points d'accès et les points de terminaison sont vus comme sources d'évènements à ordonnancer par le μBroker.

Liaison

Le service de liaison associe à la référence d'un objet distant une entité locale qui le représente et qui permet d'échanger des informations avec lui. Cette entité locale est appelée objet de liaison (binding object). Suivant que l'on se situe du coté client ou du coté serveur, l'objet de liaison peut être crée soit directement à partir d'un profil de l'objet destiné (coté client) ou à partir du point d'accès et d'une pile de protocoles (coté serveur).

Représentation

Pour pallier les problèmes d'hétérogénéité, le service de représentation permet de transformer les données à échanger à travers le médium de communication. Pour des raisons d'interopérabilité les différents noeuds doivent s'accorder sur une représentation commune des données. Les concepteurs d'applications distribuées peuvent être amenés à faire un compromis entre les capacités d'interopérabilité et les performances de l'application, le choix de la représentation des données permet d'appliquer certaines optimisations par exemple dans le cas de similarités entre les types ou entre représentations des machines.

Protocole

Le service du protocole est un service de haut niveau (indépendant de la plate-forme platform independent) permettant aux entités appartenant à plusieurs instances d'intergiciels d'interagir. Ce service définit le format des messages à échanger, la manière de transformer ces messages en données facilement transmissibles à travers le médium de communication (il peut à cet effet utiliser une instance particulière du service de représentation). Dans le cas d'un appel de procédure distante, il permet de créer un message contenant le nom de la procédure à invoquer ainsi que ses paramètres, d'envoyer ce message, de bloquer jusqu'à la réception d'un message réponse et enfin, de retrouver le résultat (éventuel) à partir de ce message. Selon le modèle de distribution, on trouve des services protocole plus ou moins avancés, le plus connu étant le protocole GIOP défini par le standard CORBA.

Activation

Le service d'activation détermine, à la réception d'un message l'entité destinataire du message. Ce service doit pouvoir gérer le cycle de vie d'identifiants d'entités enregistrées auprès de l'intergiciel. Dans le modèle d'objets distribués, il est aussi responsable de retrouver l'implantation (`servant`) associée à une référence ou d'en créer une nouvelle en cas de besoin.

Exécution

Ce service associe les ressources nécessaires à l'exécution de requêtes dans un modèle d'objets distribués. Par exemple, il permet d'associer un processus léger (`thread`) à l'exécution d'un appel de méthode. Plusieurs politiques d'exécution peuvent être envisagées. Ces politiques déterminent la manière d'utilisation, d'allocation ainsi que les moments de création de ressources d'exécution à chaque arrivée d'une requête coté serveur. Le choix de ces politiques permet d'optimiser les traitements en fonction des ressources physiques dont dispose l'intergiciel. Selon la nature de l'application cible, elles doivent satisfaire des propriétés plus ou moins contraignantes de vivacité (par exemple : toute requête finit par être traitée) et de sûreté (par exemple : absence d'interblocages lors de traitements de requêtes).

3.2.4 Avantages de l'architecture

Comme précisé dans les paragraphes précédents, l'architecture schizophrène se caractérise par plusieurs principes architecturaux importants : extensions des intergiciels génériques par la définition d'une couche neutre et le découplage entre les aspects applicatifs et protocolaires dans les personnalités, définition de services canoniques de distribution et configurabilité des services. Dans ce paragraphe, nous mettons l'accent sur les avantages qui découlent de ces principes.

3.2.4.1 Généricité

La couche neutre agit comme une couche d'adaptation entre les divers modèles de distribution. Dans l'architecture schizophrène, elle se caractérise par un taux de factorisation de code très supérieur à celui proposé par certains intergiciels génériques tels que `Jonhatan`. Ceci a un avantage significatif du point de vue réutilisation.

Le principe de découplage entre les aspects protocolaires et applicatifs d'un modèle de distribution, couplé avec la définition de la couche neutre présente au moins deux avantages. D'une part, en appliquant le principe de la séparation des préoccupations, il permet de mieux satisfaire les besoins en distribution des applications. En effet, il devient tout à fait possible de définir de nouveaux modèles de distributions "sur mesure". D'autre part, il facilite l'interopérabilité entre ces modèles de distribution, ce qui permet, par exemple, de concevoir des applications indépendamment de l'intergiciel ou de réutiliser les composants d'un intergiciel dans un autre [7]. Plus généralement, il permet d'adapter les modèles de distribution aux besoins de l'application et non l'inverse.

3.2.4.2 Configurabilité des services

La configurabilité des services permet de choisir d'une manière cohérente (c'est à dire en résolvant les conflits éventuels), les mises en oeuvre d'abstractions nécessitées par les applications. La séparation entre les interfaces et leurs implémentations permet de mieux satisfaire les besoins des applications et de mieux réagir à de nouveaux besoins. Cette configurabilité peut s'obtenir

en appliquant certains patrons de conception par exemple pont (bridge) ou façade (facade) [49]. L'application systématique de ces patrons permet de clarifier l'architecture en dégageant les abstractions et en assurant un contrôle plus fin sur les interactions au sein de l'intergiciel, ce qui augmente significativement les possibilités d'évolution et de configuration.

3.2.4.3 Services fondamentaux

La couche neutre contient sept services fondamentaux de distribution. La définition de ces services permet une meilleure compréhension de l'architecture. Notons que la définition de ces services est tout à fait compatible avec l'application des principes et patrons de conception favorisant la configurabilité. Ceci permet une meilleure optimisation de ces services en fonction des besoins de l'application. Par exemple, dans [119], les auteurs constatent que les points de variations dans les fonctionnalités d'un intergiciel générique peuvent être isolés dans six composants chacun responsable d'une fonction : représentation, adressage, transport, protocole, aiguillage, invocation (Data Marshalling and Unmarshaling, Object References, Data Transport, Dispatching, Invocation Policy, Wire Protocol). Cette décomposition leur a permis de mettre en place des mécanismes génériques permettant la mise en oeuvre de plusieurs optimisations de performances et d'empreintes mémoire.

3.2.4.4 Vérification comportementale

La définition des services fondamentaux de l'intergiciel et l'isolation du comportement d'un intergiciel dans un seul composant (le μ`Broker`), a permis de mettre en oeuvre une modélisation formelle utilisant les réseaux de Petri [63]. Certaines propriétés (symétrie, absence d'interblocages, cohérence et équité) d'instances de l'architecture schizophrène ont été vérifiées sur les modèles obtenus en utilisant des techniques de model checking. Ce travail montre que, grâce à l'architecture, il est possible de réduire les efforts de validation d'instances d'intergiciels. Ce travail se focalise sur la modélisation du comportement du μ`Broker` en assimilant les services fondamentaux de l'intergiciel à un ensemble de filtres.

3.2.5 Conclusion

L'architecture schizophrène se distingue par trois principes importants. D'abord, elle sépare les aspects applicatifs des aspects protocolaires dans un modèle de distribution. Ensuite, elle définit une couche neutre offrant des services fondamentaux de distribution ainsi que d'autres services utilitaires. Cette couche neutre permet la cohabitation des personnalités et constitue une base de briques logicielles réutilisables, facilitant le prototypage et l'implémentation de nouvelles personnalités. Enfin, la définition des services canoniques de distribution et l'isolation du comportement de l'intergiciel dans un seul composant facilite la validation d'instances d'intergiciels et assiste les éventuelles analyses de sûreté de fonctionnement.

Cette architecture présente plusieurs avantages. En particulier, elle permet de garantir des propriétés d'interopérabilité et de personnalisation de l'intergiciel en fonction des besoins des applications cibles. L'introduction de la tolérance aux fautes dans l'architecture schizophrène étend le champs des applications cibles en lui permettant de répondre aux besoins d'applications plus exigeantes. En revanche, le support de nouvelles fonctionnalités comme la détection des défaillances et la réplication ne doit pas compromettre cette architecture. Dans la prochaine section nous décrivons nos réponses à ce problème.

3.3 Conception et intégration d'un service de tolérance aux fautes dans l'architecture schizophrène

Dans cette section, nous nous proposons d'intégrer les mécanismes de tolérance aux fautes proposés par `FT CORBA` dans l'architecture schizophrène. Ce standard, malgré sa complexité, met en oeuvre plusieurs concepts théoriques et pratiques pour la tolérance aux fautes. Dans cette section, nous commençons par effectuer une analyse architecturale et fonctionnelle de `FT CORBA`. Nous nous intéressons ensuite à la mise en oeuvre de la réplication et aux mécanismes de gestion de fautes. Comme précisé dans la section 2.3.2.1, plusieurs stratégies peuvent être appliquées pour la mise en oeuvre de `FT CORBA`. La stratégie que nous adopterons peut être classée comme basée sur l'interception. Néanmoins, elle évite d'une part, les problèmes de portabilité liées à l'utilisation d'intercepteurs de bas niveau, et d'autre part, les limites imposées par l'utilisation des intercepteurs portables de `CORBA`. Pour ce faire, nous étudions les patrons et les mécanismes permettant l'interception dans `CORBA` et nous proposons une solution qui prend en compte les avantages et les contraintes de l'architecture schizophrène. Notre solution est basée sur une étude de l'architecture de `FT CORBA` et de ces besoins vis à vis de l'intergiciel. Cette étude est le sujet du prochain paragraphe.

3.3.1 Problématique

La spécification `CORBA` a été étendue pour supporter la tolérance aux fautes donnant lieu à `FT CORBA`. Dans ce paragraphe, nous présentons une description des principaux fonctionnalités de `FT CORBA` et définissons les exigences de la norme vis à vis de l'intergiciel. Une description complète de ce standard peut être trouvée dans [91].

3.3.1.1 Architecture de `FT CORBA`

La figure 3.3 donne une vue générale de l'architecture et des principaux modules de l'infrastructure de `FT CORBA`.

Le gestionnaire de réplication (`ReplicationManager`) est responsable de la gestion des cycles de vie, des propriétés et des compositions des différents groupes d'objets qu'il contrôle. Les détecteurs et les notificateurs de défaillances (`FaultDetectors` et `FaultNotifiers`) sont respectivement responsables de la détection et de la propagation des informations à propos des répliques suspectées de ne plus être en mesure de fournir le service attendu. Enfin, les mécanismes de journalisation et de reprise assurent la cohérence entre les états des membres d'un groupe d'objets.

Réplication et gestion des groupes d'objets Le `ReplicationManager` est chargé de la création des groupes d'objets (et dans certains cas les membres d'un groupe d'objets), de la gestion de leurs compositions et de la définition de leurs propriétés du point de vue de la tolérance aux fautes. Le `ReplicationManager` est l'élément le plus central et le plus important dans l'architecture de `FT CORBA`. Il peut constituer un danger pour la sûreté de fonctionnement de l'application globale. Son implémentation nécessite une attention particulière et il doit lui même être répliqué pour ne pas constituer un point unique de défaillance (single point of failure).

Le `ReplicationManager` hérite de trois interfaces `IDL` :

- Gestionnaire des propriétés (`PropertyManager`), permettant la définition et la modification (en respectant les règles imposées par la norme) des propriétés de tolérance aux fautes (par exemple, le style de réplication, le nombre minima de répliques, etc.). Ces propriétés sont

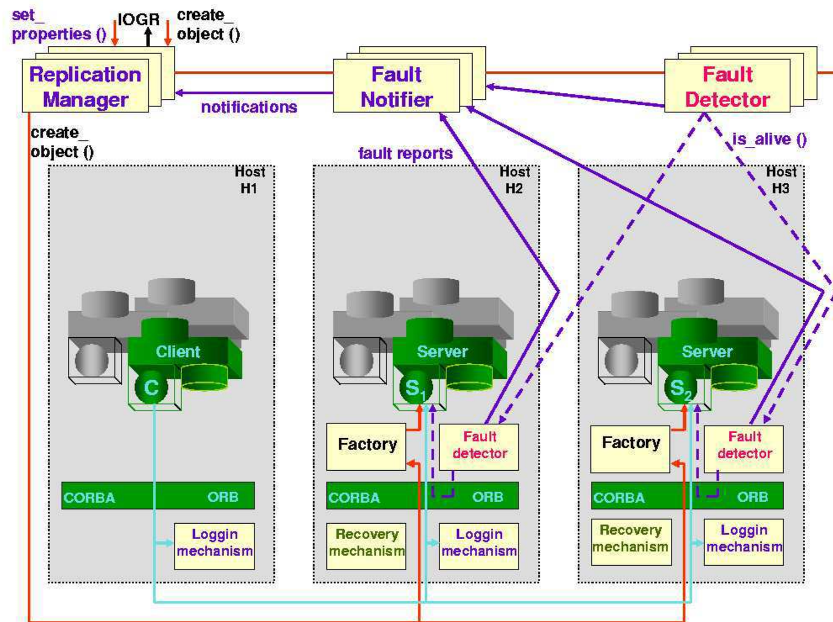

FIG. 3.3 – Architecture de `FT CORBA`

très importantes, elles permettent par exemple de résoudre le compromis entre les ressources utilisées et la sûreté de fonctionnement de l'application basée sur `FT CORBA`. L'impact de la variation de ces propriétés sur la sûreté de fonctionnement du système entier a fait l'objet de certaines études telles que [82].

- Gestionnaire des groupes d'objets (`ObjectGroupManager`), permettant la définition et la modification de la composition des groupes d'objets.
- Fabrique générique (`GenericFactory`), responsable de la création et de la destruction des répliques (si le contrôle de la composition des groupes est délégué à l'infrastructure de tolérance aux fautes) et des groupes d'objets.

Détection et notification des défaillances `FT CORBA` définit trois modules responsables de la détection et la notification des défaillances : les détecteurs, les notificateurs et les analyseurs de fautes (respectivement `FaultDetector`, `FaultNotifier` et `FaultAnalyser`).

Les détecteurs de défaillances sont responsables de surveiller les répliques. Les détecteurs définis par `FT CORBA` sont basés sur la notion de timeout. Les notificateurs sont avertis des pannes des répliques grâce aux rapports de défaillances (fault reports). Le service de notification de `CORBA` [91] peut être utilisé pour assurer les propagations des rapports de fautes. Une fois collectés, les rapports d'erreurs sont propagés au `ReplicationManager` qui prend les décisions de reconfiguration et/ou de reprise adéquates. La norme propose des analyseurs de fautes pouvant effectuer une analyse des rapports de fautes afin d'en générer d'autres plus condensés avant de les propager au `ReplicationManager`. Les analyseurs de fautes sont optionnels.

Gestion du recouvrement Les interfaces Checkpointable et Updatable ainsi que l'infrastructure de journalisation permettent aux répliques d'enregistrer leurs états ainsi que les requêtes qu'elles reçoivent. L'interface Checkpointable propose deux méthodes permettant de récupérer

et de mettre à jour l'état d'une réplique. Updatable peut être vue comme un raffinement de Checkpointable, elle permet une journalisation incrémentale. Ces interfaces permettent la synchronisation des états des répliques au sein du groupe.

3.3.1.2 Besoins vis à vis de l'intergiciel

FT CORBA peut se définir comme la coordination de trois mécanismes : la gestion et le référencement des groupes de répliques, les échanges de requêtes avec des sémantiques plus ou moins complexes et la surveillance de l'état de fonctionnement des répliques (détection/notification de défaillances). Pour chacun de ces mécanismes, nous précisons ci après les exigences de la norme vis à vis de l'intergiciel.

Gestion de la réplication Le ReplicationManager est chargé de la gestion des cycles de vie des répliques et des groupes d'objets ainsi que de la gestion des propriétés des groupes de répliques.

FT CORBA définit des références sur les groupes d'objets : les IOGR (Interoperable Object Group References). Les IOGR sont des références CORBA contenant plusieurs *profils* (voir le service fondamental d'adressage au paragraphe 3.2.3). Pour chaque réplique adressée dans l'IOGR. Pour supporter FT CORBA, l'intergiciel doit pouvoir fournir les mécanismes de construction et de manipulation des groupes d'objets.

Détection et notification des défaillances Les détecteurs de défaillances fonctionnent au dessus de la couche CORBA. Ils utilisent le schéma classique d'invocations pour vérifier l'état de marche des objets qu'ils surveillent. L'intergiciel, fournissant les fonctions de distribution permet les échanges de données résultantes des appels distants des méthodes *is_ alive* que doivent implémenter les répliques.

De plus, surveiller simultanément plusieurs répliques et envoyer les rapports de défaillances oblige le détecteur à effectuer plusieurs activités d'une manière concurrence. La concurrence est nécessaire pour lancer les timeouts et attendre les réponses des répliques. Notons que les appels CORBA sont généralement bloquants, ce qui entraîne une difficulté supplémentaire et un besoin d'avorter certaines requêtes. L'intergiciel fournit les services permettant d'abstraire les systèmes opératoires garantissant ainsi la cohérence de la gestion des ressources et la portabilité des détecteurs. Dans ce cas, l'intergiciel agit comme une interface au système opératoire (host infrastructure middleware selon la terminologie de [112]).

La notification de défaillances consiste à acheminer les rapports produits par les détecteurs de défaillances aux ReplicationManager. Le standard propose une implantation basée sur l'utilisation de fonctionnalités avancées du service de notification de CORBA (CosNotification). L'intergiciel doit donc optionnellement supporter CosNotification. Dans le cas contraire, il est possible d'assurer la propagation des rapports d'erreurs en se basant sur le schéma classique d'invocation des requêtes.

Transparence de la réplication et synchronisation d'états FT CORBA recommande et requiert la *transparence de la réplication*. Un client doit pouvoir appeler un service répliqué de la même manière qu'il aurait appelé un service "classique". Or, un groupe d'objets admet un cycle de vie plus riche que celle d'un simple objet CORBA. En plus, l'invocation d'un service répliqué dépend du style de réplication et nécessite des fonctions supplémentaires comme la maintenance de la cohérence entre les états de répliques (strong replica consistency) et le support des ré-invocations transparentes (transparent re-invocations).

L'intergiciel doit supporter les échanges de requêtes selon les différents styles de réplication, d'une manière efficace. Notons que certains détails d'implémentation reposent sur des fonctions avancées du protocole GIOP (par exemple la location forwarding et la propagation des service contexts).

Support des styles de réplications La norme FT CORBA prévoit cinq styles de réplications. Le premier style concerne la réplication des objets sans état. Les autres styles de réplication représentent le cas le plus général et le plus difficile à gérer à cause du problème de maintien de la cohérence des états des répliques. Dans le paragraphe 1.3.3.2 nous avons présenté la réplication et les styles de réplication active et passive. Les styles de réplication peuvent être classés en trois catégories : réplication sans état, réplications actives (avec et sans vote), réplications passives (à froid et à chaud).

Le style de réplication sans état est appliqué lorsque le service à répliquer n'admet pas d'état propre. Ce style ne nécessite pas de mécanismes particuliers pour la synchronisation des états et est relativement simple à mettre en oeuvre.

La norme propose deux styles de réplication active (réplication active et réplication active avec vote). Pour ces styles, toutes les répliques jouent le même rôle, elles reçoivent toutes les requêtes, les traitent (en mettant à jour leurs états internes) et envoient une réponse au client. Le client peut alors choisir une de ces réponses. Pour fonctionner correctement, ces styles supposent un traitement déterministe des requêtes. Le service fourni par les répliques ne doit donc pas se baser sur des constructions pouvant violer le déterminisme, par exemple, il ne doit pas dépendre du temps, ni se baser sur des tâches concurrentes. Même si l'introduction du mécanisme de vote impose un travail supplémentaire, le principal obstacle à la mise en oeuvre de ces deux styles est la garantie de l'hypothèse du déterminisme (strong replica consistency).

Les styles de réplication passive définissent une seule réplique, appelée *primaire*. Dans le cas de la réplication passive à chaud, les autres répliques, appelées secondaires, sont en état de marche. Le primaire synchronise les états des répliques secondaires périodiquement. Cette synchronisation permet à l'une des répliques secondaires d'assurer la continuité du service si le primaire tombe en panne. La réplication passive à froid met en jeu un support de mémoire non volatile (par exemple de la mémoire persistante ou un disque dur). Le serveur primaire enregistre son état sur ce support à partir duquel l'état des répliques est mis à jour lorsqu'elles sont promues comme primaires. Ces deux styles nécessitent la mise en place de mécanismes pour la journalisation et la restitution des états.

Même si les sémantiques, les cas d'utilisation, et les bases algorithmiques régissant ces différents styles de réplication diffèrent, leur mise en oeuvre pose les mêmes problèmes architecturaux. L'intergiciel doit avoir une architecture adaptée, permettant à l'application de choisir le style de réplication qui satisfait au mieux ses besoins. Il doit aussi fournir les mécanismes nécessaires à une adaptation dynamique permettant à l'application de réagir par exemple à la défaillance de l'un de ces noeuds. Nous traitons ce besoin indépendamment des différents styles de réplication, garantissant ainsi l'extensibilité de l'architecture et augmentant sa capacité à supporter plusieurs logiques de réplication.

3.3.1.3 Conclusion

Pour mettre en oeuvre les mécanismes de distribution, l'intergiciel doit supporter deux familles de fonctionnalités :

- Détection et notification des défaillances. L'intergiciel doit fournir le support nécessaire à l'exécution concurrente et assurer la propagation de rapports de défaillances, par exemple

grâce au service de notification (`CosNotification`).
– Mise en oeuvre de la logique de réplication. L'intergiciel doit permettre la gestion des références sur les groupes d'objets, l'acheminement des requêtes et la synchronisation des états selon les sémantiques des différents styles de réplication.

L'introduction de ces différentes fonctionnalités doit se faire en préservant les propriétés d'adaptabilité, d'interopérabilité et en assurant les exigences de qualité de service. Le reste du chapitre, est consacré à la description de l'intégration de ces mécanismes dans l'architecture schizophrène. Nous nous intéressons tout d'abord aux différentes solutions permettant le support de la réplication dans les intergiciels. Nous décrivons ensuite notre proposition pour intégrer `FT CORBA` dans l'architecture schizophrène.

3.3.2 Mécanismes d'interception dans `CORBA`

Nous avons discuté dans le paragraphe 2.3.2.1 les différentes stratégies d'intégration de la tolérance aux fautes dans les intergiciels `CORBA`. Nous avons présenté plusieurs approches et conclu que la stratégie par interception fait un bon compromis entre la transparence et les performances. Nous discutons ici les différents mécanismes d'interception dans `CORBA`. Ces mécanismes permettent de modifier l'état, le contexte et le comportement des applications `CORBA` d'une manière transparente. Nous comparons les différentes solutions en nous basant sur plusieurs critères et choisissons une solution permettant d'intégrer efficacement `FT CORBA` dans l'architecture schizophrène.

Filtres Les filtres, tels que définis par `Orbix`[12] permettent à l'application d'insérer son propre code pour intercepter les requêtes et les réponses. Les filtres peuvent être insérés dans dix points sur le chemin que parcourt une requête ou une réponse. L'utilisateur peut définir son propre filtre en dérivant la classe Filter. Il est également possible d'insérer plusieurs filtres dans un seul point. Ces filtres peuvent être mis en place du coté client comme du coté serveur et permettent par exemple de rajouter des informations supplémentaires aux requêtes (mécanisme de piggybacking). Coté client, ils peuvent être utilisés par exemple pour rajouter des données concernant l'entité appelante ou pour mettre en place des fonctionnalités de cryptage. Coté serveur, ils peuvent mettre en place un mécanisme pour déterminer et appliquer la meilleure politique d'aiguillage des requêtes (request dispatching).

Mandataires intelligents (Smart Proxies) Les mandataires intelligents peuvent être vus comme la combinaison des fonctions d'un mandataire et d'un intercepteur. Ils permettent d'intervenir sur le flot d'exécution des requêtes pour assurer des traitements additionnels (à ceux prévus par une exécution par défaut). Ce sont des mécanismes propriétaires pouvant remplacer les souches pour mettre en place les mécanismes d'interactions plus évolués comme la négociation de la qualité de service. Un mandataire intelligent peut être dynamiquement chargé et continue à assurer l'interaction avec les objets distants en fournissant la même interface de haut niveau que la souche qu'il remplace. Notons qu'il est possible de modifier le compilateur `IDL` afin de générer automatiquement ces mandataires. Ces mandataires sont supportés par plusieurs implémentations de `CORBA` dont `TAO` et `Orbix`.

Adaptateur d'objets portable `POA` `CORBA` offre la possibilité d'enregistrer deux types d'objets permettant la gestion des servants : les servant locators et les servant activators :

[12] `http://www.iona.com`

- Les servant locators permettent d'attacher deux méthodes : `preinvoke` et `postinvoke`. Comme leurs noms l'indiquent, ces méthodes sont appelées par le POA avant et après l'invocation de chaque requête.
- Les servant activators permettent d'insérer des traitements lors de la création et de la destruction d'un servant.

Ces deux types d'objets permettent l'insertion transparente de traitements additionnels selon les besoins.

Intercepteurs portables CORBA définit deux types d'intercepteurs portables (PI) : les intercepteurs d'IOR et les intercepteurs de requêtes. Les intercepteurs d'IOR permettent l'évaluation des politiques du serveur et leurs addition aux IOR sous forme de `Service Context`. Ces intercepteurs, installés lors de l'initialisation de l'ORB, sont appelés lors de la création d'une référence mais *avant* son exportation. Il est important de noter que ces intercepteurs ne peuvent plus intervenir une fois la référence créée, ce qui aurait pu être utile dans pour la création et la manipulation des références sur les groupes d'objets IOGR. Les intercepteurs de requêtes permettent de mettre des traitements supplémentaires sur le chemin d'invocation d'une requête. Ces traitements peuvent être mis en place pour surveiller ou modifier le comportement de l'invocation d'une manière transparente, sans changer le code de l'application ni celui de l'ORB. Les intercepteurs sont vus comme des boites noires par l'ORB et peuvent être appliqués sans modifier le code de l'application.

Discussion et choix du mécanisme d'interception Le besoin de mettre en place des traitements additionnels permettant de modifier le comportement standard des applications CORBA a été exprimé très tôt après la publication de ce standard. Les différents mécanismes décrits ci dessus présentent chacun un point fort. Les mandataires intelligents se distinguent par leur efficacité. Ils n'introduisent généralement pas de pénalités de point de vue performances ou encombrement mémoire. En revanche, ces mécanismes ne sont disponibles que coté client. Les filtres se caractérisent par la possibilité de les appliquer en série (patron chaîne de responsabilité [128]) et leur efficacité surtout pour la gestion des transactions. Cependant, leur utilisation se limite généralement à l'addition d'informations aux requêtes. Par exemple, ils ne permettent pas de mettre en place des re-directions de requêtes. Le POA et les PI fournissent des mécanismes standards et portables pour l'interception des requêtes. Cependant les mécanismes du POA ne sont disponibles que coté serveur. Les PI souffrent également de certaines limitations [83], en particulier tels qu'ils sont définis par l'OMG, ils ne peuvent pas modifier les paramètres d'une requête ou d'une réponse, il imposent de passer par des passerelles pour implanter FT CORBA.

L'architecture schizophrène encourage le développement de composants neutres du point de vue du modèle de distribution. L'introduction de composants trop spécifiques aux personnalités rend difficile leur généralisation ultérieure. Même si notre objectif n'est pas de proposer une solution permettant une tolérance aux fautes indépendante des modèles de distribution, nous nous efforçons lorsque c'est possible de maximiser les composants neutres par rapport aux modèles de distribution. En particulier, nous continuons à assurer l'indépendance entre les personnalités applicatives et protocolaires, ce point sera traité dans la section 3.3.5.1.

L'interception basée sur le POA ou sur les mandataires intelligents sont trop liés à CORBA ce qui nous empêche de les retenir. L'absence de support de re-directions de requêtes dans les filtres élimine également cette technologie. Nous avons alors décidé de baser notre architecture sur des intercepteurs proches des PI. Les intercepteurs que nous proposons permettent de faire toutes sortes d'opérations d'interception. Nous les décrivons ainsi que l'architecture qui résulte de leur application dans la prochaine section.

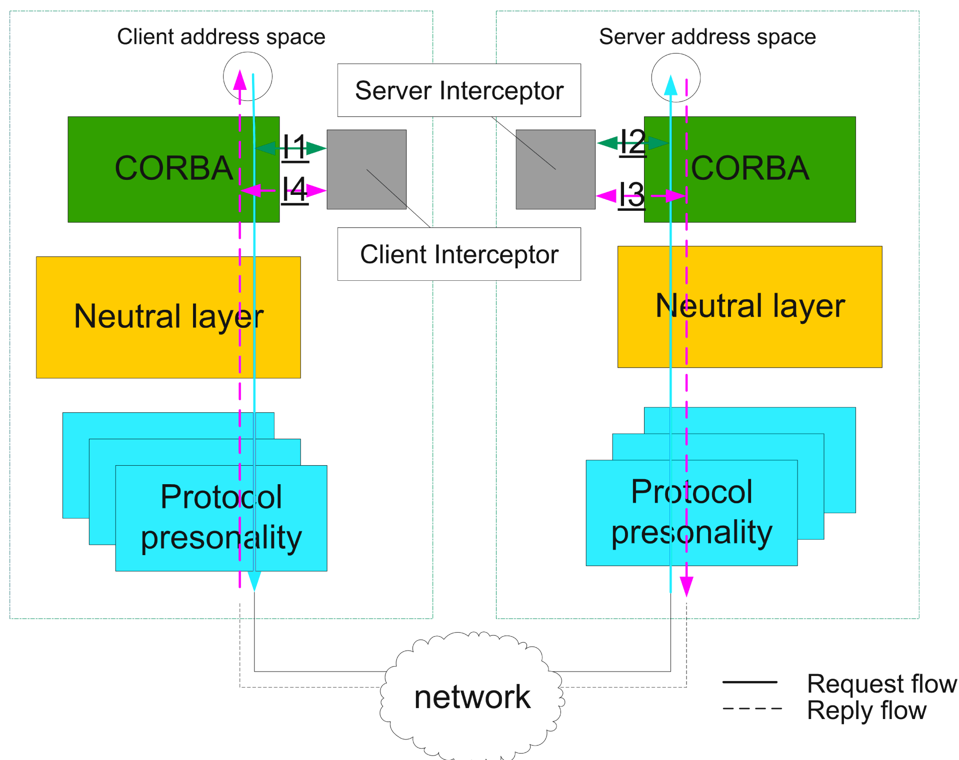

FIG. 3.4 – Positions des intercepteurs dans l'architecture

3.3.3 Mise en oeuvre de la réplication à l'aide des intercepteurs

L'implantation de FT CORBA nécessite une grande attention à l'architecture. L'état de l'art des stratégies d'intégration de FT CORBA montre plusieurs conceptions inefficaces souffrant par exemple de problèmes de performances et portabilité.

L'objectif des intercepteurs que nous proposons est de supporter d'une manière transparente toute la logique de réplication définie par la norme. La compatibilité avec l'architecture schizophrène est le second objectif de ces intercepteurs. Il faut donc mettre en place les mécanismes d'échanges de requêtes, de ré-invocations transparentes et de synchronisation d'états requis par les différents styles de réplication sans violer les règles de l'architecture schizophrène.

3.3.3.1 Architecture

Les intercepteurs que nous avons définis s'inspirent des intercepteurs portables de CORBA. Ils présentent deux différences majeures par rapport à ces derniers. D'une part, ces intercepteurs ne sont pas chargés en utilisant l'interface ORBInitializer. Nous nous basons sur des mécanismes évolués pour permettre leur enregistrement auprès de l'intergiciel indépendamment des interfaces spécifiques à la personnalité CORBA. D'autre part, nos intercepteurs sont autorisés à modifier les champs des requêtes.

Pour chaque style de réplication, nous définissons deux intercepteurs, l'un est installé au niveau des clients, l'autre au niveau des répliques. La figure 3.4 montre la position des intercepteurs dans l'architecture. Ces nouveaux composants appartiennent à la personnalité CORBA et sont considérés comme des membres à part entière de cette personnalité applicative. Ainsi ces derniers n'interagissent qu'avec les autres composants de la personnalité CORBA et la couche

neutre. En particulier, ces derniers n'ont aucune visibilité sur la personnalité GIOP (voir 3.3.5.1).

Pour mettre en oeuvre les différents styles de réplication nous nous basons sur quatre points d'interception. Ils seront notés (I1, I2, I3 et I4). Les intercepteurs sont déployés sur le chemin d'invocation (I1 et I2) et sur celui de réponse (I3 et I4) de chaque requête. Dans les deux prochains paragraphe, nous décrivons les différents traitements appliqués au niveau de ces points. La description se base sur le déploiement des intercepteurs coté client et coté serveur.

3.3.3.2 Intercepteurs coté client

Les points d'interception coté client sont I1 (appelé juste avant d'envoyer les requêtes et I4 (appelé une fois la réponse décodée et juste avant renvoyer la réponse à l'entité appelante).

Le point I1 permet d'analyser les requêtes. Il s'agit ici de déterminer si la référence de la destination est un groupe d'objets. Cette première analyse permet de savoir si une logique de réplication doit être appliquée. Selon le protocole de transport utilisé et le style de réplication, l'intercepteur peut soit invoquer directement la requête soit créer une ou plusieurs requêtes intermédiaires dont il demandera l'envoi à l'ORB. Par exemple dans le cas d'une réplication passive, l'intercepteur cherchera le serveur primaire dans la référence du groupe (en utilisant le Tagged ComponentTag_FT_Group dans le cas de GIOP). Il construit ensuite une requête avec la référence du primaire. Notons que pour respecter les règles de visibilités de l'architecture schizophrène, les Tagged Component sont manipulés sous forme neutre par les intercepteurs. La construction des références se fait également en se basant exclusivement sur des fonctionnalités de la couche neutre.

Le point d'interception I4 est utilisé pour intercepter les réponses des requêtes envoyées après le traitement au point I1. Selon le style de réplication, l'intercepteur peut créer, au niveau I1, plusieurs requêtes à partir d'une seule (par exemple, pour un style de réplication actif avec IIOP comme protocole d'échange de données). Le point I4 permet alors de stocker les réponses à ces requêtes afin de choisir la réponse la plus adéquate. Une fois les traitements appliqués, le résultat est transféré à l'entité appelante en modifiant le paramètre résultat de la requête interceptée au niveau d'I1.

L'introduction de ces points d'interception n'offre aucune visibilité sur le protocole de transport utilisé. Ceci est d'une grande importance lorsque la configuration du groupe d'objets change (par exemple suite à la défaillance d'une réplique). La norme prévoit dans ce cas une exception spéciale (LOCATION_FORWARD_PERM). I4 n'est pas mis au courant et n'applique aucun traitement, puisque le protocole GIOP se charge de renvoyer les requêtes en utilisant la nouvelle référence. Notons que si des exceptions sont retournées par les répliques en réponse aux requêtes, l'intercepteur peut prendre la décision adéquate selon le style de réplication et selon que l'exception reçue est de type exception de basculement (failover exception) ou pas. Certaines exceptions peuvent être simplement masquées au niveau de d'I4, par exemple si la cause de cette dernière est jugée transitoire et si le style de réplication active est mis en oeuvre.

3.3.3.3 Intercepteurs coté serveur

Côté serveur, les logiques de réplication peuvent être appliquées à deux niveaux : au point I2, juste après le décodage de la requête et avant l'appel au servant, et au point I3, une fois le servant invoqué.

Au point I2, l'intercepteur vérifie si le client invoque la requête avec la référence sur le groupe d'objets la plus récente. Si ce n'est pas le cas, une exception LOCATION_FORWARD_PERM contenant la nouvelle référence sur le groupe est levée, l'ORB est sollicité pour renvoyer cette exception au

client appelant. Ces opérations se basent uniquement sur des primitives de la couche neutre. Ce point d'interception permet également de décider s'il faut invoquer ou non la requête. En effet, une requête dupliquée ne doit pas être invoquée puisqu'elle correspond à une demande de service déjà effectuée. L'invocation d'une telle requête met en cause la cohérence des états des répliques. Dans ce cas, l'intercepteur empêche l'invocation de la requête et essaye de retrouver la réponse à partir d'un cache. Le point I2 peut également être utilisé pour voter les paramètres (*in* et *inout*) des requêtes dans le cadre d'une réplication active par exemple. Un protocole de consensus comme celui que nous décrivons dans le prochain chapitre peut être appliqué.

Le point d'interception I3 est utilisé pour mettre à jour le cache de l'intercepteur avec les couples requêtes/résultat permettant ainsi de gérer le cas de duplication des requêtes sans faire d'invocations inutiles. Ce point est également utilisé pour synchroniser les états des répliques. La synchronisation des états des répliques dépend de plusieurs paramètres de configuration du groupe d'objets : style de réplication, période de synchronisation, etc. L'interception coté serveur nécessite l'appel du `ReplicationManager` pour vérifier si la configuration du groupe a changé. Notons enfin que, comme pour les intercepteurs coté client, ces intercepteurs n'ont aucune visibilité sur les personnalités protocolaires.

3.3.3.4 Respect des principes de l'architecture schizophrène

L'introduction de nouveaux composants dans l'architecture schizophrène ne doit pas violer les règles de visibilité définies par cette dernière. En particulier, la séparation entre les aspects applicatifs et protocolaires doit être respectée. Lors de la mise en oeuvre d'un nouveau composant, ce dernier doit appartenir à une personnalité ou alors être placé dans la couche neutre. La séparation entre l'applicatif et le protocolaire n'existe pas dans les spécifications `CORBA`. Plusieurs des fonctions de tolérance aux fautes proposées par `FT CORBA` se basent sur un couplage fort entre l'`API` proposée par `CORBA` et les fonctions d'interaction définies dans `GIOP`. La conception proposée permet de préserver l'architecture schizophrène, d'une part et de produire une architecture claire et facile à faire évoluer pour nos intercepteurs, d'autre part.

Par exemple, au niveau du client, les invocations dépendent du style de réplication mais surtout d'une analyse de la référence `CORBA` de l'objet ou du groupe d'objets distants. Cette analyse s'effectue uniquement en se basant sur les structures de données et en utilisant les primitives de la couche neutre. Bien entendu, la couche neutre, peut, se baser sur les mécanismes spécifiques de la personnalité protocolaire pour effectuer la tâche demandée. Pour ce faire, des mécanismes comme le polymorphisme et les fonctions de rappel (Callback) peuvent être utilisés. Les `IOR` admettent une représentation "neutre", il est par exemple possible de référencer un profil particulier dans une `IOR` tout en restant dans la couche neutre. Cependant les fonctions de représentation (Marshalling et Unmarshaling) dépendant de `GIOP` sont chargées sous forme de fonctions de rappel et sont utilisées en cas de besoin. Ces fonctions permettent de passer du format neutre au format spécifique à la personnalité et inversement. Nous définissons également des fonctions de rappel permettant de récupérer les composants étiquetés sous format neutre à partir des références et des profils. L'intercepteur, n'utilise que ce type de primitives pour décider, selon le style de réplication dont il est responsable, comment et vers qui envoyer les requêtes.

D'une manière similaire , pour les ré-invocations transparentes des requêtes, l'exception `LOCATION_FORWARD_PERM` est ajoutée lorsqu'il est nécessaire sous forme neutre comme réponse à la requête. Lors de l'emballage et de l'envoi de la réponse, les mécanismes spécifiques à la personnalité protocolaire sont appelés. La ré invocation de la requête au niveau de l'appelant se fait lors de l'analyse des exceptions en retour à sa requête. Cette analyse se fait en se basant sur les mécanismes de la couche neutre (rappelons que les exceptions admettent un format neutre

indépendant des personnalités applicatives en général et de CORBA en particulier). Pour conclure, toutes les tâches réalisées au niveau des intercepteurs n'ont aucun lien direct vers GIOP. Nos intercepteurs sont totalement indépendants des personnalités protocolaires. Cette constatation sur l'architecture a également été vérifiée lors de la phase d'expérimentation et de la réalisation de prototypes.

Notons enfin que nos intercepteurs dépendent de la personnalité CORBA. Ces derniers se basent en effet sur des routines spécifiques à cette personnalité pour retrouver le ReplicationManager (Resolve_Initial_References) et pour interagir avec lui (par exemple pour déterminer la dernière version d'une IOGR ou pour accéder aux propriétés des groupes d'objets). L'interaction avec le ReplicationManager est la seule dépendance de ces intercepteurs vers la personnalité CORBA. Cette dépendance peut être levée en cas de besoin, mais ce travail n'a aucune justification à présent. Une tolérance aux fautes totalement indépendante des modèles des distributions dépasse les objectifs de cette thèse.

3.3.3.5 Conclusion

Nous avons présenté une architecture basée sur des intercepteurs portables s'inspirant de ceux définis par la norme CORBA. Nous avons défini quatre points d'interception et affecté les différents traitements requis par les styles de réplication à ces quatre point. Nous nous sommes également intéressés à la position de ces intercepteurs dans l'architecture. Ces intercepteurs font partie de la personnalité applicative CORBA et n'ont aucune visibilité sur les personnalités protocolaires. Les détails de mise en oeuvre des différents intercepteurs seront fournis ultérieurement dans ce manuscrit (paragraphe 5.3.5).

Les intercepteurs sont vus comme des boîtes noires pouvant être appliquées d'une manière totalement transparente aux applications et à l'intergiciel. Ce choix permet de passer facilement d'une application classique à une application tolérante aux fautes. Ce choix présente d'autres avantages au niveau de la préservation des propriétés de l'architecture schizophrène, en particulier la configurabilité et le support de la vérification formelle. Nous revenons avec plus de détails sur ces avantages après la présentation de notre proposition pour la détection et la notification des défaillances.

3.3.4 Détection et notification des défaillances

Pour la mise en place de la détection et de la notification des défaillances nous nous sommes fixés les objectifs suivants : performance, passage à l'échelle, support des restrictions définies par le profil Ravenscar et flexibilité.

La détection des défaillances est requise par le ReplicationManager pour garantir une vue cohérente des groupes d'objets. Un détecteur de défaillances doit pouvoir surveiller plusieurs répliques simultanément. La norme définit deux modes de surveillance :
- le mode pull, dans ce cas l'application doit implémenter l'interface PullMonitorable. Le détecteur de défaillances se base sur cette interface pour mettre en place des appels périodiques permettant de déterminer si les objets surveillés sont opérationnels.
- le mode push nécessite que l'application envoie régulièrement des messages qui “prouvent” son bon fonctionnement.

Le mode push requiert un effort important de configuration et de déploiement et probablement des interactions supplémentaires entre l'infrastructure de tolérance aux fautes et l'application si le détecteur de défaillances change pendant l'exécution de l'application. Le mode pull ne pose pas ce problème, les répliques ne doivent plus être au courant de l'existence et des différents

paramètres des détecteurs des défaillances. Elles se comportent naturellement comme de simples serveurs répondant à des requêtes *is_ alive* pouvant être émises par n'importe quelle entité.

Les performances du détecteur des défaillances sont primordiales pour toute application basée sur `FT CORBA`. En effet, pendant le temps qui sépare la défaillance d'une réplique de la prise en compte de cette information par le `ReplicationManager`, les groupes d'objets sont incohérents ce qui peut, selon le style de réplication causer une perte du temps importante dans les mécanismes de ré-invocations, de mises à jour des `IOGR` au niveau des clients et de synchronisation/restitution des états au niveau des répliques. Le passage à l'échelle est important pour faciliter le déploiement des applications `FT CORBA` qui est déjà complexe.

Un détecteur de défaillances doit pouvoir surveiller simultanément plusieurs répliques. La nature synchrone du modèle de distribution de `CORBA` impose l'allocation de plusieurs tâches qui doivent s'exécuter en concurrence pour effectuer les appels synchrones et lancer les temporisations. La compatibilité avec le profil `Ravenscar` est l'une des exigences des applications de haute intégrité. Ce dernier facilite, d'une part, les analyses d'ordonnancement statiques et d'autre part, limite les exigences vis-à-vis du système opératoire et de l'architecture matérielle. Outre ses impacts sur le comportement temporel et sur la consommation des ressources, le choix de se restreindre à ce profil pousse à définir les besoins exacts en termes de service de concurrence défini par l'intergiciel. Les deux prochaines sections présentent les choix architecturaux que nous avons faits pour mettre en place la détection et la notification des défaillances.

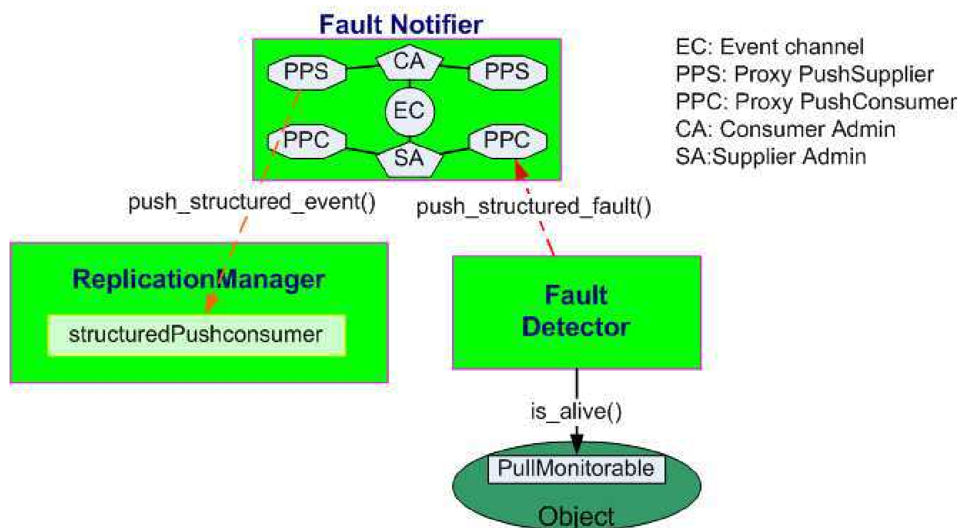

FIG. 3.5 – Détection et notification des défaillances

3.3.4.1 Détection des défaillances

Les détecteurs de défaillances que nous avons conçus permettent un contrôle simultané de plusieurs répliques. Chaque réplique à surveiller est enregistrée auprès du détecteur de défaillances. Les paramètres d'enregistrement les plus importants pour une réplique sont sa référence ainsi qu'un `timeout` spécifiant la latence maximale au delà de laquelle la réplique est suspectée. La nature synchrone des invocations `CORBA` et la difficulté de mise en place de l'avortement de requêtes nous impose d'allouer deux tâches par objet à surveiller. La première tâche est en quelque sorte “esclave” ; elle est responsable de faire les invocations synchrones. Les invocations au niveau des objets à surveiller sont construites en se basant exclusivement sur des constructions de la

couche neutre. Cette tâche ne présente aucune dépendance sur les personnalités protocolaires et applicatives. La seconde tâche est "maîtresse" ; elle est responsable de déterminer le début et la fin des cycles de détection, de lancer les `timeouts` et de décider si l'objet `CORBA` surveillé est tombé en panne ou pas.

La gestion des `timeouts` nécessite une grande attention surtout sous les restrictions `Ravenscar`. Nous nous sommes inspirés du patron présenté dans [17]. Ce patron permet de signaler une variable conditionnelle après une certaine durée (représentant le `timeout`). Nous avons généralisé ce patron pour gérer plusieurs `timeouts` à la fois et pour pouvoir annuler une requête de `timeout`. L'utilisation d'objets protégés et d'abstractions de variables conditionnelles a permis une implantation compatible avec le profil `Ravenscar`. Par ailleurs, l'utilisation des services génériques disponibles au niveau de la couche neutre définie par l'architecture schizophrène facilite la configuration de la détection des défaillances.

3.3.4.2 Notification des défaillances

Le service de notification de `CORBA` permet d'implanter un modèle Publication/Souscription (pub/sub). Ce modèle permet une propagation asynchrone des données tout en se basant sur `CORBA` pour le transport effectif des données.

Le découplage assuré par le service de notification entre les détecteurs de défaillances et le `ReplicationManager` permet une propagation efficace des rapports d'erreurs et facilite surtout le déploiement de l'infrastructure de tolérance aux fautes. Le schéma proposé par la norme définit des `PushSupplier` (producteurs actifs) et des `PushConsumer` (consommateurs réactifs). A la détection d'une défaillance, un rapport d'erreur est produit et envoyé au canal de notification maintenu par le `FaultNotifier`. Ce dernier propage ce rapport au `ReplicationManager`. Le schéma 3.5 montre les principales interfaces et méthodes utilisées pour la propagation de ces rapports.

Notons que pour être vraiment utile, le service de notification doit être fiable. La fiabilité de ce service est un problème qui sort du cadre de cette thèse. Cependant l'architecture de ce service, son support par l'architecture schizophrène et le schéma de propagation ne semblent pas avoir de rapport direct avec cette question de fiabilité.

Le `ReplicationManager` contient une tâche spécifique en attente des rapports de défaillances. Cette tâche analyse ces différents rapports afin de découvrir les répliques défaillantes. Les groupes d'objets sont alors mis à jour. La nouvelle composition du groupe d'objets est également propagée aux clients sous forme d'exceptions en cas de besoin (voir section 5.3.5).

3.3.4.3 Conclusion

Nous ne trouvons que très peu de travaux concernant la détection et la notification de défaillances dans les intergiciels mettant en oeuvre `FT CORBA`. Ces deux éléments ont pourtant un impact immédiat sur le comportement et sur certains certains attributs de la sûreté de fonctionnement de l'application finale comme la fiabilité et la disponibilité.

Les détecteurs de défaillances que nous avons mis en place ne sont pas dépendants de `CORBA` ; il ne font qu'une seule hypothèse quant au modèle de distribution : pouvoir répondre d'une manière synchrone aux requêtes *is_ alive*. Notre architecture ne dépend de la personnalité `CORBA` que par l'utilisation du service de notification. Nous proposons dans le chapitre suivant un modèle de notification de haut niveau n'utilisant que des services neutres.

L'architecture que nous avons proposée pour la détection et la notification des défaillances complète notre proposition pour intégrer `FT CORBA` dans l'architecture schizophrène. Nous évalu-

ons cette architecture dans la prochaine section.

3.3.5 Avantages de l'architecture

L'objectif de cette section est de discuter les avantages de l'architecture que nous proposons. Cette discussion sera complétée par les différents tests de validation et mesures de performances dans la troisième partie de ce manuscrit. Nous discutons trois points : le support des personnalités protocolaires, la facilité de la vérification formelle ainsi que les aspects de configuration.

3.3.5.1 Indépendance des personnalités protocolaires

Lors de la description de nos intercepteurs nous avons souligné leur indépendance des personnalités protocolaires. Nous nous sommes efforcés de rester neutre par rapport aux personnalités protocolaires. A chaque fois qu'une fonctionnalité fournie par une personnalité protocolaire est nécessitée, nous l'utilisons en passant par la couche neutre. En cas de besoin, une représentation "neutre" de la fonctionnalité utilisée est conçue et utilisée. Ainsi, les intercepteurs n'utilisent directement aucune construction spécifique au protocole de communication. Même si la norme prévoit l'utilisation de certaines constructions propres à `GIOP` comme la "location forwarding", il est possible d'utiliser ces mêmes intercepteurs avec tout protocole supportant ces mécanismes ou des mécanismes équivalents. En coutre, grâce au même principe, nos détecteurs de défaillances sont développés en utilisant des constructions exclusivement neutres et sont totalement indépendants de toutes les personnalités. Les invocations à distance n'utilisent en effet que la représentation neutre de la référence de l'objet à surveiller. Ceci implique que nos détecteurs sont capables de surveiller n'importe quel objet, bien entendu s'il est possible d'appeler cet objet grâce à une invocation synchrone.

Une perspective à ce travail consiste à supporter l'utilisation d'autres personnalités protocolaires fournissant des fonctionnalités comme le multicast fiable (reliable multicast). Ce support nécessite a priori d'établir la correspondance entre les identifiants des objets et leurs références. Le `ReplicationManager` peut être le meilleur composant permettant la mise en oeuvre de cette correspondance. Pour les spécifications `UMIOPUnreliable Multicast Inter-ORB Protocol`), cette correspondance est relativement simple. Elle consiste à associer les `IOGR` de `FT CORBA` à la référence du groupe multicast (`Group IOR`). Les structures de données des deux références sont en outre très similaires. Notons enfin que certains travaux de recherche récentes visent à fournir des implémentations fiables de `UMIOP` [11].

3.3.5.2 Support de la vérification formelle

L'un des avantages de l'architecture schizophrène est le support de la vérification formelle. La méthode de vérification appliquée se base sur l'isolation des aspects comportementaux dans un unique composant (μ`Broker`). Ce composant est ensuite modélisé en se basant sur les réseaux de Petri colorés. Les propriétés comportementales peuvent être vérifiées grâce à ces modèles.

Les intercepteurs que nous avons proposés isolent tout les aspects comportementaux liés aux traitements des requêtes, les techniques utilisées pour la vérification du μ`Broker` peuvent être également appliquées pour valider le comportement des implantations des différents styles de réplication. Notons que les larges capacités de l'interception, et en particulier la possibilité de déclencher et stopper des invocations ainsi que l'action sur les paramètres des requêtes, peuvent causer des problèmes de vivacité ou même de sûreté s'ils sont mal utilisés. Ce problème n'est pas propre à nos intercepteurs, les intercepteurs portable de `CORBA` souffrent de ce genre de problèmes [10].

Les différents tests et mesures que nous avons effectués montrent un bon comportement de ces intercepteurs. Une perspective de notre thèse consiste en l'utilisation de techniques de modélisation formelle comme les réseaux de Petri colorés pour valider nos intercepteurs.

3.3.5.3 Configurabilité de la tolérance aux fautes

Les concepts de l'architecture schizophrène, associés au principe de séparation des préoccupations que nous nous sommes efforcés d'appliquer et à la transparence des intercepteurs par rapport à l'application et à l'ORB facilitent le choix et l'application des différents styles de réplication d'une manière efficace et flexible. L'introduction de la tolérance aux fautes dans l'architecture schizophrène ne limite pas les possibilités de configuration de cette dernière. Elle les augmente avec toutes les propriétés de tolérance aux fautes que définit la norme. En revanche, elle nécessite plus de cohérence dans les choix des configurations. L'utilisation d'un langage de description d'architecture comme AADL [109] pour gérer les nombreux paramètres de configuration ainsi que la compatibilité dans le choix de ces paramètres permet de profiter efficacement de la souplesse de notre architecture. Nous reviendrons sur ce point dans notre conclusion générale.

3.4 Conclusion

Dans ce chapitre, nous avons décrit par décrire l'architecture schizophrène ainsi que ces avantages. Cette architecture présente deux particularités : d'une part, elle favorise la configuration et l'interopérabilité des services intergiciels. D'autre part, elle supporte la vérification formelle. Les principes de cette architecture ont influencé nos choix de conceptions. La description des principes architecturaux et des différents services de cette architecture nous sert de base pour la présentation de nos propositions tout au long de ce mémoire. L'architecture schizophrène fournit une solution à deux problèmes essentiels : la réduction des coûts et le support d'exigences de qualité comme la compatibilité avec le profil de restrictions Ravenscar et le support de la vérification formelle. Ces différentes exigences sont satisfaites simultanément.

Dans la seconde partie de ce chapitre, nous nous sommes intéressés à l'intégration de FT CORBA dans l'architecture schizophrène. Nous avons analysé cette architecture et défini les différentes exigences de cette spécification. Cette analyse architecturale et fonctionnelle nous a permis d'isoler les abstractions requises pour l'introduction de la tolérance aux fautes dans l'architecture schizophrène sans porter préjudice aux différents propriétés de cette architecture. Nous avons ensuite étudié les différentes mécanismes d'interception dans CORBA et discuté leurs apports et surtout leur compatibilité avec l'architecture schizophrène. Nous avons adopté une solution proche des intercepteurs portables de CORBA qui souffrent de certaines limitations que nous avons détaillées et contournées dans notre proposition. Les intercepteurs que nous avons définis nous ont permis de mettre en oeuvre tous les styles de réplication proposés par la norme. Les différents intercepteurs font partie de la personnalité applicative CORBA, nous avons vérifié leur indépendance de toute personnalité protocolaire. Ces intercepteurs facilitent le passage d'une application classique à une application tolérante aux fautes. Ces intercepteurs facilitent également le changement du style de réplication. Nous avons également proposé un schéma de détection de défaillances se basant uniquement sur la couche neutre de l'architecture schizophrène et compatible avec les restrictions Ravenscar. Le service de notification a été utilisé pour assurer la propagation des rapports d'erreurs. Les avantages architecturaux, notamment les aspects de configuration, le respect des règles de visibilité entre l'applicatif et le protocolaire et la totale indépendance de GIOP de nos propositions ont été explicités. Ces différentes propriétés seront complétées par les tests et les mesures de performances dans la troisième partie de ce manuscrit.

Chapitre 4

Architecture d'un service générique de consensus

Contents

4.1 Conception d'un service générique de consensus 82
4.1.1 Architecture générale 82
4.1.2 Service du consensus 86
4.1.3 Interactions entre les composants du service du consensus 92
4.2 Intégration dans l'architecture schizophrène et cas d'applications 98
4.2.1 Groupe de participants 98
4.2.2 Intégration dans l'architecture schizophrène 100
4.2.3 Configuration de l'intergiciel 102
4.2.4 Application à la personnalité CORBA 104
4.3 Conclusion 105

La définition d'un service intergiciel pour le consensus a deux motivations essentielles. D'une part, les nombreux cas d'utilisation font du consensus une abstraction de première nécessité pour plusieurs applications. D'autre part, les technologies intergicielles permettent un développement rapide et peu coûteux des applications distribuées. Le spectre des applications cibles d'un intergiciel peut significativement s'élargir s'il dispose d'un service de consensus. Pour être vraiment utile, ce service doit présenter certaines propriétés lui permettant de répondre aux exigences de réutilisation et de flexibilité tout en présentant un comportement temporel stable.

Ce chapitre se compose de deux parties. Dans la première, nous présentons les différents aspects architecturaux d'un service de consensus générique. Nous mettons l'accent d'une part, sur le rôle des services intergiciels lors du support de cette abstraction et d'autre part sur les interactions entre les différents composants de ce service. L'architecture résultante prend compte des différents besoins présentés par les systèmes critiques : configurabilité, adaptation, minimisation de la consommation des ressources et comportement temporel stable.

La seconde partie de ce chapitre s'intéresse aux différentes possibilités de configuration du service que nous proposons. Nous nous intéressons également à divers cas d'utilisation. Outre les cas d'utilisation basiques, nous traitons l'intégration de ce service au sein de l'architecture schizophrène et les avantages de cette intégration. Nous nous intéressons enfin à l'utilisation de ce service dans le cadre de l'architecture schizophrène. La compatibilité avec la personnalité `CORBA` sera prise comme exemple.

4.1 Conception d'un service générique de consensus

La généralité de l'abstraction du consensus et ses nombreux cas d'utilisation imposent des contraintes sur le service qui la fournit. Ce dernier doit répondre aux problèmes de réutilisation et de généricité tout en minimisant la consommation des ressources et en préservant le déterminisme et les performances. Ces problèmes sont posés par plusieurs applications, en particulier par les applications critiques tolérantes aux fautes.

Les intergiciels fournissent des moyens puissants permettant de répondre à ces différents objectifs. Nous nous intéressons à deux types d'interactions : d'une part celles entre le service du consensus et les autres services de l'intergiciel et d'autre part celles au sein même de ce service. Le premier type d'interaction est défini grâce à l'isolation du rôle de l'intergiciel lors du support de l'abstraction du consensus. Il sert plutôt à offrir des propriétés de généricité, de portabilité et d'interopérabilité à notre service. L'étude des interactions au sein de ce composant a pour objectif de maximiser ses performances et plus généralement à optimiser ses propriétés comportementales.

Dans cette section, nous commençons par présenter une vue d'ensemble de notre architecture et détaillons le rôle des services intergiciels. L'abstraction du consensus est doublement dépendante de ces services. D'une part, ces services fournissent les abstractions nécessaires permettant par exemple l'exécution concurrente et les échanges de données. D'autre part ces services doivent répondre aux exigences dérivées des hypothèses sur la synchronie et sur les défaillances requises par les algorithmes.

Nous définissons ensuite l'architecture de notre service. Il se compose de trois composants pour le consensus, la détection des défaillances et la gestion des messages. Nous nous sommes inspirés des patrons de conception Pont et Stratégie pour structurer ce service et pour maîtriser les dépendances entre les différents composants tout en assurant la séparation des préoccupations.

Nous proposons enfin une solution basée sur les évènements pour définir et contrôler les interactions entre les différents composants du service du consensus. Cette solution permettra d'assurer un faible couplage entre les différents composants et d'optimiser le comportement temporel de notre service.

4.1.1 Architecture générale

Les algorithmes de consensus s'expriment généralement en utilisant des primitives simples (envoi ou attente d'un message, attente d'un évènement particulier, par exemple l'expiration d'un timeout, etc.). Il y a donc plusieurs degrés de liberté à exploiter afin de maximiser la généricité et la qualité du service.

Dans ce paragraphe nous proposons une architecture d'un service générique pour le consensus. Nous commençons par une discussion sur le modèle de calcul à adopter et donc des algorithmes à supporter. Ensuite, nous présentons les traits généraux de l'architecture que nous proposons. Cette architecture comporte deux niveaux. Le premier contient l'ensemble des composants constituant le service du consensus. Ce service peut être vu comme un ensemble de composants de haut niveau représentant trois abstractions : consensus, détection des défaillances et échanges de messages. Ces composants interagissent avec des services intergiciels pour fournir les différentes fonctionnalités dont ils sont responsables. La seconde couche contient les services intergiciels nécessaires au bon fonctionnement des composants du service du consensus. Le rôle de ces services de base sera détaillé dans la troisième partie de cette section.

4.1.1.1 Choix des modèles de calcul et de défaillances

Les systèmes asynchrones, parce qu'ils imposent moins de contraintes sur la plate-forme de communication, sont plus flexibles, plus faciles à déployer et à (re)configurer. Ils souffrent cependant d'un handicap majeur. Ils ne permettent pas de résoudre des problèmes essentiels comme le consensus ou la diffusion atomique dès lors qu'au moins un des participants est sujet à une défaillance même bénigne [45].

Dans un souci de généralité, et pour limiter les contraintes sur les composants de l'intergiciel, nous nous basons sur le modèle asynchrone avec détecteurs de défaillances. Pour l'abstraction du consensus, ce modèle est plus général que le modèle asynchrone puisque tout algorithme résolvant le problème du consensus dans le modèle asynchrone, le résout aussi dans ce modèle. En plus, comme le montre [22], ce modèle permet de contourner le résultat fondamental d'impossibilité de Fisher, Lynch et Patterson [45]. Notons que ce modèle est parfaitement utilisable dans le cadre d'applications temps réel distribuées [76].

Le choix de ce modèle n'empêche pas de supporter les algorithmes de consensus faisant des hypothèses plus fortes sur la synchronie du système. Cependant, l'utilisation de ces algorithmes nécessitera une configuration adéquate des services de l'intergiciel. Des détails complémentaires seront fournis plus loin dans le paragraphe 4.1.1.3.

4.1.1.2 Architecture générale

D'un point de vue purement fonctionnel, nous définissons deux couches intergicielles. Comme le montre la figure 4.1, la première couche contient les composants requis pour fournir les abstractions de consensus et de détection de défaillances. Le second niveau dénommé "services intergiciels de base" contient les composants fournissant les services intergiciels nécessaires pour la bonne exécution des différents composants du service du consensus.

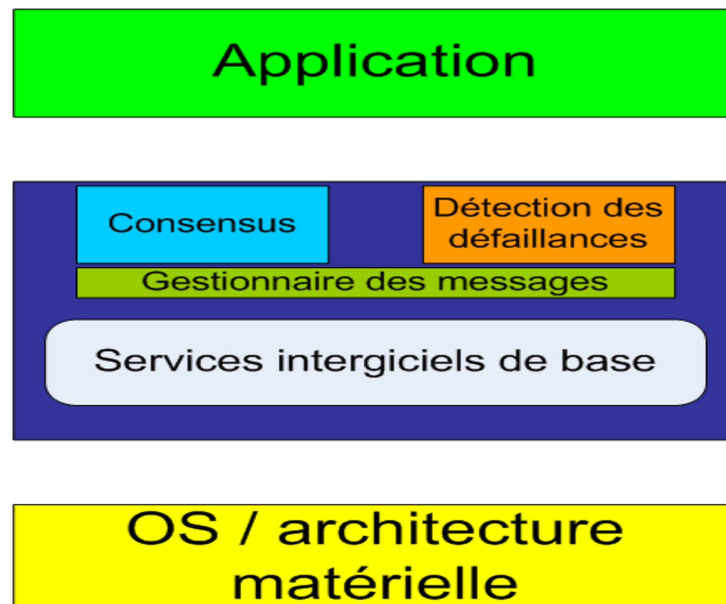

FIG. 4.1 – Architecture générale

Le niveau supérieur contient les nouveaux composants que nous introduisons pour implanter le service de consensus. Il se compose de trois composants qui coopèrent pour fournir l'abstraction

du consensus. Le premier composant fournit une interface permettant aux entités applicatives de participer au consensus. Le second permet la détection de défaillances. Le troisième composant permet aux différents participants exécutant des algorithmes de consensus ou de détection de défaillances d'échanger des données. Notons que l'abstraction des détecteurs de défaillances n'est pas toujours nécessaire. Ainsi, si l'algorithme de consensus suppose un système synchrone, le composant de détection des défaillances devient optionnelle voire inutile.

La seconde couche "services intergiciels de base" est une couche que nous définissons pour désigner les services intergiciels basiques nécessaires aux composants mettant en oeuvre le service du consensus. Les composants constituant cette couche sont primordiaux pour le fonctionnement des composants de la couche supérieure. En particulier, la couche du niveau de base doit être correctement configurée afin de garantir, à tout instant, les hypothèses faites par les composants de consensus et de détection des défaillances. Nous donnons une première description des services intergiciels de base dans le paragraphe suivant.

4.1.1.3 Services intergiciels de base

L'architecture que nous proposons définit une couche intergicielle fournissant les services requis par les nouveaux composants que nous avons proposés pour mettre en oeuvre le service du consensus. Cette couche a deux rôles principaux.

Premièrement, l'intergiciel fournit un support fonctionnel à l'exécution de ces algorithmes en agissant comme un intergiciel de distribution et d'interfaçage avec l'OS. L'intergiciel d'interfaçage avec l'OS isole les entités applicatives et les entités intergicielles de haut niveau de l'OS et du matériel. L'intergiciel de distribution fournit plusieurs services permettant de faire abstraction des différents paramètres de déploiement. Deuxièmement, l'intergiciel doit garantir les hypothèses faites par les algorithmes de consensus et de détection des défaillances durant toute le cycle de vie de l'application.

Interfaçage avec le système opératoire L'intergiciel d'interfaçage avec l'OS (host infrastructure middleware dans [112]) fournit les abstractions nécessaires à la conception et au développement des applications indépendamment des OS et des architectures matérielles.

Plusieurs algorithmes de consensus et de détection des défaillances nécessitent l'exécution concurrente de tâches. C'est le cas de la majorité des algorithmes de détection des défaillances basés sur le mécanismes de timeout, mais aussi d'un grand nombre d'algorithmes de consensus. Ces différents algorithmes s'expriment en utilisant deux ou même trois tâches pour certains. Le support de ces algorithmes nécessite donc un service pour la gestion des exécutions concurrentes. L'intergiciel d'interfaçage avec l'OS fournit ces services indépendamment des architectures matérielles et des OS et assure la portabilité des composants de consensus et de détection de défaillances.

Distribution L'intergiciel de distribution distribution middleware permet l'échange de données entre plusieurs entités séparées physiquement ou logiquement. L'intergiciel de distribution résout les problèmes d'hétérogénéité des configurations matérielles, des langages de programmation et même des modèles de distribution, comme le permet l'architecture schizophrène. Comme nous l'avons montré dans le paragraphe 3.2.3, un service de transport efficace permet, à l'application, en cas de besoin, d'utiliser des protocoles de transport dédiés, comme `SCPS-TP` [35] ou ceux supportant les liens `SpaceWire` [95].

Les services d'interfaçage avec l'OS et de distribution assurent la portabilité et l'interopérabilité des applications mais également des autres services qui se situent au dessus de ces deux

couches. Pour les algorithmes qui ne présentent pas d'hypothèses sur les modèles de synchronisme et de défaillance, le support fonctionnel fourni par les services intergiciels de base suffit pour assurer ces propriétés. Si ces algorithmes présentent des exigences en terme de synchronie ou de modèle de défaillances, cette couche doit les garantir. Le prochain paragraphe traite cette question.

Garantie des modèles de calcul et de défaillances Les algorithmes de consensus font un ensemble d'hypothèses sur les modèles de défaillances et les modèle de calcul de l'environnement dans lequel ils s'exécutent. Ces hypothèses doivent être mis en application par l'ensemble composé par le système opératoire, l'architecture matérielle et par les liens de communications. L'intergiciel, par définition, cache les détails du système opératoire et du matériel. La couche des services de base, par sa position dans l'architecture assure cette tâche. Il est alors de sa responsabilité d'exporter le modèle de calcul et celui des défaillances requis par les composants de la couche supérieure. Par conséquent, le choix et la configuration des services intergiciels de base doit, en plus de répondre efficacement aux différentes exigences de l'application, assurer la satisfaction des hypothèses des différents algorithmes choisis.

Par exemple, si un algorithme de consensus nécessite un modèle de calcul synchrone, le gestionnaire des messages doit garantir une borne supérieure sur les délais de transmissions de messages [33]. Or ce dernier n'est qu'une interface entre les algorithmes et service de transport de bas niveau assuré par l'intergiciel ; son comportement temporel est limité par le protocole effectif utilisé pour les échanges des données qui dans cette architecture doit être fourni sous forme d'un service intergiciel de base correctement configuré. Cet exemple montre l'impact d'une hypothèse de synchronie sur la sélection et la configuration d'un service intergiciel. De la même manière, les services intergiciels de base et la propriété de configurabilité doivent permettre de satisfaire les hypothèses des différents algorithmes en termes de modes de défaillances. En cas de besoin, l'utilisation des technologies telles que l'emballage (wrapping) peut s'avérer utile. Ces technologies permettent, en effet, aux différentes entités d'exporter un modèle de défaillances plus fort que celui qu'ils auraient exhibé autrement [107].

Même si nous nous sommes limités pour notre études aux algorithmes évoluant dans un modèle asynchrone sujet aux défaillances par crash, l'attention que nous portons à la configurabilité et à la relation entre le service du consensus et la couche des services de base facilitera le support ultérieur d'autres algorithmes venant avec des hypothèses plus fortes sur la synchronie et les modes de défaillances.

4.1.1.4 Conclusion

Nous avons présenté et explicité les motivations du choix de la classe des algorithmes de consensus supportés par notre service. Nous avons ensuite présenté les traits architecturaux de ce service. Notre architecture définit deux niveaux : une couche supérieure constituée par les différents composants du service du consensus et une seconde couche comprenant les services intergiciels de base. Cette deuxième couche fournit toutes les abstractions nécessaires à la bonne exécution des services de consensus et de détection des défaillances. Elle a un double rôle : d'une part elle permet la portabilité et la réutilisation du service du consensus. D'autre part, elle supporte la généricité de ce dernier. La généricité du service du consensus est la propriété qui lui permet de supporter un grand nombre d'applications. Elle est obtenue en supportant plusieurs algorithmes et en respectant les exigences non fonctionnelles. Le choix et la configuration de cette couche doit d'une part permettre le support de plusieurs algorithmes en satisfaisant les hypothèses

faites par ces derniers et d'autre part d'optimiser les performances et la consommation mémoire des différents algorithmes selon l'environnement et le contexte d'exécution.

4.1.2 Service du consensus

Nous nous intéressons dans cette section à l'architecture du service du consensus. Le principal objectif de cette architecture est de maximiser la généricité de ce service. La diversité des applications et les changements possibles des environnements d'exécution rend difficile l'existence d'un seul algorithme satisfaisant à toutes ces exigences. Un tel algorithme devrait par exemple réconcilier, à la fois, la variabilité des environnements d'exécution et des hypothèses de synchronisme aux besoins en performances et en déterminisme, ce qui est impossible dans plusieurs cas. Il faut donc que notre service de consensus supporte l'exécution de plusieurs algorithmes de consensus en maximisant la partie générique commune à tous les algorithmes et en minimisant l'impact du changement d'algorithme sur le code de l'application.

Nous commençons tout d'abord par présenter deux patrons de conception permettant de répondre, sur le plan architectural, aux besoins de configuration et de support de plusieurs algorithmes. Nous nous basons, ensuite sur ces patrons pour définir et décrire les différents composants pour le consensus, la détection des défaillances et la gestion des messages constituant le service que nous proposons. Pour avoir le résultat attendu, l'utilisation de ces patrons doit s'associer à la maîtrise des interactions entre les entités et de la configuration des différents services du niveau de base et du niveau supérieur.

4.1.2.1 Patrons de conception "Pont" et "Stratégie"

Pour répondre aux besoins de configurabilité et d'adaptation nous nous baserons sur deux patrons de conception : les patrons Stratégie (strategy) et Pont (bridge) [49]. Ces patrons permettent de découpler l'interface d'un composant ou d'un objet de son implémentation. Il est alors possible pour une seule interface de supporter plusieurs algorithmes. Dans ce cas, chaque algorithme peut être vu comme une implémentation particulière de l'interface du composant. Ces patrons sont utilisés pour la mise en oeuvre des différents composants du service du consensus. Les deux prochains paragraphes décrivent ces deux patrons. Nous passerons ensuite à la description des différents composants constituant notre service.

Patron Stratégie Le patron de conception Stratégie permet à un service de sélectionner les algorithmes qu'il utilise. Cette sélection peut se faire d'une manière dynamique (pendant l'exécution). Le diagramme de classes de la figure 4.2 met en oeuvre trois types de participants : les classes strategy et context ainsi que les classes de la forme Strategy[A,B,C]. La classe context peut être configurée avec un objet de type strategy. Elle maintient une référence sur cet objet et peut contenir une interface permettant à cet objet d'accéder aux données qu'elle maintient. La classe strategy déclare une interface commune à tous les algorithmes supportés. La classe context utilise cette interface pour utiliser l'algorithme défini par les différents stratégies concrètes. Les stratégies concrètes sont mises en oeuvre par les différentes classes Strategy[A,B,C].

Patron Pont Le patron Pont permet de découpler une abstraction de son implémentation permettant aux deux de varier indépendamment. Ce patron est utile non seulement quand un objet varie mais aussi quand ses services sont amenés à évoluer. Grâce à ce patron, un objet peut représenter une abstraction, le service offert par l'objet est vu comme l'implémentation.

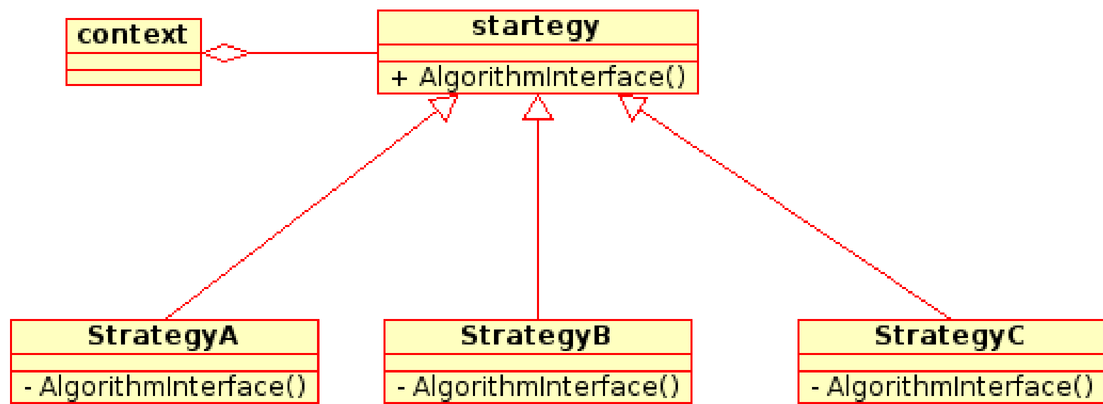

FIG. 4.2 – Patron de conception Stratégie

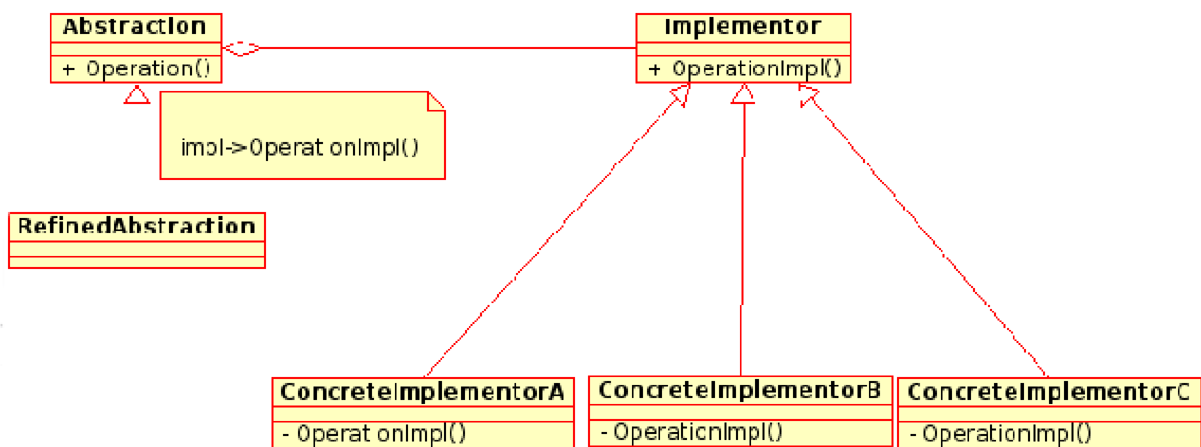

FIG. 4.3 – Patron de conception Pont

Le diagramme de classes de la figure 4.3 présente quatre classes essentielles Abstraction, Refined Abstraction, Implementor et ConcreteImplementor[A,B,C]. La classe Abstraction définit une interface abstraite et maintient une référence sur Implementor. Cette dernière définit l'interface des classes d'implémentation ConcreteImplementor[A,B,C]. Typiquement, la classe Abstraction définit des services de haut niveau basés sur l'interface définie par Implementor.

Conclusion Les deux patrons que nous avons décrits ci dessus se ressemblent. Nous notons néanmoins deux différences. D'une part, le patron stratégie est un patron comportemental alors que le patron Pont est un patron structurel et il est plus lié à la programmation orientée objet (classes abstraites, héritage et polymorphisme). D'autre part le couplage entre la stratégie et ses implémentations est plus faible dans le patron Stratégie.

Pour utiliser efficacement l'un de ces patrons, la définition de l'interface à implémenter doit se faire avec la plus grande attention. En effet, comme elle doit être supportée par toutes les classes concrètes, elle doit être à la fois minimale et représentative. Notons que si l'utilisation du polymorphisme n'est pas possible ou n'est pas souhaitable (par exemple dans certaines implémentation devant passer des standards de certification), il est possible de mettre en oeuvre ces patrons en se basant par exemple sur la notion de pointeur sur fonction. Nous nous basons sur ces deux patrons afin de décrire les différents composants du service du consensus. Nous nous basons également sur ces deux patrons lors de la mise en oeuvre effective de ces composants.

4.1.2.2 Consensus

Ce composant fournit l'abstraction du consensus aux différentes entités applicatives ou intergicielles qui le souhaitent. Pour participer à un consensus, chaque participant propose une valeur. A la fin du consensus, tous les processus disposent de la même valeur décidée. Nous proposons une procédure permettant aux différents processus de proposer une valeur. La valeur décidée est récupérée à la fin de l'exécution de cette procédure.

En s'inspirant des patrons et des principes décrits dans le paragraphe 4.1.2.1, le composant que nous proposons est indépendant des différents algorithmes qui sont réellement exécutés afin d'obtenir un consensus. La figure 4.4 présente le diagramme de classes de notre composant. La classe consensus est équivalente à la classe context proposée par le patron strategy. Les classes Perpetual_Accuracy et Rotating_Coordinator sont les stratégies concrètes. Elles implémentent chacune un algorithme de consensus. Notons la présence de deux méthodes, l'une pour enregistrer les différentes stratégies, l'autre pour créer un objet de type consensus.

Les implantations des algorithmes se basent sur les abstractions fournies par les services intergiciels de base décrits dans le paragraphe 4.1.1.3. Les composants implémentant les différents algorithmes de consensus peuvent également nécessiter l'abstraction de détection de défaillances. Dans ce cas, la compatibilité entre les algorithmes de consensus et les détecteurs de défaillances doit être assurée. Le détecteur de défaillances doit en effet avoir des propriétés plus fortes que celles nécessitées pour le fonctionnement correct de l'algorithme du consensus. Des détails supplémentaires seront fournis dans 4.1.2.3.

Le composant du consensus ne doit interagir qu'avec des composants fournissant des services compatibles avec les besoins et les hypothèses faits par l'algorithme qu'il implante. Ceci peut être assuré grâce à un processus de configuration global comme celui que nous décrirons plus loin dans ce chapitre. Pour le composant du consensus, en plus des services intergiciels de base, les principales interactions sont avec les composants pour :

– *La détection de défaillances*, permettant au composant du consensus d'accéder à l'information concernant les processus suspectés selon deux modes. Le premier consiste à donner la

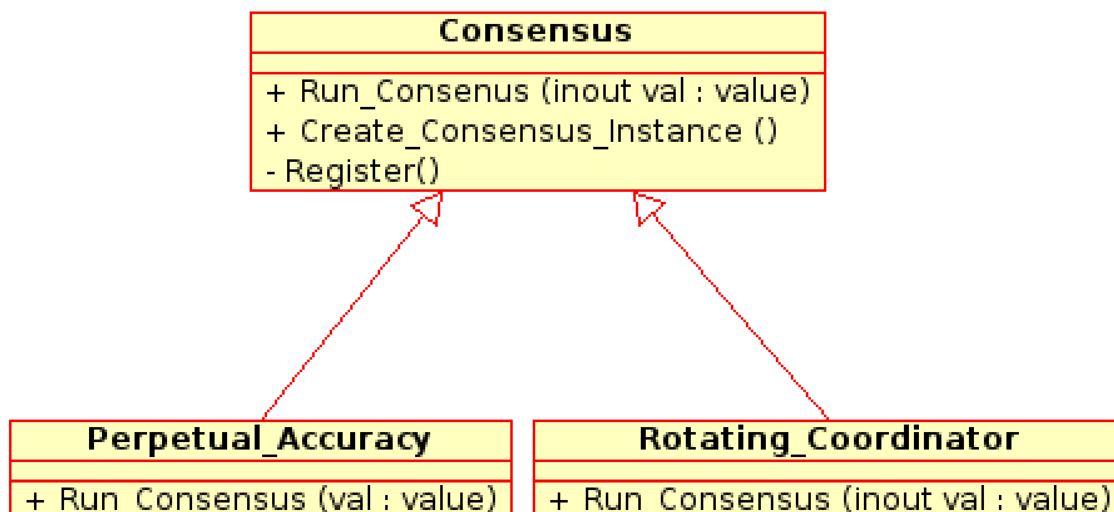

FIG. 4.4 – Interface du composant du consensus

possibilité de lecture d'une liste de suspects. Le second, plus évolué consiste à notifier le composant de consensus lorsqu'un ou plusieurs processus sont suspectés. Nous détaillerons les interactions entre ces deux composants dans le paragraphe 4.1.3.

- *L'échange de messages*, permettant au composant du consensus d'échanger des messages entre les différents participants. Notons que chaque algorithme de consensus se base sur un ensemble de messages ayant plusieurs types (permettant par exemple d'échanger des acquittements, les valeurs proposées, les valeurs décidées, etc.). Ce composant peut coopérer avec d'autres services intergiciels pour supporter plusieurs opérations telles que l'encodage, l'envoi, la réception et le décodage de chacun de ces types.

Dans le paragraphe suivant, nous nous intéressons au composant fournissant l'abstraction du détecteur de défaillances. Cette abstraction est primordiale pour tout une classe d'algorithmes de consensus. Elle inclut naturellement des algorithmes comme ceux proposés par Chandra et Toueg dans [22].

4.1.2.3 Détection des défaillances

Un détecteur de défaillances a pour objectif de maintenir une liste contenant les processus dont il suspecte la défaillance.

L'architecture du composant de détection de défaillance suit le même patron que celui du composant du consensus. La figure 4.5 montre un diagramme de classe décrivant l'abstraction d'un détecteur de défaillance ainsi que deux concrétisations correspondant à deux algorithmes. La classe Failure_Detector fournit aux applications des méthodes pour lancer et stopper un détecteur de défaillances. Elle permet également aux différentes entités implantant les algorithmes de s'enregistrer (Register) et de notifier la défaillance d'un ou de plusieurs participants (Notify_Suspects_Lists).

La qualité d'un détecteur peut être définie grâce aux propriétés comme la complétude et la précision (voir section 1.1). Ces propriétés permettent aux algorithmes de consensus de définir

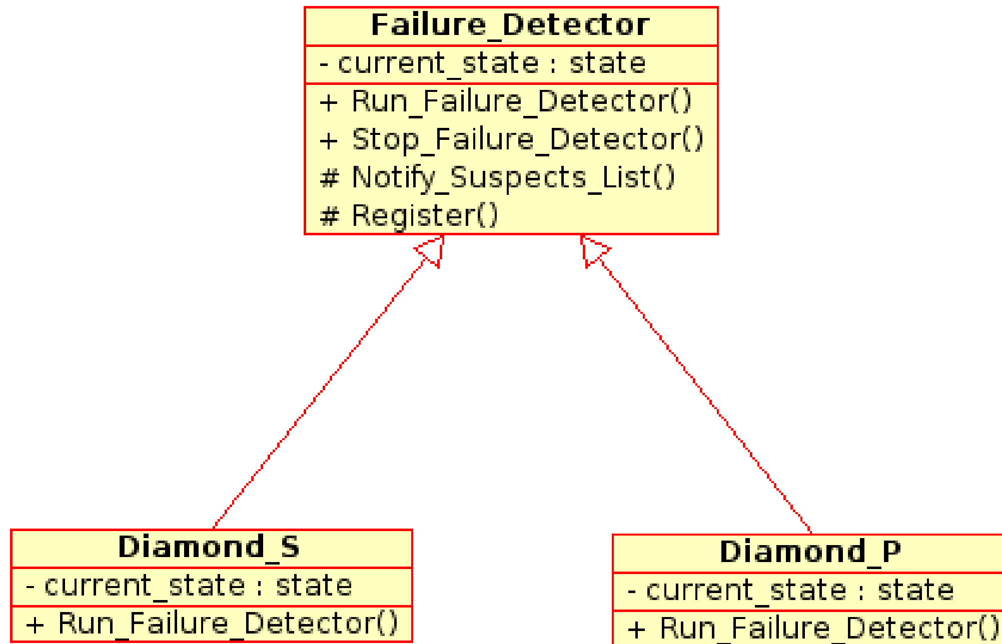

FIG. 4.5 – Interfaces du détecteur des défaillances

les propriétés du détecteur des défaillances dont ils ont besoin. Un détecteur de défaillances est dit compatible s'il présente un ensemble de propriétés plus fortes que celles requises par le composant du consensus. Cette contrainte peut être assurée à deux niveau. Au niveau du processus d'assemblage et de configuration permettant la sélection de l'algorithme de consensus et celui pour la détection de défaillances. Cette contrainte peut également être vérifiée grâce à une assertion lors de l'association du détecteur de défaillances au composant du consensus.

Les algorithmes de détection de défaillances présentent des besoins plus forts que ceux pour le consensus en termes de services intergiciels de base. La mise en place de classes de détecteurs comme les détecteurs parfaits peut par exemple nécessiter la vérification d'hypothèses sur les transmissions des messages. Parmi les services intergiciels de base les plus utiles, nous trouvons le support des échanges de messages bas niveau et celui des exécutions concurrentes. Le composant de détection des défaillances peut également nécessiter l'abstraction d'une horloge lui permettant de lancer les temporisations. Pour des raisons de portabilité, il est préférable que cette abstraction soit fournie au niveau de l'intergiciel parmi les services de base. Dans le prochain paragraphe, nous nous intéressons au troisième composant de notre service. Il est responsable des échanges des messages.

4.1.2.4 Échanges des messages

Chaque algorithme de consensus ou de détection de défaillances définit un ensemble de messages de plusieurs types dépendant des données transportées. Ces messages doivent pouvoir être échangés facilement. Nous définissons un composant fournissant ces fonctionnalités : le gestionnaire des messages. Il fournit en outre un ensemble de primitives permettant a une entité donnée

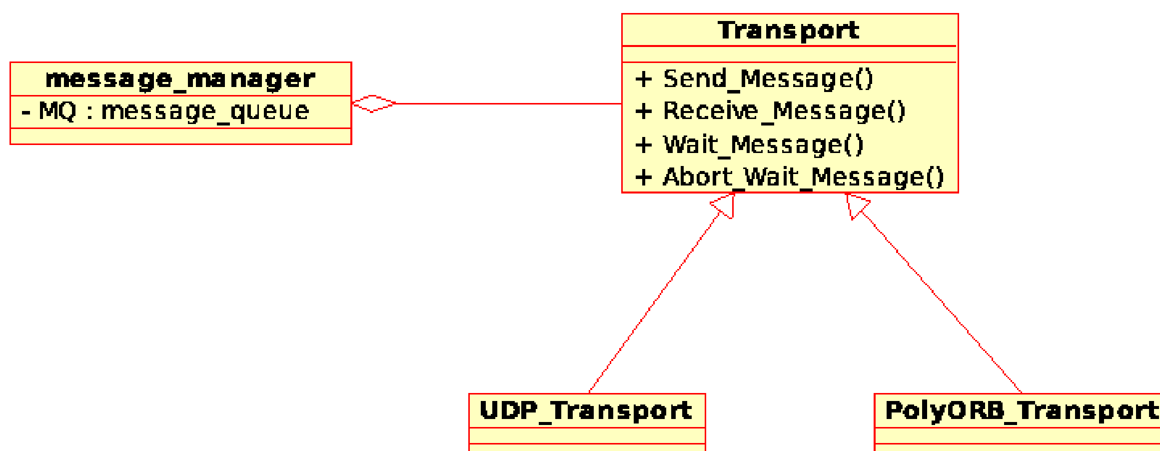

FIG. 4.6 – Interfaces pour la gestion des messages

d'attendre l'arrivée d'un message répondant à certaines caractéristiques comme par exemple le type, le numéro de séquence ou l'expéditeur.

Si les composants de consensus et de détection des défaillances supportent plusieurs algorithmes, ce composant présente le même type de généricité en supportant plusieurs protocoles de transport. Un diagramme de classes pour la gestion des messages est proposé dans la figure 4.6. La classe Message_Manager maintient une file de messages et fournit une interface pour répondre aux besoins présentés par les entités applicatives et celles implantant les algorithmes de consensus et de détection de défaillances. Le gestionnaire des messages se base sur l'interface Transport pour maintenir les queues de messages et permettre par exemple l'attente d'un message satisfaisant un ensemble de critères (expéditeur, destinataire, types des données transportées, etc.). Ce gestionnaire fournit également des primitives d'échange de messages de haut niveau comme le multicast fiable ou la simulation de pertes de messages, etc.

Les opérations (Send_Message, Receive_Message, Wait_Message et Abort_Wait_Message) forment l'interface requise par le gestionnaire des messages et doit donc être implémentée par toute classe concrétisant Transport. Ce choix architectural sépare la gestion des messages des paramètres de déploiement et des protocoles de transport effectivement utilisés pour les échanges de donnés. Ces paramètres sont en effet "cachés" par les implémentations des classes concrètes comme UDP_Transport et PolyORB_Transport.

Le gestionnaire des messages ainsi que les entités fournissant l'interface de Transport coopèrent avec les différents services intergiciels afin d'assurer l'encodage et le transport des différents messages requis par les mises en oeuvre des différents algorithmes de transport et de détection des défaillances. Les primitives évoluées pour l'attente des messages seront réalisées grâce à un modèle d'interactions basé sur les évènements. Ce modèle sera présenté dans la section suivante.

4.1.2.5 Conclusion

Dans cette section, nous avons proposé une architecture générale pour un service de consensus générique. Les lignes directrices de nos choix ont été l'utilisation de patrons de conception comme les patrons "pont" et "stratégie" mais aussi l'application du principe de séparation des préoccupations. L'application de ce principe se manifeste surtout par la définition des services intergiciels de base dont ce service dépend, mais également en définissant des modules au sien

de ce service et en contrôlant les interactions entre ces modules. La définition et le contrôle des interactions entre les composants du service du consensus est le sujet du prochain paragraphe.

4.1.3 Interactions entre les composants du service du consensus

Notre architecture définit trois modules principaux pour le consensus, la détection des défaillances et les échanges des message. Ces modules sont dépendants les uns des autres. Ils doivent échanger une quantité importante de données avant de fournir le résultat d'un consensus. La maîtrise des interactions entre les différents composants permet de satisfaire deux objectifs. D'une part, il ne faut pas impacter les performances et le déterminisme, un mauvais schéma d'interactions peut avoir un effet néfaste sur ce point. D'autre part, il faut contrôler les interactions entre les différents composants afin de limiter le couplage entre eux. Ceci facilite la configuration séparée de chacun des composants de l'architecture.

Afin de garantir les interactions entre les différents composants tout en minimisant le couplage entre ces derniers, nous isolerons les différents évènements produits et consommés par les composants essentiels de notre architecture. Les principes des architectures orientées évènements seront appliqués afin de garantir un schéma d'interactions unifié, favorisant un bon comportement temporel des implantations.

4.1.3.1 Problématique

Les algorithmes de consensus s'expriment généralement en utilisant des primitives simples comme l'envoi et la réception de messages. Certains algorithmes peuvent cependant nécessiter d'autres fonctionnalités plus avancées. Par exemple, il peut être requis d'attendre l'arrivée d'un message d'un type donné ou ayant pour émetteur un processus donné. Certains algorithmes nécessitent l'attente simultanée de l'arrivée d'un message de la part d'un processus particulier ou la suspicion de ce dernier par le détecteur des défaillances. Dans ce cas, il est très important de gérer cette attente simultanée (elle ne peut être décomposée en étapes) efficacement.

Dans cette section, nous nous intéressons essentiellement à deux classes d'évènements produits ou consommés par la majorité des algorithmes. Il s'agit de la notification des défaillances par les composants de détection de défaillances ainsi que la gestion et la livraison des messages aux différents composants. Nous mettons l'accent sur l'importance de ces deux types évènements ainsi que sur la nécessité de gérer les évènements complexes qui en résultent.

Notification des défaillances Une fois la défaillance d'un processus détectée, elle doit être notifiée. Cette information n'est pas utile à tous les composants. Plusieurs algorithmes de consensus nécessitent l'attente de la défaillance d'un processus particulier. Dans l'algorithme du coordinateur tournant (rotating coordinator), chaque processus doit pouvoir détecter la défaillance du coordinateur. Cette détection est très importante pour la propriété de vivacité, car à défaut, le processus demeure dans l'attente d'un message envoyé par ce coordinateur. En cas de panne, le temps de convergence de cet algorithme dépend fortement du temps mis pour détecter la défaillance du coordinateur. Si le temps de la détection d'une défaillance dépend de l'algorithme utilisé, le temps de la notification de cette défaillance dépend des mécanismes fournis pour propager cette information aux composants intéressés. Ce temps de notification doit alors être optimisé.

Échanges des messages La majorité des algorithmes de consensus et de détection de défaillances se base sur les échanges de messages. Certains exigent l'attente d'un message d'un type

donné ou provenant d'un processus particulier. Par exemple, certains détecteurs de défaillances attendent un message de suspicion avant de répondre par un autre message prouvant que le processus est encore actif. Notons que les différents composants sont en concurrence sur les messages, et que certains composants peuvent se mettre en attente d'un message particulier après que celui là soit effectivement arrivé. Ceci nécessite donc un certain mécanisme pour stocker les messages qui ne peuvent pas être délivrés immédiatement. Il est possible d'utiliser une liste de messages (message queue) afin de stocker ce genre de messages. C'est d'ailleurs la solution que nous avons retenue.

Évènements complexes En plus des évènements simples, certains algorithmes nécessitent la gestion de combinaisons d'évènements simples. Par exemple l'attente jusqu'à la satisfaction d'une condition complexe (par exemple : arrivée d'un message d'un processus donné *ou* suspicion de ce processus par le détecteur de défaillances local). La gestion des évènements complexes de ce genre doit se faire avec une grande attention car leur complexité, associée au fait qu'ils fassent intervenir plusieurs producteurs et à la concurrence qui existe sur certains évènements au niveau des consommateurs peut être source de pertes de performances et de dysfonctionnements. En effet, la spécification de primitives bloquantes dans les différentes interfaces ne résout pas le problème de l'attente bloquante d'un évènement parmi plusieurs. L'utilisation de primitives non bloquantes peut constituer une alternative pouvant exhiber un mauvais comportement temporel et sur-consommer les ressources de calcul. C'est le cas avec l'attente active surtout lorsque le nombre d'entités devant réaliser les différentes scrutations devient important. La solution que nous retenons échappe à ces deux problèmes. Elle sera explicitée dans le paragraphe 4.1.3.3.

Récapitulation Les algorithmes de consensus se basent généralement sur des primitives simples comme l'envoi ou l'attente de l'arrivée d'un message ou encore l'attente de la suspicion d'un processus particulier. Les deux classes d'évènements que nous avons isolées ne présentent pas les mêmes contraintes. Si la défaillance d'un processus est une information qui peut intéresser plusieurs composants en même temps, les différents composants sont en concurrence sur les différents messages. Le modèle publication/souscription (pub/sub) peut être appliqué pour la notification des défaillances. En revanche, l'interdiction de la duplication des messages par les canaux de transmissions, généralement supposée par la majorité des algorithmes rend l'application de ce modèle impossible pour l'échange des évènements transportant des messages. En effet, les différentes entités sont en concurrence sur les messages et chaque message ne peut être délivré qu'une fois durant sa durée de vie.

Les différents algorithmes peuvent nécessiter des fonctions plus avancées. Par exemple, il peut être question de bloquer jusqu'à la réalisation d'une condition particulière, par exemple, l'arrivée d'un message ou la suspicion d'un processus. Il faut donc un modèle d'interactions supportant les échanges d'évènements complexes. Dans la section prochaine nous décrivons une solution basée sur la notion d'évènement que nous adapterons à nos besoins. Les différentes primitives nécessaires à l'exécution des algorithmes pourront alors s'écrire en fonction de ces évènements.

Les prochains paragraphes donnent un aperçu sur les architectures orientées évènements, et décrivent en particulier leur capacité à assurer deux objectifs : d'une part, la définition de primitives simples pouvant gérer les évènements d'une manière uniforme et surtout les évènements complexes formés à partir de plusieurs évènements de base et d'autre part, l'efficacité temporelle des interactions dans cette architecture.

4.1.3.2 Architectures orientées évènements

Définition 21 (Évènement) *Un évènement est un changement significatif de l'état d'un système. Ce changement doit être suffisamment important pour nécessiter une réponse.*

Définition 22 (Architecture orientée évenements) *Une architecture orientée évènements est un patron de conception architectural promouvant la production, la détection, la consommation et la réaction aux évènements* [13].

Le concept des architectures orientées évènements est un nouveau concept introduit par la conception des systèmes d'information [21]. Ce terme est utilisé pour désigner les architectures supportant le traitement des évènements complexes en facilitant l'intégration d'unités "métier" tout en préservant un bon comportement temporel [65, 21]. Notons que certaines technologies intergicielles comme les intergiciels orientés messages ont déjà été utilisées pour traiter des évènements simples [65].

Un système orienté évènements consiste typiquement en un ensemble de producteurs et de consommateurs d'évènements. Ces différentes entités peuvent interagir avec un gestionnaire d'évènements qui, comme un canal de notification dans un modèle pub/sub découple les différents producteurs des consommateurs. Les architectures orientées évènements ont en plus l'avantage de pouvoir corréler différents évènements simples et gérer les évènements complexes qui en résultent. Le gestionnaire des évènements peut notifier un ou plusieurs consommateurs, selon la nature de l'évènement. Les architectures orientées évènements adoptent un modèle d'interactions asynchrone selon le mode push [23]. Dans ce modèle, les consommateurs enregistrent l'ensemble des évènements qui les intéressent, par exemple en donnant un ensemble de conditions à satisfaire par les évènements. Le gestionnaire des évènements achemine les évènements crées par les producteurs selon ces conditions. Les consommateurs réagissent enfin aux évènements qu'ils reçoivent. La figure 4.7 présente un exemple d'architecture orientée évènements.

Les architectures orientées évènements se caractérisent par leur support des opérations asynchrones. Les événements peuvent alors être gérés et échangés exactement comme des messages. En plus, dans ces architectures, le flot des évènements est poussé vers les consommateurs qui appliquent les traitements adéquats en réaction à ces évènements. Les architectures orientées évènements présentent deux avantages importants qui motivent l'adoption de ce modèle pour gérer les interactions entre les différents modules de notre service du consensus : le couplage faible entre les composants et les performances.

Couplage faible Les architectures orientées évènements présentent deux caractéristiques favorisant un couplage faible entre les différentes entités. D'une part, elles découplent les producteurs des consommateurs. D'autre part elles facilitent le changement des consommateurs des évènements. Le premier avantage résulte de la définition d'un gestionnaire d'évènements. Ce gestionnaire démultipliée les différents évènements et les achemine aux différentes entités du système. Les consommateurs sont alors découplés des producteurs et l'ajoût ou le retrait d'un producteur ou d'un consommateur peut se faire très facilement.

Le second avantage découle de la constatation suivante : la définition d'un évènement est indépendante de l'opération qu'exécute le consommateur à la réception d'un évènement. Ce qui facilite la définition de nouveaux consommateurs.

Grâce à ce type d'architectures, il est alors possible de changer ou de définir de nouveaux types d'évènements. Il est également possible d'ajouter ou d'enlever des producteurs et des consom-

[13] `http://en.wikipedia.org/wiki/Event_Driven_Architecture`

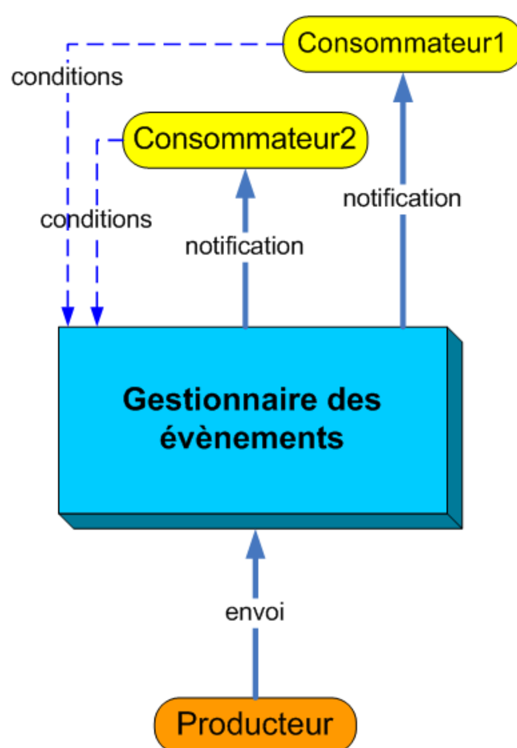

Fig. 4.7 – Architecture dirigée par les évènements

mateurs sans impact sur la plus grande partie de l'application. Pour les schémas de type requête/réponse, le fournisseur du service et les clients doivent s'accorder et adhérer à une interface donnée. Ce qui peut rendre la réutilisation et l'adaptation difficiles. Les architectures orientées évènements évitent justement ce problème.

Comportement temporel Outre le faible couplage et la réutilisation, les architectures orientées évènements garantissent un comportement temporel efficace. Comme le montre Chandy dans [23], les architectures orientées évènements présentent généralement un meilleur comportement temporel que les architectures en `pull` ou celles se basant sur un ordonnancement prédéterminé.

Un modèle de notification se basant sur un ordonnancement prédéterminé peut souffrir d'inefficacité et dans certains cas, s'avérer impossible à réaliser. Par exemple dans une application ou des imprévus peuvent avoir lieu (défaillance d'un calculateur, rupture d'un lien de communications, etc.). Dans un modèle de type `pull` il faut attendre que les évènements soient "tirés" par les consommateurs. Ces derniers doivent, périodiquement s'assurer de la disponibilité de données, généralement en se basant sur des mécanismes peu efficaces comme la scrutation (polling) connu par son efficacité.

Dans le modèle en push, les différents consommateurs peuvent traiter les évènements dès leur arrivée. Le modèle de notification en push, sur lequel se basent les architectures orientées évènements, assure que les producteurs contrôlent le flot des évènements. Ces évènements sont en effet envoyés d'une manière asynchrone aux différents consommateurs. Contrairement aux modèles d'interactions de type `pull` ou ceux se reposant sur un ordonnancement prédéterminé, les architectures orientées évènement permettent donc un acheminement efficace des données depuis les producteurs jusqu'aux aux consommateurs.

Les architectures orientées évènements se basent donc sur un modèle d'interactions garantis-

sant un comportement temporel efficace. Ils permettent également un couplage faible entre les différentes entités et favorise ainsi le principe de la séparation des préoccupations et la réutilisation de composants. Le prochain paragraphe décrit l'application de ce modèle d'interactions à notre architecture.

4.1.3.3 Application de l'architecture orientée évènements

L'application de cette architecture à notre problème nécessite l'identification de trois éléments : les évènements supportés, le rôle des composants par rapport aux différents évènements et le gestionnaire des messages.

Les types des évènements que nous souhaitons traiter, leurs producteurs et consommateurs ont été identifiés dans 4.1.3.1. Nous avons isolé deux types d'évènements essentiels auxquels s'ajoute un troisième type définissant les évènements complexes pouvant être décrits à partir de ces deux évènements de base. Les évènements représentant les arrivées des messages sont produits et consommés par les différents algorithmes de consensus et de détection de défaillances. Les suspicions de processus sont générées par les composants de détection de défaillances et consommées par les composants de consensus et éventuellement d'autres composants. La différence entre les types d'évènements basiques est que les différents composants sont en concurrence sur les évènements d'arrivées de messages et qu'il faut par conséquent les acheminer correctement et sans les dupliquer ; contrairement aux suspicions des processus qui doivent être acheminées à tous les consommateurs potentiels et dont la duplication ne pose pas de problème. Nous traitons ces trois classes d'évènements plus tard dans ce paragraphe.

Nous avons défini un gestionnaire d'évènements permettant aux différentes entités de créer et/ou de réagir à l'occurrence de différents évènements. Les producteurs informent le gestionnaire dès la production de chaque évènement. Les composants de gestion des messages et ceux de détection des défaillances produisent des évènements dont les types dépendent des différents algorithmes.

Pour des raisons de simplicité et d'efficacité, le gestionnaire des évènements, essaye dès l'arrivée de l'évènement de trouver un consommateur potentiel. Si un tel consommateur existe, l'évènement est délivré. Dans le cas contraire, les évènements qui n'ont pas de consommateurs immédiats sont moins importants et peuvent être gérés par d'autres composants. Ainsi nous optimisons la taille des queues de consommateurs enregistrés auprès du gestionnaire des évènements et minimisons la durée d'attente des consommateurs bloqués dans l'attente de l'arrivée d'un évènement (ce sont les plus prioritaires). Sans prétendre que le gestionnaire garantit un comportement du type temps réel dur, ce gestionnaire minimise le temps de notification et la durée globale des blocages. Nous rappelons que le temps réel dur reste en dehors du cadre de cette thèse. La garantie d'un comportement du type temps réel dur nécessite généralement une analyse statique d'ordonnancement et l'établissement de garanties sur les temps d'exécution assez difficiles à gérer vu le nombre d'évènements pouvant être produits et consommés, l'établissement de ce genre de garanties dépend également de la qualité de service des détecteurs de défaillances.

Une entité intéressée par évènement commence par vérifier si l'évènement qui l'intéresse n'a pas déjà eu lieu. Dans le cas contraire, elle demande au gestionnaire des évènements de notifier l'occurrence de cet évènement (ou d'un autre évènement complexe contenant cet évènement). Le gestionnaire des évènements doit enregistrer ces demandes et propager, en fonction des conditions spécifiées, les évènements à chaque entité intéressée. Ce gestionnaire s'inspire du patron "Observateur" [19]. Si ce dernier permet à différentes entités se s'abonner pour être notifiées lors de changements d'états dans les sources, notre gestionnaire permet en plus une notification intelligente puisque nous nous notifions que les processus intéressés, et permet une mise à jour

des "sources" lors de la consommation d'un certain type d'évènements (arrivée de messages).

Les trois paragraphes suivants donnent plus de détails sur la gestion des évènements de suspicion des participants et d'arrivées de messages, ainsi que celle des évènements complexes définis à partir de ces deux types.

Suspicion d'un processus Les évènements de suspicion d'un ou plusieurs processus sont produits par les différents détecteurs de défaillances. A la détection de la défaillance d'un processus un évènement est produit et acheminé vers le gestionnaire des évènements. Ce dernier notifie toutes les entités intéressées par la suspicion de ce processus en attente d'un tel évènement. Toute entité consommatrice (par exemple une implémentation d'un algorithme de consensus) commence par voir si le processus ne figure pas dans la liste de suspects du détecteur de défaillances local. Si ce n'est pas le cas, elle bloque jusqu'à l'arrivée de cet évènement (ou d'un autre évènement, par exemple l'arrivée d'un certain message. Ce cas motive la définition et l'utilisation des évènements complexes, comme nous le verrons plus loin). Pour ce faire, cette entité s'enregistre comme consommatrice d'une évènement de type "suspicion d'un processus" et précise l'identité de ce processus.

Arrivée d'un message Plusieurs algorithmes nécessitent l'attente d'un message ayant certaines caractéristiques comme le type ou l'expéditeur. A l'arrivée d'un message, le composant responsable de l'échange des messages crée un évènement contenant ce message et notifie le gestionnaire des évènements. S'il existe une entité consommatrice intéressée par l'évènement, le message est alors délivré. Dans le cas contraire, il sera stocké dans la file des messages au niveau du gestionnaire des messages (rappelons que le gestionnaire des évènements ne stocke pas les messages).

Toute entité consommatrice peut spécifier les caractéristiques du message qu'elle souhaite recevoir. Les caractéristiques sont par exemple, l'émetteur, le numéro de séquence et le type. L'acheminement des messages doit satisfaire deux exigences. D'une part un message ne doit être stocké que s'il n'existe pas de consommateurs intéressés. D'autre part, un message ne doit pas être délivré à plus d'un consommateur, même s'il répond aux critères définis par plusieurs entités. Pour répondre à ces deux exigences, les requêtes exprimées par les consommateurs sont satisfaites dans l'ordre de leurs arrivées (premier arrivé premier servi) et le gestionnaire des évènements informe le producteur (c'est à dire le gestionnaire des messages) si son message a été consommé ou pas.

Évènements complexes Comme nous l'avons mentionné dans le paragraphe 4.1.3.1, il peut être nécessaire d'attendre simultanément l'occurrence d'un événement parmi plusieurs. Les évènements complexes sont nécessaires pour garantir la vivacité des différents algorithmes. Ils permettent aux différentes entités de ne pas bloquer éternellement dans l'attente de l'arrivée d'un seul évènement dont l'occurrence n'est pas sûre, mais plutôt sur un ensemble d'évènements simples mais complémentaires (dans le sens qu'inévitablement, l'un de ces évènements aura lieu) par exemple l'arrivée d'un message *ou* la détection d'une défaillance.

Le gestionnaire des évènements fournit une interface permettant de récupérer un évènement satisfaisant à une parmi plusieurs conditions donné en paramètre. Une entité intéressée par un ensemble d'évènements commence toujours par vérifier si l'un de ces évènements a déjà eu lieu. Si ce n'est pas le cas elle envoie une requête au gestionnaire des évènements en s'enregistrant comme consommatrice. Le gestionnaire des évènements débloque cette entité à l'occurrence d'un parmi les évènements attendus. Les autres évènements (correspondant aux autres conditions passées

en paramètre) ne sont plus intéressants pour le consommateur (à moins d'une autre requête explicite) et peuvent par conséquent être acheminés vers d'autres consommateurs potentiels ou stockés au niveau des producteurs s'ils ne sont pas consommés.

4.1.3.4 Conclusion

Afin de garantir les interactions entre les différents composants tout en minimisant le couplage entre ces derniers, nous avons isolé les différents évènements produits et consommés par les composants essentiels de notre architecture : consensus, détection de défaillances et échanges de messages. Cette architecture facilite le travail d'implémentation des algorithmes puisqu'ils disposent d'une interface adéquate pour la consommation et la production des évènements nécessaires à leur fonctionnement. Elle garantit également un acheminement et une gestion performante des différents évènements, contrairement à d'autres schéma comme la scrutation. Le comportement temporel des différents algorithmes de consensus et de détection de défaillances sera décrit dans la troisième partie de ce manuscrit.

Nous avons défini dans cette section deux types d'évènements. Il est possible d'en rajouter d'autres. Par exemple il est possible de définir un nouvel évènement qui sera généré par un temporisateur (`timer`) à l'expiration d'un `timeout`. Un évènement de ce type permettra de s'abstraire des détails internes des temporisateurs.

Le gestionnaire des évènements ne fait pas partie des interfaces des différents composants de l'architecture. Les producteurs et les consommateurs d'évènements l'utilisent d'une manière implicite. L'étude architecturale que nous avons menée ainsi que la gestion rigoureuse des différentes interactions minimise le couplage entre les différents composants. Ce couplage faible, associé aux possibilités de configuration des composants appartenant aux différents niveaux de notre architecture augmente la flexibilité et sera d'une grande aide lors de l'intégration du service du consensus dans l'architecture schizophrène.

4.2 Intégration dans l'architecture schizophrène et cas d'applications

Cette section s'intéresse à l'utilisation pratique du service de consensus que nous avons présenté et dont nous avons détaillé l'architecture dans la section précédente. Afin de présenter les problématiques pratiques de l'utilisation d'un service du consensus, nous commençons par décrire la notion de groupe de participants nécessaire à l'exécution de tout algorithme de consensus. Nous nous intéressons ensuite à l'intégration de ce service dans l'architecture schizophrène. Cette intégration élargira les possibilités de configuration et d'utilisation de ce service que nous détaillerons ensuite. Nous nous intéressons enfin à l'utilisation de notre service dans une application basée sur `CORBA`, puis à l'utilisation de ce service pour mettre en oeuvre certains styles de réplication de `FT CORBA`.

4.2.1 Groupe de participants

Lors de la définition de l'architecture du service du consensus, nous avons défini deux couches. La première comprend les différents composants de ce service. La seconde couche fournit les services intergiciels de base réalisant les abstractions nécessaires au bon fonctionnement du service du consensus.

Pour utiliser un algorithme de consensus donné, il faut tout d'abord choisir et configurer les différents composants et services garantissant le bon fonctionnement de l'algorithme choisi. Nous traitons les différents aspects de configuration dans le paragraphe 4.2.3.

La notion de participant est très importante pour tout algorithme de consensus. Un participant peut être identifié par deux ensembles de paramètres. Le premier représente les différentes ressources de calcul et de stockage disponibles (mémoire, processeur, etc.), ces ressources sont nécessaires à l'exécution des composants des algorithmes de consensus et de détection de défaillances. Le second ensemble de paramètres permet à chaque participant d'échanger des données avec le monde extérieur. Ces deux ensembles suffisent pour identifier un participant et pour lui permettre de participer à un consensus.

Nous utilisons le terme *partition* pour identifier ces deux ensembles de données. Pour une application critique, une partition identifie une unité fonctionnelle physiquement ou logiquement indépendante disposant de ressources de calcul et de stockage. Les partitions fonctionnent en isolation les unes des autres dans l'espace (une partition n'a pas d'accès direct aux données d'une autre partition) et dans le temps (une partition ne souffre pas de la surconsommation des ressources dans les autres partitions). Les partitions peuvent être définies sur plusieurs machines, sur la même machine et même en concurrence au niveau d'un seul processus ; dans ce cas les deux formes d'isolation peuvent être assurées par un noyau de séparation (separation kernel) comme dans [3]. Dans notre cas, nous avons effectué le travail nécessaire pour supporter plusieurs participants en concurrence déployés sur un seul processus, c'est la raison pour laquelle nous avons retenu ce terme.

Pour pouvoir participer à un consensus, chaque participant doit avoir un certain nombre de données concernant la configuration globale du groupe auquel il appartient. Parmi les paramètres nécessaires à l'exécution d'un algorithme de consensus, nous trouvons les différents paramètres de déploiement et surtout les informations de liaison définissant les associations entre les identifiants des participants et leurs localisations. Les informations sur les localisations permettent aux différents participants d'échanger des données. Ces informations dépendent du protocole de transport utilisé. Par exemple, pour des protocoles comme `UDP` ou `TCP`, elles peuvent se résumer par exemple aux adresses `IP` et aux numéros de ports. Ces informations, ainsi que d'autres paramètres dépendant de la nature de l'application et de la configuration du groupe (nombre total des processus dans le groupe, nombre maximal des processus autorisés à tomber en panne, etc.) sont alors nécessaires au fonctionnement de toute application souhaitant utiliser un algorithme de consensus.

Chaque participant doit disposer des différents éléments de la configuration du groupe dont il est membre afin d'initialiser le composant responsable des échanges des messages. Cette étape est très importante pour le fonctionnement des algorithmes de consensus et de détection des défaillances. Les paramètres de configuration du groupe peuvent être obtenus, soit statiquement, par exemple à partir d'un fichier de configuration, soit dynamiquement, en utilisant un serveur de configuration. Ces deux modes s'inspirent de méthodes couramment utilisées dans les intergiciels. Le premier correspond à l'utilisation du système de fichiers pour transmettre l'adresse d'un service. Le second correspond à un service de nommage. Ces deux modes de configuration dépendent de l'application et du protocole utilisés pour les échanges de données. Dans certains cas, il est souhaitable de ne pas fixer tous les paramètres statiquement (par exemple, la configuration du groupe, le schéma de déploiement, etc.) et de le faire dynamiquement lors de la phase d'initialisation des participants. Dans ces cas, l'utilisation d'un serveur central pour définir la configuration du groupe devient nécessaire.

Selon les caractéristiques de l'application (système opérationnel, schéma de déploiement, ressources disponibles, etc) et de l'environnement (architectures matérielles, liens de communi-

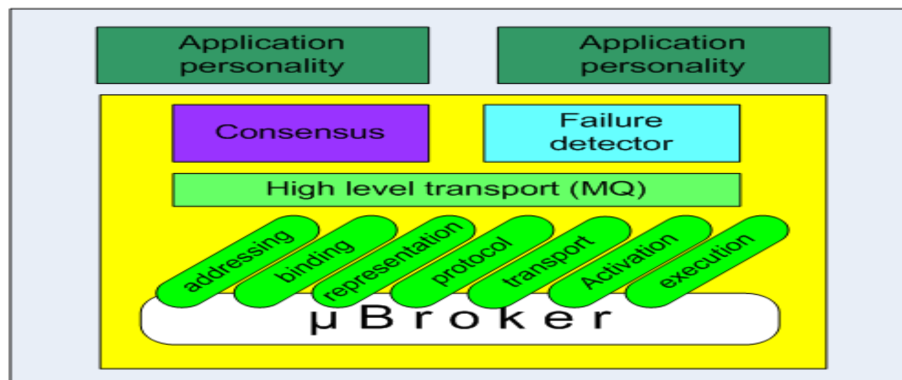

Fig. 4.8 – Intégration des composants dans l'architecture schizophrène

cation), il faut définir le protocole utilisé pour l'échange de données et les stratégies de gestion des ressources. Il faut aussi initialiser chaque partition en lui fournissant les paramètres du groupe et surtout les localisations des participants. L'intégration du service du consensus dans l'architecture schizophrène impose certains choix sur la définition des partitions, sur les interfaces fournies aux applications et sur les services utilisés pour l'échange des données. Nous traitons ces aspects dans la prochaine section.

4.2.2 Intégration dans l'architecture schizophrène

Comme le montre la section 3.2, l'architecture schizophrène définit une couche neutre permettant le découplage entre les aspects applicatifs et protocolaires des modèles de distribution. La couche neutre contient un ensemble de services fondamentaux pour la distribution, mais aussi un ensemble de services utilitaires comme celui de la gestion de la concurrence.

Afin de maximiser les possibilités de sa configuration, le service du consensus est placé dans la couche neutre au dessus des services canoniques de distribution. L'architecture résultante est montrée par la figure 4.8. D'une part, les entités applicatives peuvent l'utiliser comme toutes les autres abstractions de la couche neutre. Cette partie ne pose pas de problèmes puisque les interfaces des différents composants sont définies indépendamment de toute hypothèse sur l'entité qui l'utilise. D'autre part, les personnalités protocolaires ne sont pas visibles par notre service, ce dernier n'utilisant que les services neutres. Par conséquent, notre service est compatible avec toute personnalité protocolaire supportée par la couche neutre.

L'architecture schizophrène est une architecture générale fournissant plusieurs abstractions et services. L'intégration du service du consensus revient à utiliser cette architecture pour concrétiser les services intergiciels de base dont dépendent les composants de consensus, de détection de défaillances et d'échanges de messages. Nous avons vu que les services intergiciels de base peuvent à leur tour se décomposer selon leur rôle : distribution et interfaçage avec l'OS.

L'utilisation des services permettant d'abstraire l'OS ne pose pas de problèmes particuliers. Il suffit en effet de passer par les interfaces de ces services. L'utilisation des services de distribution est plus difficile. Il faut en effet résoudre deux problèmes résultant de la nécessité d'utiliser les services canoniques de l'architecture schizophrène pour échanger les messages entre les différents participants des algorithmes de consensus et de détection des défaillances. Ces services doivent être utilisés pour créer et référencer les participants. Comme nous l'avons détaillé dans le paragraphe 4.2, les paramètres de configuration du groupe et surtout les associations entre les identités des participants dans le groupe et leur localisation doivent également être déterminés

soit statiquement soit dynamiquement.

Pour permettre le fonctionnement correct du composant pour l'échange des données, il faut définir la notion de partition et permettre à chacun des participants d'accéder à la configuration du groupe. Il faut également choisir le protocole de transport et implémenter l'interface Transport définie dans le diagramme 4.6 [14]. Nous nous intéressons à la résolution de ces deux points sous les contraintes de l'architecture schizophrène dans les deux prochains paragraphes.

4.2.2.1 Partitions et groupes de participants

L'intégration dans l'architecture schizophrène impose le passage par les différents services canoniques de cette architecture pour les échanges des données entre les différents participants. En outre, il faut que la définition des participants et que la configuration du groupe restent indépendantes des différents modèles de distribution.

Dans l'architecture schizophrène, les données sont échangées entre les noeuds en passant par les requêtes. Pour recevoir une requête, chaque entité doit s'enregistrer auprès d'un adaptateur d'objet (voir la description des services canoniques, paragraphe 3.2.3). Chaque partition doit alors être attachée à un adaptateur d'objets. Par conséquent, nous définissons les partitions comme des servants particuliers. Leur particularité est qu'ils n'exécutent pas les requêtes, mais se contentent de récupérer les informations qu'elles contiennent. L'adaptateur d'objets gère alors le cycle de vie des différents participants au consensus, il coopère avec les différents services de la couche neutre pour échanger les données et pour construire les références désignant les partitions (et donc les différents participants aux consensus). Notons que ce schéma reste générique, indépendant des protocoles effectivement utilisés pour le transport des données (grâce à l'interface Transport) et indépendant de toute personnalité applicative. En effet, chaque personnalité applicative peut proposer un adaptateur d'objet particulier. Par exemple, pour la personnalité CORBA, les partitions sont attachées à un POA.

Pour simplifier la gestion du groupe, nous faisons l'hypothèse suivante : une fois le groupe des participants formé, il n'est pas possible de rajouter de nouveaux participants. Si la personnalité applicative ne permet pas de construire les objets de liaisons à partir de références connues à l'avance, un serveur dédié peut être utilisé pour établir et échanger les paramètres du groupe. L'utilisation d'un tel serveur ne sert qu'à cet objectif, il est en général arrêté une fois les partitions créées et les paramètres du groupe échangés. En plus, grâce aux propriétés d'interopérabilité de l'architecture schizophrène, le serveur d'initialisation peut être développé selon n'importe quel modèle de distribution.

4.2.2.2 Échanges des messages

Une fois le groupe configuré, les échanges de messages se font grâce aux différents services fondamentaux de la couche neutre. Dans le paragraphe précédent nous avons précisé que chaque partition peut être considérée comme un servant passif. Pour permettre au gestionnaire des messages de consensus d'assurer les échanges de messages, il faut fournir une concrétisation de l'interface Transport. Dans l'architecture schizophrène, les données sont échangées sous forme de requêtes. Nous avons défini deux fonctions permettant de passer des messages échangés par les différents algorithmes aux requêtes pouvant être gérées par les différents services de la couche neutre et inversement. La concrétisation de l'interface Transport basée sur la couche neutre de l'architecture schizophrène se fait en se basant sur les services fondamentaux de distribution

[14] cette interface n'est pas directement en relation avec le service de transport de PolyORB. Cependant, elle peut être concrétisée avec les services de la couche neutre

définis par cette architecture. Nous donnons dans le paragraphe 5.4.3.1 un exemple de concrétisation de l'interface `Transport` par la couche neutre de `PolyORB`. Cette concrétisation est basée principalement sur les services de représentation, de transport, d'activation et sur le μ`Broker`. Notons que cette concrétisation reste de haut niveau, car les services fondamentaux de la couche neutre définissent eux aussi des dépendances devant être résolues lors du choix du modèle de distribution.

Il est maintenant possible pour les différentes entités applicatives, indépendamment du modèle de distribution d'accéder à l'abstraction du consensus. Les messages échangés résultants de l'exécution des différents algorithmes de consensus et de détection de défaillances passeront par le même chemin que toutes les autres requêtes produites par les différentes entités applicatives instanciées. Les échanges de messages supportent tous les protocoles de transport de bas niveau (grâce aux services de transport et de protocole de la couche neutre). En particulier, ceux faisant partie des implémentations de `GIOP` comme `DIOP` et `IIOP` (implémentations de `GIOP` basées respectivement sur `UDP` et `TCP`).

4.2.3 Configuration de l'intergiciel

L'application vient avec un ensemble de besoins que l'intergiciel doit satisfaire. Si l'application nécessite une forme d'accord, un algorithme de consensus doit être sélectionné. Ce choix doit se faire en fonction des besoins de l'application mais aussi des contraintes imposées par l'environnement. Les hypothèses faites par les algorithmes doivent également être prises en compte.

Un processus de configuration doit commencer par l'analyse des besoins de l'application, l'algorithme de consensus et si besoin celui de détection de défaillances sont respectivement déterminés. Les services basiques de l'intergiciel sont ensuite configurés.

La configuration se fait en sélectionnant les versions des services et en fixant leurs paramètres pour satisfaire les objectifs globaux de l'application (performances, consommation des ressources, etc), les hypothèses que font les algorithmes de consensus et de détection des défaillances sélectionnés (synchronisme, modèles de défaillances, etc.) et les caractéristiques de l'environnement dans lequel évolue l'application (liens physiques entre les noeuds, etc.). Cette étape est très importante car elle doit concilier plusieurs dimensions de configuration :

- *Paramètres des algorithmes* comme les timeouts pour certaines classes de détecteurs de défaillances. La définition de ce paramètre est d'une importance extrême. D'une part, une très petite valeur peut mettre en cause la précision du détecteur de défaillances et peut causer une sur-consommation des ressources. D'autre part, une valeur très grande peut mettre en cause la durée de détection de défaillances et la durée de convergence de l'algorithme.
- *Transport.* Même si le choix du modèle asynchrone limite les exigences sur le protocole de transport, le comportement temporel des différents algorithmes dépend du protocole de transport. La possibilité de pertes de messages et les délais de transmissions sont des exemples montrant l'importance du choix et de la configuration du protocole de transport.
- *Exécution concurrente.* Le choix du service d'exécution concurrente (tasking) est important par exemple pour les applications nécessitant de limiter la consommation de la mémoire et/ou nécessitant des analyses statiques de leur comportement temporel. Ce sujet a été discuté dans 1.5.2.
- *Environnement et paramètres de déploiement.* L'infrastructure de déploiement de l'application cible impacte la synchronie et le modèle de défaillances. En outre, le nombre de participants au consensus a une influence directe sur les temps d'exécution.
- *Besoins spécifiques de l'application.* L'application cible peut utilise le consensus pour ac-

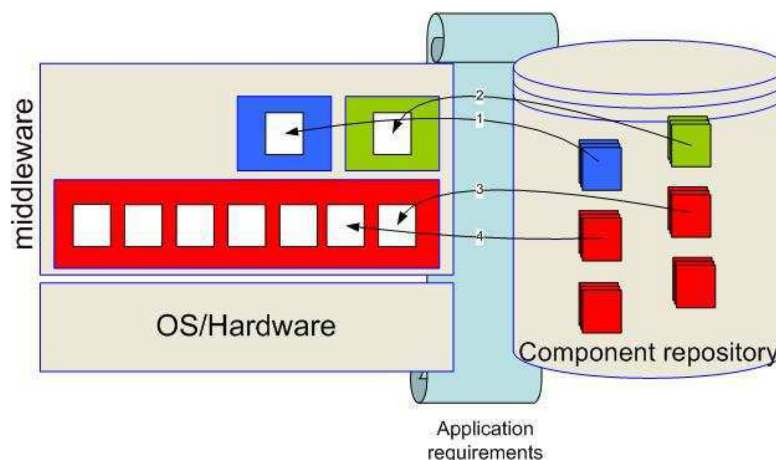

Fig. 4.9 – Configuration des composants intergiciels

complir d'autres objectifs comme la réplication ou la lecture cohérente sur un ensemble de capteurs. Ces objectifs doivent également être satisfaits.

L'architecture que nous proposons isole les principaux services, chacun dans un composant dédié et contrôle les interactions entre les différents composants. Par conséquent, cette architecture facilite la configuration de l'intergiciel. Le processus de configuration (voir figure 4.9) consiste en sélectionnant des services que nous avons conçu pour être totalement indépendants les uns des autres. Ceci motive la notion de "dépôt de composants" permettant d'avoir plusieurs versions de chaque composant. Par exemple, le composant fournissant le consensus existe en plusieurs versions, chacune implémentant un algorithme donné et faisant des hypothèses sur d'autres composants et services de l'intergiciel. Rappelons que grâce à l'architecture que nous avons proposés, un algorithme de consensus peut fonctionner avec plusieurs algorithmes de détection de défaillances sous réserve de compatibilité (voir paragraphe 4.1.2.3).

Le choix d'un algorithme de consensus doit se faire en respectant les besoins de l'application mais aussi les services que peut fournir l'intergiciel dans l'environnement où le produit final est sensé fonctionner. Le choix de l'algorithme de consensus fixe une partie de la configuration de l'intergiciel. L'autre partie est déterminée par les autres exigences imposées par l'application. Un problème de compatibilité peut, dans certains cas, avoir lieu. Le processus de configuration doit alors recommencer en utilisant, si possible, un autre algorithme. Ce processus de configuration peut facilement être supporté par plusieurs langages de description d'architecture. Dans ce cas, l'ensemble des composants du dépôt et surtout leurs propriétés doivent être modélisés, ainsi que les exigences et les paramètres de configuration. Certains travaux traitent de la problématique de description et de configuration de l'intergiciel en utilisant les langages de description d'architecture [127]. Ces travaux peuvent constituer une base pour une configuration (semi) automatique de l'intergiciel en fonction des besoins que présentent les applications. Cette automatisation sort du cadre de cette thèse mais constitue une perspective à notre travail.

Dans ce paragraphe, nous avons présenté les différents éléments permettant de profiter de la généricité du service de consensus. L'architecture claire de ce service permet son utilisation dans plusieurs contextes. Dans les prochains paragraphes, nous nous intéressons à certains cas d'utilisation. Nous prenons pour exemple la personnalité `CORBA`.

4.2.4 Application à la personnalité CORBA

La position des composants de consensus et de détection des défaillances dans la couche neutre et leurs dépendances de haut niveau envers ses services permet de profiter des avantages de l'architecture schizophrène garantissent le support de toutes les personnalités applicatives et protocolaires. L'utilisation du service du consensus dans le cadre de `CORBA` revient, d'une part, à résoudre les différentes dépendances de ce service en utilisation les composants et les interfaces supportés par ce standard et d'autre part, à configurer correctement le groupe des participants. Nous traitons la configuration du groupe plus loin dans cette section.

Une fois ces deux étapes effectuées, il devient possible d'effectuer des opérations comme la lecture cohérente des valeurs de sortie de capteurs par un ensemble d'entités répliquées compatibles avec `CORBA` (ou tout autre modèle de distribution supporté par l'architecture schizophrène). En outre, il est possible de se plier à plusieurs contraintes comme celles du profil `Ravenscar`. Les applications résultantes peuvent alors être déployées sans difficulté sur des systèmes opératoires comme ORK ou MarteOS.

Pour la personnalité `CORBA`, nous nous basons sur un serveur d'initialisation permettant aux différents participants de s'échanger identifiants et `IOR`. L'utilisation de ce serveur permet aux différents participants de s'enregistrer auprès du `POA` afin d'obtenir des `IOR` leur permettant de s'échanger des données puis demander un identifiant pour pouvoir participer aux instances du consensus. L'extrait de code 1 montre une interface `IDL` pouvant être implémentée afin de

Extrait de code 1 Interface du serveur d'initialisation

```
1  module Group_Communication{
2    struct Participant{
3      short Identifier; // Identifiant du participant dans le groupe
4      string The_Ref;   // IOR sous forme de chaîne de caractères
5    };
6
7    typedef sequence<Participant> Participant_Seq;
8
9    struct Group_Properties {
10     short Group_Id;                // Identifiant du groupe.
11     short Process_Count;           // Nombre des processus dans le groupe.
12     short Correct_Process_Count;// Nombre des processus corrects.
13     float Timeout;                 // Valeur par défaut pour la détection
14                                    // des défaillances.
15     [...]
16     Participant_Seq Participants;// Liste des participants.
17   };
18
19   interface Group_Initializer
20   {
21     Group_Properties Register (in short id, in string The_Ref);
22     Group_Properties Get_Group_Config ();
23     short Get_Member_Identifier ();
24   };
25 };
```

permettre l'initialisation du groupe de participants au consensus. La spécification que nous proposons définit les paramètres nécessaires à la configuration et à l'exécution des différents services. Ces paramètres peuvent être des simples paramètres d'algorithmes comme par exemple le `timeout` d'un détecteur de défaillances ou le nombre maximal de processus pouvant tomber en panne. Parmi ces paramètres, nous trouvons également la liste des associations entre les références et les identifiants des participants. Le serveur d'initialisation implémente une interface contenant trois

méthodes permettant aux participants de demander un identifiant, de s'enregistrer en utilisant cet identifiant et son IOR, et d'obtenir la configuration du groupe une fois celle ci établie (il s'agit surtout de garantir que les différentes informations de localisation sont connues par les différents participants). Le serveur d'initialisation attend que tous les participants soient enregistrés pour permettre l'accès à la configuration du groupe. En faisant ce choix nous garantissons que tous les participants se sont enregistrés et que des messages ne se sont pas perdus à cause de ce problème d'initialisation. La défaillance d'un participant, lors de l'initialisation, cause certes un blocage, mais nous pensons que redémarrer l'application est nettement plus avantageux que de la lancer avec l'handicap de devoir se passer d'un participant pendant tout son cycle de vie.

4.3 Conclusion

Dans ce chapitre, nous avons proposé une architecture pour un service de consensus générique. Nous avons défini les principaux composants pour le service du consensus. Nous nous sommes intéressés au rôle de l'intergiciel comme support aux différents algorithmes de consensus que nous avons étudiés. Pour augmenter la configurabilité de ce service nous nous sommes basés sur plusieurs patrons de conception et nous avons limité les dépendances entre les différents composants de consensus, de détection de défaillance et de gestion de messages formant ce service.

Pour maintenir un couplage faible entre les différentes entités de notre service, nous nous sommes basés sur une architecture orientée évènements. Cette architecture modifie les objectifs des composants de détection de défaillances et de gestion des évènements qui ne sont plus de maintenir simplement des listes de suspects et des queues de messages. Il doivent également assurer la propagation des modifications de l'état du système dès leur occurrence. L'introduction d'un gestionnaire d'évènements que nous avons également conçu en s'inspirant de patrons de conception existants permet d'accentuer le principe de la séparation des préoccupations sans nuire à la qualité des interactions. Le comportement temporel sera étudié à travers les mesures de performances que nous détaillerons dans la troisième partie de ce manuscrit.

La seconde partie de ce chapitre s'est intéressée à l'intégration de notre service dans l'architecture intergicielle schizophrène. La clarté de notre architecture ainsi que l'identification précise des besoins des algorithmes de consensus en terme de services intergiciels de base nous a été d'une grande aide lors de cette intégration. L'architecture du service du consensus définit en effet des services intergiciels de base qui doivent être concrétisés pour assurer des fonctions de base comme le transport des données et le support des exécutions concurrentes. L'utilisation des services de la couche neutre de l'architecture schizophrène résout ces différentes dépendances. Il est important de rappeler que la résolution de ces dépendances reste encore de haut niveau, car certains services de la couche neutre doivent être à leurs tours concrétisés en fonction du modèle de distribution (par exemple choix du POA comme adaptateur d'objet et du protocole IIOP pour la personnalité CORBA, etc.) et des besoins de l'application (modèle de concurrence, etc.).

L'intégration dans l'architecture schizophrène augmente les capacités d'adaptation du service du consensus. Les dimensions de configuration de ce service ont été décrites et un processus de configuration et d'assemblage prenant en compte les différents besoins d'applications susceptibles d'utiliser notre service a été explicité. L'étude de l'automatisation de ce processus fait partie des perspectives de notre travail et sera étudiée dans la conclusion de ce manuscrit. Pour illustrer l'utilisation de l'architecture schizophrène, un exemple d'application basé sur CORBA a été proposé. L'intégration dans l'architecture schizophrène implique la possibilité d'utiliser ce service avec toute combinaison de personnalités protocolaires et applicatives. Le processus de configuration ainsi que les différents cas d'utilisation de ce service montrent la validité des choix de conception

que nous avons effectués.

La troisième partie de ce manuscrit détaillera nos choix de mise en oeuvre et fournira des éléments supplémentaires validant les architectures proposées pour la mise en oeuvre de `FT CORBA` et du service de consensus.

Troisième partie

Réalisation et expérimentation

Chapitre 5

Composants et services pour la tolérance aux fautes et le consensus

Contents

5.1 Introduction 109
5.2 Ada et PolyORB 110
5.2.1 Le langage Ada 110
5.2.2 Architecture de PolyORB 112
5.3 Réalisation d'un service de tolérance aux fautes 114
5.3.1 Impact sur l'architecture de PolyORB 114
5.3.2 Gestion des groupes d'objets 115
5.3.3 Fonctionnalités avancées de GIOP 117
5.3.4 Gestion de la réplication et détection des défaillances 119
5.3.5 Styles de réplication 121
5.3.6 Construction d'applications basées sur FT CORBA 124
5.3.7 Conclusion 126
5.4 Réalisation d'un service générique de consensus 127
5.4.1 Composants du service du consensus 127
5.4.2 Gestion des évènements 132
5.4.3 Construction d'applications basées sur le consensus 133
5.4.4 Conclusion 138
5.5 Conclusion 139

5.1 Introduction

Dans ce chapitre, nous nous intéressons à la mise en oeuvre des différents services que nous avons conçu et décrit dans la seconde partie de ce manuscrit.

Dans une première section, nous présentons `Ada95` et `PolyORB`. Nous mettons l'accent sur les différents arguments qui ont motivé ces choix. D'une part, nous nous attardons sur les différents avantages offerts par le langage de programmation `Ada`, notamment au niveau architectural et au niveau du support des applications critiques et temps réel. D'autre part, nous présentons `PolyORB`, un intergiciel mettant en oeuvre l'architecture schizophrène que nous avons détaillée dans la section 3.2.

La seconde section s'intéresse à l'implantation de l'architecture du service de tolérance aux fautes que nous avons proposé dans le chapitre 3. Nous détaillons en particulier les principaux composants pour la gestion des groupes de répliques, des styles de réplication ainsi que pour la détection et la notification de défaillances.

La section suivante s'intéresse au composant générique du consensus. Dans cette section, nous mettons l'accent sur les principaux composants constituant le service générique de consensus proposé dans le chapitre 4. Nous nous intéressons en particulier aux interfaces de ces composants ainsi qu'à la mise en oeuvre de la gestion des évènements. Nous présentons enfin des cas pratiques de configuration et d'utilisation de ce service.

La dernière section présente le bilan du travail de mise en oeuvre ainsi que nos conclusions.

5.2 `Ada` et `PolyORB`

La réalisation des architectures des différents composants pour le consensus et la détection de défaillances nécessite le choix d' un langage de programmation et d'un intergiciel. Dans un premier paragraphe, nous présentons `Ada` et les différents avantages qu'il offre pour la conception et la réalisation d'applications. Nous présentons plusieurs points forts de ce langage. Nous nous intéressons en particulier à la définition des types, à l'encapsulation des données, au support de la programmation modulaire et des architectures basées sur les composants. Nous nous intéressons également au support de la concurrence et à la définition des restrictions qui sont des avantages faisant d'`Ada` l'un des meilleurs langages adaptés aux systèmes critiques et temps réel .

La seconde partie de cette section présente une brève description de `PolyORB`. Les concepts architecturaux de cet intergiciel ont déjà été présentés dans le chapitre 3.

5.2.1 Le langage `Ada`

La première version d'`Ada` a été conçue dans les années 80 en réponse à un cahier de charges défini par le Département de la Défense américain. Depuis deux révisions ont donné lieu à `Ada`95 et `Ada`05. `Ada` est un langage standardisé et tous les compilateurs doivent subir des tests de validation extrêmement rigoureux. Ceux-ci assurent la portabilité et la sûreté des applications écrites en `Ada`. `Ada` présente plusieurs autres avantages que nous détaillons ci dessous.

Typage et encapsulation `Ada` définit très peu de types natifs. Néanmoins, la création et l'utilisation de nouveaux types est très facile. `Ada` est fortement typé, il interdit le transtypage (`cast`) et permet de détecter plusieurs erreurs de codage dès la phase de la compilation. Pour encapsuler les données et structurer les applications, `Ada` propose les paquetages (`package`) comme structures fondamentales. Les paquetages `Ada` permettent l'application des règles de visibilité sur toutes les structures qu'ils contiennent. Les données peuvent être publiques, protégées (visibles uniquement par les descendants) ou privées.

Modularité, généricité et composants Les paquetages d'`Ada` se composent de deux parties : spécification et réalisation. Ces deux parties peuvent être séparées dans des fichiers distincts. Cette séparation facilite le prototypage et incite à se concentrer sur les aspects architecturaux sans se préoccuper de la réalisation. Les paquetages `Ada` peuvent être étendus. La définition de paquetages fils permettent d'étendre les fonctionnalités sans remettre en cause le travail déjà réalisé dans les paquetages ascendants. même si la programmation modulaire supportée par les

paquetages permet de répondre aux besoins de plusieurs applications, Ada supporte d'autres styles de programmations comme la programmation orientée objets.

Les paquetages Ada supportent également la notion de composants logiciels. Comme nous l'avons plus haut, la programmation modulaire est facilitée par ce langage de programmation. En outre, les interfaces et les dépendances entre les composants sont facilement exprimées. En effet, les fonctionnalités offertes sont définies par les spécifications des paquetages. Les dépendances peuvent être gérées et initialisées durant la phase d'élaboration.

Activités concurrentes Ada propose un moyen élégant pour programmer les processus concurrents. Il définit des constructions très puissantes comme les tâches et les objets protégés. Les tâches sont les unités d'exécution concurrentes. Contrairement aux fils d'exécution POSIX ([66]), les tâches s'intègrent avec les autres constructions Ada, leur synchronisation est beaucoup plus facile. Les objets protégés mettent en oeuvre le concept de moniteurs. Il devient alors facile d'implémenter plusieurs objets de synchronisation comme les verrous et les sémaphores. Ada permet également la mise en place de politiques d'ordonnancement complexes. De plus, il fournit des moyens efficaces pour gérer à la fois les aspects de la programmation système et de la programmation temps réel. Pour ces raisons, Ada est souvent utilisé dans les systèmes temps réel et embarqués nécessitant un haut niveau de sûreté de fonctionnement.

Profils et Restrictions Comme ϙu JAVA, Ada est un langage de programmation généraliste. Il offre certaines constructions peu ou pas adaptées aux applications nécessitant un niveau élevé de sûreté. Ces applications doivent avoir des garanties sur les temps d'exécution et sur les ressources consommées. Il est très difficile de prédire ces informations dès lors que certaines constructions du langage sont utilisées. Ada permet de placer des restrictions sur les constructions du langage autorisées. Les restrictions de type pragma Restrictions ou pragma Profile ont un impact sur la phase de la compilation qui vérifie que les constructions dangereuses sont évitées mais aussi sur l'exécutable généré qui est plus léger et qui s'exécute plus rapidement. Les restrictions possibles sont nombreuses et permettent par exemple d'éviter les allocations dynamiques de mémoire (assurant que les différents composants de l'application disposent de toute la mémoire dont ils ont besoin dès l'initialisation) ou encore de restreindre l'utilisation des constructions pour la concurrence afin de réduire les tailles des exécutables ou de permettre des analyses d'ordonnancement statiques (profil Ravenscar).

Support de la vérification formelle Ada présente trois caractéristiques favorisant son support de la vérification formelle : constructions normalisées, gestion efficace de la concurrence et support des restrictions.

Ada est l'un des meilleurs langages supportant la vérification formelle. Le support des restrictions et la facilité de leur mise en place permettent d'ajuster l'espace d'analyse à un niveau supportable par les outils de modélisation et de vérification sans perdre le pouvoir d'expressivité du langage. Spark [20] est un exemple de méthodes pour l'analyse et la vérification formelle. Il se base sur un ensemble de restrictions pour permettre l'analyse du flot d'exécution à partir d'instructions ajoutées sous forme de commentaires au code de base.

Avec Ada, il est possible d'extraire plusieurs informations depuis la structure du code et augmente l'efficacité d'outils permettant comme [26]. ASIS supporte en effet les analyses des flots de données ainsi que la vérification formelle. Cet outil peut également faciliter d'autres analyses comme l'utilisation de la mémoire et les analyses temporelles.

Il est également possible de générer des modèles formels partir du code Ada. Quasar [40] produit des modèles en réseaux de Petri décrivant les aspects de la concurrence dans les programmes Ada. Les modèles sont ensuite vérifiés afin de s'assurer de différentes propriétés du produit (absence d'interbloages, équité, etc.).

5.2.2 Architecture de PolyORB

Cette section présente PolyORB, l'intergiciel implémentant l'architecture schizophrène. Après une brève présentation de PolyORB, nous fournissons une description détaillée de l'architecture de PolyORB. Cette description servira, d'une part, à montrer l'impact de l'introduction de la tolérance aux fautes et d'autre part, à présenter les services que nous avons utilisés dans le cadre de la mise en oeuvre du service de consensus que nous avons proposé.

5.2.2.1 Présentation de PolyORB

PolyORB [100] est un intergiciel schizophrène. Même s'il est utilisé pour valider les principes de l'architecture schizophrène, PolyORB supporte plusieurs standards fréquemment utilisés par les industriels. PolyORB permet en effet d'instancier plusieurs modèles de distribution conformément à plusieurs standards tels que CORBA et l'annexe des systèmes distribués d'Ada95 (Distributed System Annex, DSA). PolyORB supporte plusieurs personnalités applicatives et protocolaires.

- Personnalités applicatives : PolyORB supporte CORBA, DSA, MOMA un intergiciel orienté messages (MOM) pour Ada et AWS permettant la réalisation d'applications Web (serveurs Web, etc.).
- Personnalités protocolaires : PolyORB supporte SOAP et plusieurs instances de GIOP (la partie protocolaire spécifié par le standard CORBA). Les instances de GIOP implémentés dans PolyORB sont IIOP (se basant sur TCP/IP), DIOP (se basant sur UDP/IP) ainsi que MIOP (fournissant un support pour la communication de groupe utilisant le multicast).

PolyORB est un logiciel libre disponible sur htpp://polyorb.objectweb.org/ et sur https://libre.adacore.com/polyorb/. Afin de valider nos propositions et nos apports à l'architecture schizophrène, nous nous baserons sur PolyORB.

5.2.2.2 Vue globale de l'architecture

L'architecture schizophrène de PolyORB définit un ensemble de personnalités applicatives et protocolaires ainsi qu'une couche neutre. Dans la section 3.2, nous avons fourni une description de cette architecture détaillant les deux types de personnalités ainsi que les services canoniques de distribution de la couche neutre. Nous nous intéressons dans ce paragraphe à la mise en oeuvre de cette architecture en détaillant certains services utilitaires importants et en présentant les interactions entre les différents modules de cette architecture.

La figure 5.1 détaille la vision globale de l'architecture de PolyORB. Elle montre les différentes personnalités supportées ainsi que les principaux services de la couche neutre. Nous séparons les services de distribution des services utilitaires. Notons que ces services utilitaires sont fréquemment utilisés dans le code et ont généralement un impact important sur le comportement de l'application basée sur PolyORB.

5.2.2.3 Mise en oeuvre de la couche neutre

La couche neutre est l'élément central de l'architecture. Elle utilise des services neutres de point de vue de la distribution. Ces services sont généralement en relation indirecte avec les

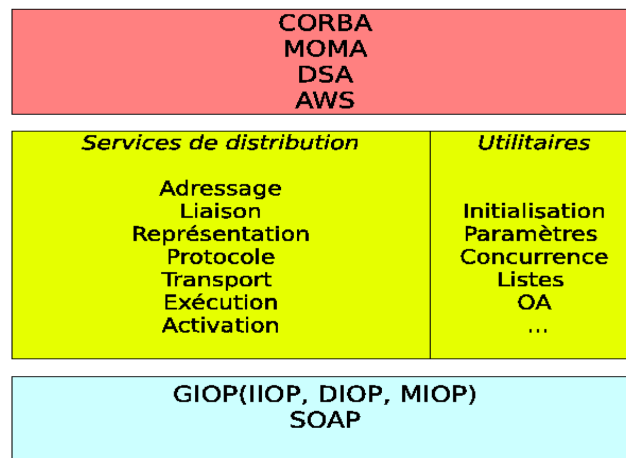

FIG. 5.1 – Composants de `PolyORB`

personnalités protocolaires et applicatives. Dans `PolyORB`, ce sont les personnalités qui dépendent de la couche neutre et non pas l'inverse. Pour obtenir cette inversion de dépendances, des règles de visibilité strictes empêchant chacune des personnalités d'accéder directement aux données des autres personnalités ont été mises en place. Pour assurer l'interopérabilité et la coopération entre les personnalités, des représentations neutres de toutes les données définies au niveau des personnalités et participant aux échanges de données sont mises en place. la relation entre les vues neutres des services et leurs concrétisations au niveau des personnalités est assurée grâce à l'utilisation d'étiquettes, à travers des fonctions de rappel ou grâce aux types abstraits et au mécanisme du polymorphisme.

5.2.2.4 Services utilitaires

Initialisation et assemblage Dans `PolyORB`, la construction de l'instance de l'intergiciel à déployer sur un noeud commence à la phase de compilation et finit pendant la phase d'exécution lors de l'élaboration d'un programme.

L'utilisateur sélectionne les différents paquetages correspondant à l'ensemble des composants requis pour le bon fonctionnement de l'application. Dans le cas d'`Ada`, cette sélection est permise en utilisant le mot clé `with` qui rajoute une dépendance sur le composant requis. Cette dépendance n'est généralement pas utilisée pour compiler l'application mais pour permettre l'enregistrement pour chaque composant d'une routine d'initialisation et d'une liste des dépendances.

Lors de l'élaboration du noeud, le service d'initialisation de `PolyORB` permet d'exécuter les procédures d'initialisation des différents composants finissant ainsi l'assemblage des composants requis par l'application et sélectionnés par l'utilisateur. Le service d'initialisation procède également à un ensemble de vérifications en s'assurant par exemple que toutes les dépendances sont correctement résolues et qu'aucun cycle ne soit détecté dans le graphe des dépendances. Une fois les différentes étapes d'initialisation finies, l'application peut commencer sa propre exécution.

Paramètres de configuration `PolyORB` dispose d'un service facilitant la définition et le chargement de paramètres de configuration. Les valeurs des paramètres peuvent être définies dans un fichier, en tant que variables d'environnement, mais également à partir d'un paquetage `Ada` ou

tout autre mécanisme de stockage de paramètres. Ce service, est chargé en premier par le service d'initialisation permettant ainsi aux unités qui le souhaitent de compléter leur initialisation en fonction des paramètres définis par l'utilisateur.

Concurrence Le service de concurrence de `PolyORB` s'inspire de l'interface normalisée `POSIX` (1003.4) et propose un ensemble d'interfaces abstraites permettant la définition et l'utilisation de tâches, de variables conditionnelles et de verrous. Ce service supporte trois politiques d'exécution : tâche unique, `Ravenscar`, et parallélisme général [100].

Le profil sans tâches assure l'utilisation d'une seule tâche par l'ensemble des composants de l'intergiciel. Le choix de ce profil ne nécessite aucune primitive de concurrence au niveau de l'exécutif. Ce profil est à utiliser de préférence pour limiter la consommation des ressources et le non déterminisme. Cependant, il propose un modèle très restrictif ne répondant pas au besoin d'un grand nombre de personnalités et d'applications cibles.

Le profil avec parallélisme général profite de la richesse de la bibliothèque de concurrence offerte par `Ada`, mais au prix d'un coût au niveau du déterminisme et de la consommation des ressources. Ce coût peut être trop fort pour les applications nécessitant une analyse statique d'ordonnancement ou un comportement temporel déterministe. En outre l'exécutif peut ne pas supporter certaines des fonctionnalités autorisées par ce profil.

Le profil `Ravenscar` est un juste milieu entre les deux profils, il propose une concrétisation compatible avec le profil `Ravenscar` des différentes interfaces du service. L'utilité du profil `Ravenscar` a été étudiée notamment pour les applications critiques (paragraphe1.5.2)

Listes, dictionnaires, adaptateurs d'objets `PolyORB` propose également un ensemble riche de paquetages génériques définissant des structures de données couramment utilisées. En cas de besoin, il est possible d'instancier des listes chaînées, des tableaux dynamiques, mais également des tables de hachage dynamiques parfaites ([63]), des dictionnaires et des adaptateurs d'objets génériques[100].

5.3 Réalisation d'un service de tolérance aux fautes

La mise en oeuvre du support de `FT CORBA` dans `PolyORB` nécessite la résolution de deux types de problèmes. D'une part, il faut gérer les problèmes architecturaux résultants de l'intégration des nouveaux composants proposés par le standard. D'autre part, il faut une maîtrise totale du flot des données, surtout lors de l'occurrence de défaillances. Cette section présente l'impact de l'introduction de `FT CORBA` sur l'architecture schizophrène de `PolyORB` ainsi que les composants que nous avons mis en place pour assurer les différentes fonctionnalités requises par le standard.

5.3.1 Impact sur l'architecture de `PolyORB`

Lors de la définition de l'architecture du service de tolérance aux fautes (chapitre3), nous avons opté pour une stratégie non intrusive basée sur l'interception. Cette stratégie a l'avantage de limiter l'impact de la tolérance aux fautes sur l'`ORB` et sur le code de l'application. La figure 5.2 montre les composants qui ont été touchés lors de la mise en place de ce service. Le service de gestion des paramètres a été sujet d'une légère évolution permettant une meilleure interaction avec les fichiers de configuration. Ces fichiers peuvent être mis à jour avec de nouveaux paramètres ou avec des mises à jour des paramètres. Ces modifications ont été introduites pour faciliter les tests et notamment la propagation de la référence du gestionnaire de réplication et la résolution

de cette référence par les clients. Au niveau des services fondamentaux de la couche neutre, le service d'adressage a subi une évolution pour supporter la construction des références sur les groupes d'objets, la construction de ces références sera détaillée dans le paragraphe 5.3.2.

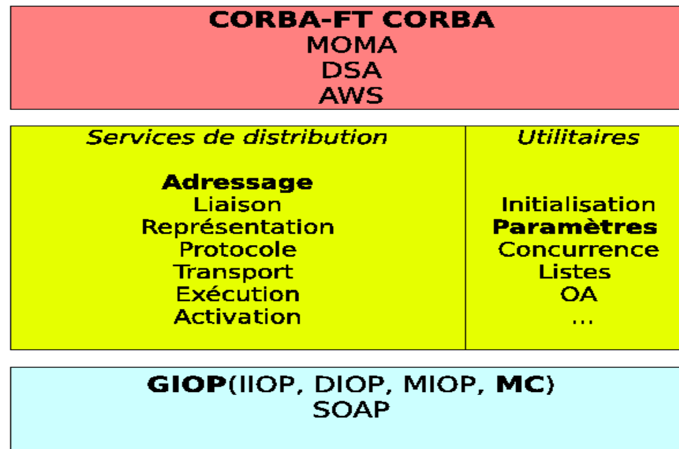

Fig. 5.2 – Impact de `FT CORBA` sur l'architecture de `PolyORB`

La personnalité `CORBA` a été naturellement augmentée avec des composants définis par `FT CORBA`. Nous traitons la mise en oeuvre de la gestion de la réplication et des défaillances dans le paragraphe 5.3.4. La définition des intercepteurs permettant la mise en oeuvre des styles de réplication sera détaillée dans le paragraphe 5.3.5.

En ce qui concerne `GIOP`, nous avons défini et mis en oeuvre les structures permettant l'encodage de l'information concernant le groupe d'objets dans les profils `GIOP` (paragraphe 5.3.2). Nous avons également contribué à la définition et à la mise en oeuvre de certains services permettent le support des ré-émissions des requêtes surtout en cas de défaillances. Nous détaillerons nos contributions dans le paragraphe 5.3.3.

5.3.2 Gestion des groupes d'objets

`FT CORBA` définit la notion de groupe d'objet comportant plusieurs répliques fournissant le même service. Un groupe d'objets est lui même un objet `CORBA`, sa référence doit être vue comme référence sur un simple objet. Dans ce paragraphe, nous nous intéressons à la structure des références sur les groupes d'objets et nous détaillons la mise en oeuvre de ces références dans `PolyORB`.

Définition 23 (`IOR`) *Pour référencer les objets, `CORBA` définit les `IOR` (Interoperable Object Reference). Il s'agit d'une chaîne de caractère, représentant une référence sur un objet donné et pouvant être décodée par afin de localiser et invoquer des opérations sur l'objet référencé. La forme des `IOR` est standardisée par `CORBA`.*

Définition 24 (Profil) *Un profil est une structure fournissant toute l'information nécessaire à la communication avec l'objet qu'il désigne. Un profil comprend toute l'information sur la pile protocolaire utilisée, le point d'accès et l'identifiant de l'objet au sein de l'`ORB`. Un profil peut contenir des composants étiquetés.*

Définition 25 (Composant étiqueté (`Tagged Component`)) *Les composants étiquetés sont des composants optionnels pouvant faire partie des profils. Ils sont souvent utilisés pour encapsuler d'une manière interopérable des informations diverses (par exemple le nom du groupe d'objets auquel appartient la réplique désignée par le profil).*

Définition 26 (`IOGR`) *Une `IOGR` est une `IOR` représentant une référence sur un groupe d'objets.*

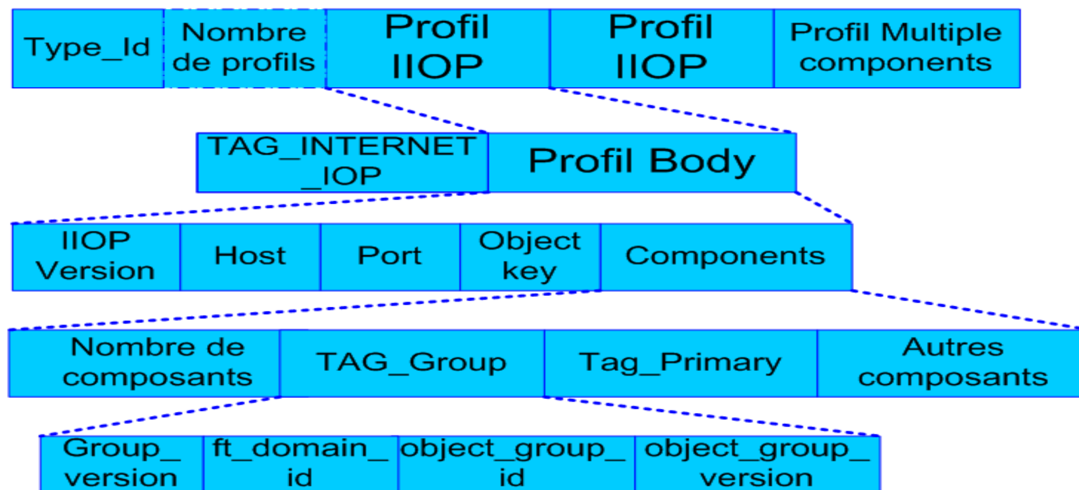

Fig. 5.3 – Structure d'une `IOGR`

La figure 5.3 détaille la structure d'une `IOGR` typique. La structure de l'`IOGR` est standardisée par la norme `CORBA` et permet, d'une manière interopérable, de référencer et de refléter la composition des groupes d'objets tout au long de leur cycle de vie. La construction et la mise à jour des `IOGR` est un élément clef pour la gestion de la réplication. En effet, la malformation ou la corruption d'une `IOGR` peuvent causer la perte du groupe d'objets qui est n'est joignable que grâce à cette référence. Les conséquences des malformations ou des corruptions des références sont plus graves que la défaillance d'une ou même de plusieurs répliques. Une mise en oeuvre correcte des `IOGR` nécessite la résolution des problèmes suivants :

- Identifier le groupe d'objets et retrouver ses propriétés à partir d'une référence. Ce problème se résout grâce à l'ajout de composants étiquettes aux profils `IIOP` de la référence.
- Identifier le groupe d'objets et retrouver ses propriétés même lorsque la référence ne contient pas de profils `IIOP`. Ce problème se résout grâce à l'emploi des profils `MultipleComponents`.
- Accèder et mettre à jour des informations sur le groupe d'objets tout au long de son cycle de vie.
- Prise en compte des contraintes de l'architecture schizophrène lors de la résolution des trois premiers problèmes.

Pour chacun de ces quatre problèmes, nous présentons les solutions prévues par la norme ainsi que nos choix de mise en oeuvre.

Composants étiquetés Les composants étiquetés (`Tagged Component`) sont ajoutés à chaque profil de l'`IOGR`. Ils encodent deux types d'information. le composant étiqueté `Tag_FT_Group` encode l'identifiant et les propriétés du groupe d'objet auquel le profil appartient. Le composant étiqueté `TAG_FT_PRIMARY` indique si le profil en question est primaire.

Les composants étiquetés de PolyORB sont définis en dérivant un type de base et en concrétisant des méthodes abstraites. Les structures de données sont directement inspirés de la norme. L'encodage et le décodage de ces structures de données vers et à partir d'une chaîne d'octets standardisée se base sur le service de représentation de PolyORB. Le mécanisme du polymorphisme permet de prendre en compte ces composants étiquetés et d'insérer lors de la génération d'IOR et d'IOGR.

Profils MultipleComponents Les profils de type MultipleComponents servent à maintenir l'information sur le groupe d'objets même lorsque ce dernier ne contient pas (ou plus) de réplique. Ce problème se pose par exemple à la création d'un groupe d'objets vide ou suite aux défaillances de toutes les répliques d'un groupe. Les profils MultipleComponents ne servent pas à désigner les répliques, nous les implémentons comme sous composants de GIOP.

Nous avons introduit les profils MultipleComponents dans PolyORB. Nous avons choisi de les implémenter sous forme de sous profils de GIOP. Ce choix a permis de réutiliser tous les mécanismes fournis par GIOP pour gérer et encoder les Tagged Component. Ce choix est conforté par le fait que les profils MultipleComponents ne sont pas utilisés pour référencer des objets concrets.

Opérations sur les groupes d'objets et mise à jour des IOGR La création des références ainsi que l'encodage et le décodage de la composition et des propriétés des groupes d'objets à partir des IOGR constituent un élément essentiel pour la réplication. Les opérations changeant les propriétés des groupes d'objets doivent être reflétées immédiatement sur l'IOGR représentant ce groupe. Nous avons défini une structure de données neutre (indépendante de GIOP) et une interface de programmation permettant de mettre à jour les types étiquetés en fonction des changements. Les profils MultipleComponents sont ajoutés lorsque le groupe ne contient plus aucune réplique et retirés dans le cas contraire.

Impact sur le service d'adressage Les structures de données appartenant à la couche neutre ne doivent donc pas avoir de visibilité sur les structures appartenant aux personnalités. Or, les profils et les types étiquetés sont des structures propres à la personnalité GIOP. Nous nous basons sur les fonctions de rappel permettant d'ajouter, d'extraire et de mettre les informations de tolérance aux fautes dans les profils. En procédant de cette manière nous assurons la neutralité de la gestion des IOGR et facilitons le support de la notion de groupe d'objet par d'autres personnalités protocolaires que GIOP. En outre, nous avons vérifié que ces fonctionnalités sont d'ores et déjà applicables à des profils autres que IIOP (des tests ont été effectués pour DIOP).

Le code du service d'adressage n'as pratiquement pas changé durant cette phase. L'introduction de la notion de groupe d'objets nous a amené à définir des méthodes permettant de vérifier l'équivalence des références sur les groupes d'objets. Cette équivalence est vraie lorsque les identifiants des groupes d'objets et leurs domaines de tolérance aux fautes respectifs sont égaux. Cette équivalence est vérifiée en retrouvant ces deux informations depuis les profils de la référence grâce l'interface définie ci dessus.

5.3.3 Fonctionnalités avancées de GIOP

Outre les différents éléments mis en place pour la construction des IOGR, la mise en oeuvre de FT CORBA a nécessité l'utilisation de plusieurs mécanismes avancés de GIOP. Nous décrivons ici les Service Context, le mécanisme de LOCATION_FORWARD ainsi que la définition et la négociation

des modes d'adressage. Ces trois fonctionnalités assurent les ré-invocations transparentes des requêtes suite aux défaillances.

Service Contexts GIOP fournit le moyen de passer un ensemble d'informations spécifiques aux services. Les Service Context peuvent être ajoutés aux requêtes ou aux réponses, il est également possible d'enregistrer et de récupérer ces services au niveau des intercepteurs. Contrairement au Tagged Component qui sont spécifiques aux profils, les Service Context sont spécifiques aux requêtes et aux réponses.

FT CORBA définit deux Service Contexts : FT_GROUP_VERSION et FT_REQUEST. Le premier est utilisé par la réplique pour valider l'IOGR utilisé par le client pour envoyer la requête. En cas d'IOGR non valide, par exemple si elle désigne une version obsolète ou corrompue d'un groupe d'objets, le serveur réagit soit en levant une exception (si la version de l'IOGR utilisée est supérieure à celle de la réplique), soit en suggérant l'utilisation d'une nouvelle IOGR grâce au mécanisme de LOCATION_FORWARD. Le second Service Context est utilisé pour garantir qu'une requête ne soit pas invoquée plus qu'une seule fois. Une requête peut être en effet envoyée plus qu'une fois par un client. Ce Service Context peut être également utilisé pour définir des timeouts au niveau des requêtes.

LOCATION_FORWARD Le mécanisme de LOCATION_FORWARD permet la re-direction des requêtes vers d'autres objets non référencés par l'IOR ou l'IOGR utilisée par le client. Ce mécanisme permet en particulier d'avoir des références persistantes et de supporter la re-localisation des services. Ce mécanisme est totalement transparent pour le client. GIOP définit deux exceptions LOCATION_FORWARD et LOCATION_FORWARD_PERM, pouvant être levées par le serveur. Ces exceptions contiennent la nouvelle adresse de l'objet. La seconde exception engendre un changement permanent de la référence de l'objet au niveau du client. Ce mécanisme est d'une extrême importance pour la tolérance aux fautes dans FT CORBA. Il permet en effet, au client de se rendre compte de la re-configuration du groupe ayant généralement lieu en cas de détection de la défaillance d'une réplique. Il permet également une ré-invocation transparente de la requête en utilisant la dernière référence de l'IOGR.

Modes d'adressage de GIOP GIOP 1.2 étend les possibilités d'identification de la cible d'une requête en définissant deux nouveaux modes d'adressage (Profile et Reference). Les deux premières versions ne proposent que le mode d'adressage (Key), ce mode utilise la clé de l'objet pour transmettre la requête. Les modes Profile et Reference se basent respectivement sur les profils et sur l'IOR complète (en précisant toutefois le rang dans IOR du profil utilisé). Le serveur peut répondre à une requête par une demande de changement du mode d'adressage. Dans ce cas l'exception NEEDS_ADDRESSING_MODE est levée et la requête est renvoyée. Dans le cas de FT CORBA le mode d'adressage (Key) ne permet pas de transporter les différents paramètres de qualité de service comme les Tagged Component, un mode d'adressage plus fort est alors requis. Pour notre implémentation, nous nous basons généralement sur le mode d'adressage profil.

Nous avons contribué à l'architecture de PolyORB en proposant plusieurs solutions pour la mise en place de ces trois mécanismes de GIOP. En particulier, nous avons proposé une mise en oeuvre du support des différent modes d'adressage de ce protocole. Les différents cas d'utilisation présentés par des applications se basant sur FT CORBA ont été un excellent moyen de vérifier le bon fonctionnement de ces trois mécanismes.

5.3.4 Gestion de la réplication et détection des défaillances

5.3.4.1 ReplicationManager

Le gestionnaire de réplication (ReplicationManager) est un service très central dans l'architecture de FT CORBA. Ce service permet en effet de créer et de détruire les groupes d'objets répliqués mais également de gérer leur composition et leurs propriétés tout au long de leur cycle de vie.

La spécification de ce service hérite de trois interfaces IDL (PropertyManager, GenericFactory et ObjectGroupManager). PropertyManager est une interface pour la gestion des propriétés des groupes d'objets. GenericFactory permet la création de groupes d'objets et de répliques. ObjectGroupManager permet la manipulation de la composition des groupes d'objets.

Mise en oeuvre Le PropertyManager maintient, pour chaque groupe d'objets l'ensemble des propriétés qui lui sont associées. La norme définit quatre types de propriétés (pour tous les groupes d'objets, pour tous les groupes d'objets d'un certain type, lors de la création d'un groupe d'objets, lors de l'exécution du service répliqué). Elle définit également certaines règles de mise à jour. Par exemple, les définitions les plus spécifiques peuvent écraser les définitions les plus générales, et certaines propriétés ne peuvent être mises à jour pendant l'exécution du service.

La mise en oeuvre de l'ObjectGroupManager se base sur l'interface que nous avons proposée pour la gestion des IOGR (paragraphe 5.3.2). L'ObjectGroupManager gère en plus la notion de localité (location). Une localité peut être vue comme une région de confinement de fautes devant contenir au moins une réplique. Cette notion que nous avons mis en oeuvre propose à l'utilisateur un confort de programmation et lui permet d'ajouter des contraintes facilement vérifiables et même assurées automatiquement comme par exemple le nombre minimum de répliques par localité.

La GenericFactory définit une seule interface qui a un double rôle selon l'entité qui l'implémente. Lorsqu'elle est implémentée par le ReplicationManager, elle permet la création et la destruction de groupes d'objets. Elle peut également être implémentée par des usines locales afin de permettre la création de répliques.

Résultats Nous implémentons les différentes interfaces du ReplicationManager en respectant la norme. Son comportement a été vérifié grâce à plusieurs applications témoins, nous présenterons une application complète se basant sur FT CORBA et mettant en évidence le bon fonctionnement du ReplicationManager plus tard dans ce chapitre.

5.3.4.2 Détection et notification des défaillances

La détection des défaillances est requise par le ReplicationManager afin de garantir une vue cohérente des groupes d'objets. Un détecteur de défaillances doit pouvoir surveiller plusieurs répliques simultanément. Le schéma de notification doit permettre la propagation des rapports d'erreurs jusqu'au ReplicationManager. Ces deux phases sont très importantes. En effet, le temps de détection et de notification d'une défaillance peut selon le style de réplication avoir un impact plus ou moins important sur le temps de réponse global du service répliqué.

Détection des défaillances Nous avons choisi de mettre en place une détection de défaillances selon le mode pull. Ce mode est plus flexible que le mode push, en particulier, il n'impose pas la connaissance des détecteurs de défaillances par les répliques (voir paragraphe 3.3.4). Pour le

Extrait de code 2 Interface pour la détection des défaillances

```
interface FaultDetector{
    void Register_Monitorable_Object(
     in Object Ref,
     in _TypeId Type_Id,
     in ObjectGroupId Object_Group_Id,
     in FTDomainId FT_Domain_Id,
     in Location The_Location,
     in TimeBase::TimeT Monitoring_Interval,
     in TimeBase::TimeT Timeout);

    void Unregister_Monitorable_Object(in Object ref);
};
```

mode pull, la norme prévoit que les répliques implémentent l'interface PullMonitorable afin de permettre aux détecteurs d'envoyer les requêtes de “suspicion”.

La nature synchrone des invocations CORBA rend la mise en oeuvre de détecteurs de défaillances difficile. Dans ce modèle, il est possible d'avoir des invocations qui ne se terminent jamais. Nous proposons une solution prenant en compte ce problème. Le nombre de répliques à surveiller étant généralement assez grand, il devient nécessaire de définir plusieurs tâches et de gérer la concurrence qui résulte de leur exécution simultanée. La solution que nous proposons permet également à l'utilisateur de choisir de limiter le modèle de concurrence au profil Ravenscar ou non.

L'extrait de code 2 montre l'interface que nous proposons pour enregistrer et demander l'arrêt de la surveillances d'objets CORBA. L'utilisation de cette interface nécessite que l'objet passé en paramètre implémente PullMonitorable. Cette interface peut être utilisée par le ReplicationManager pour demander la surveillance des répliques. Lors de l'enregistrement, toute l'information concernant la réplique (localité, groupe d'objets, référence) ainsi que les caractéristiques temporelles de la surveillances sont passés. Ces informations servent d'une part lors de l'établissement des rapports d'erreurs à propager lorsqu'une défaillance est détectée. D'autre part elle définissent les fréquences d'envoi de ces requêtes *is_ alive* ainsi que le timeout pour chaque réplique.

Pour chaque réplique à surveiller, le détecteur de défaillances alloue deux tâches. La première est une tâche “esclave”, responsable d'invoquer la méthode *is_ alive* au niveau de la réplique. La seconde tâche est “maîtresse” car elle décide du début de chaque cycle de détection et de la défaillance ou non de la réplique surveillée.

Afin de gérer les temporisations tout en restant compatibles avec le profil Ravenscar, nous généralisons le gestionnaire de temporisations proposé dans [17]. Le temporisateur que nous proposons permet en effet de gérer l'expiration de plusieurs timeouts selon nos besoins. Ce temporisateur permet à la tâche maître d'annuler une demande de notification. Ce cas se présente si la réplique est encore fonctionnelle et si la tâche esclave réussit son invocation.

A l'expiration d'un timeout, le temporisateur retrouve la tâche ayant demandé la notification, vérifie si cette dernière n'a pas annulé sa demande et signale une variable conditionnelle permettant de débloquer la tâche. La même variable conditionnelle est également utilisée par la tâche esclave afin de signaler l'arrivée d'une réponse. La tâche maîtresse est en effet bloquée dans l'attente de l'un de deux évènements : expiration d'un timeout ou arrivée d'une réponse à la requête envoyée. A son réveil, la tâche maîtresse détermine l'évènement ayant eu lieu, et décide si la réplique est vivante ou pas. Notons que la tâche maîtresse et la tâche esclave se synchronisent au début de chaque cycle.

Nous profitons de ce schéma afin de détecter trois types de fautes pouvant avoir lieu. La

défaillance d'une réplique ou l'occurrence d'une exception lors de l'invocation de *is_alive* et enfin le cas où *is_alive* renvoie une réponse négative. Ce cas n'est pas envisagé par la norme mais permet aux répliques de s'isoler du groupe auquel elles appartiennent (ce cas est utile par exemple si la réplique sait que le service qu'elle fournit n'est plus correct).

Lors de la conception des détecteurs de défaillance, nous avons garanti leur compatibilité avec le profil `Ravenscar` tout en nous basant sur les interfaces de haut niveau proposées par `PolyORB`. Le service de concurrence de `PolyORB` peut être configuré pour permettre à l'utilisateur de choisir, selon ses besoins, entre un profil de concurrence complet et un profil de concurrence compatible `Ravenscar`.

Notification des défaillances Nous nous basons sur le service de notification de `CORBA` pour assurer la propagation des rapports d'erreurs depuis les détecteurs de défaillances jusqu'aux gestionnaires de réplication. Le détecteur de défaillances s'enregistre comme fournisseur d'évènements auprès d'un canal de transmission maintenu par le notificateur des défaillances. Nous nous basons toujours sur le mode push pour la notification des défaillances (à ne pas confondre avec la détection qui se fait en mode pull). A l'occurrence d'une erreur, le détecteur de défaillances crée un rapport d'erreur qui arrive au `ReplicationManager` en passant par le canal des évènements. Une tâche spécifique est alors réveillée afin de traiter le rapport d'erreurs. Elle enlève la réplique défaillante du groupe d'objets, et crée une nouvelle `IOGR` correspondant à la nouvelle version du groupe d'objets répliqués. D'autres actions peuvent également avoir lieu selon les propriétés du groupe d'objets. Par exemple, si un nombre minimal de répliques est défini, il peut être nécessaire de créer une ou plusieurs répliques suite à la notification d'une défaillance.

5.3.5 Styles de réplication

5.3.5.1 Définition des intercepteurs

Les intercepteurs permettent d'appliquer des traitements additionnels d'une manière transparente à l'`ORB` et à l'application. Dans le paragraphe 3.3.3, nous avons défini deux types d'intercepteurs : coté client et coté serveur. Nous donnons ici les différents traitements appliqués par les intercepteurs.

Coté client Le pseudo code 3 montre les différents traitements que subit une requête au niveau de l'intercepteur coté client. Comme toutes les requêtes sont interceptées, il commence par s'assurer que la requête doit subir des traitements en vérifiant si elle est destinée à un groupe d'objets. Si c'est le cas, il détermine le style de réplication du groupe d'objets de destination. Pour optimiser les envois, nous faisons la différence entre les styles de réplications actifs et passifs pour lesquels nous appliquons deux politiques différentes. La requête est en effet envoyée à une seule réplique en cas de réplication passive et à toutes les répliques dans les cas de styles de réplication actives. Lors de l'occurrence d'un changement dans la composition du groupe d'objets, l'exception `LOCATION_FORWARD_PERM` est levée par les différentes répliques (encore opérationnelles). Dans ce cas, la requête est de nouveau traitée en utilisant la nouvelle `IOGR` du groupe d'objets.

Coté serveur Le pseudo code 4 montre les différents traitements que subit une requête lorsqu'elle est interceptée au niveau des répliques. Tout d'abord le mode d'adressage de la requête est vérifié. Nous nous basons en effet sur le mode d'adressage Profile. L'exception `NEEDS_-ADDRESSING_MODE` est levée si le mode d'adressage de la requête entrante n'est pas adéquat. Cette exception est en fait une demande de ré-envoi de la requête par le client. Si le mode d'adressage

Extrait de code 3 Intercepteur coté client

```
À l'arrivée d'une requête
Si (Requête interceptable) Alors
  | Ajouter les Service Context de la requête
  | Déterminer le style de réplication en utilisant la référence
  | Répéter
  |   | [Appliquer les traitements spécifiques au style de réplication]
  |   | Re-diriger vers l'intercepteur spécifique
  |   | Si (LOCATION_FORWARD_PERM) Alors
  |   |   | Mettre à jour la destination de la requête
  |   | Fin Si
  | jusqu'à ce que (Succès de l'invocation ou LOCATION_FORWARD_PERM ou échec définitif)
Fin Si
```

Extrait de code 4 Intercepteur coté serveur

```
À l'arrivée d'une requête
Si (Requête interceptable) Alors
  | Déterminer le mode d'adressage de la requête
  | Si (Mode d'adressage non supporté) Alors
  |   | Exception NEEDS_ADDRESSING_MODE retournée au client
  | Fin Si
  | Récupérer les Service Context de la requête (FT_REQUEST et FT_GROUP_VERSION)
  | Déterminer la version du groupe d'objets
  | Déterminer le style de réplication (en contactant le ReplicationManager)
  | Si (Version du groupe supérieure à celle de la requête) Alors
  |   | Exception LOCATION_FORWARD_PERM retournée au client
  | Fin Si
  | [Appliquer les traitements spécifiques au style de réplication]
Fin Si
```

est supporté alors les `Service Contexts` de la requête sont récupérés. Le est ensuite contacté afin de récupérer la version du dernier groupe d'objets. L'exception `LOCATION_FORWARD_PERM` est levée si le client avait utilisé une version obsolète de l'`IOGR` référençant le groupe d'objets. Des traitements additionnels pour la synchronisation des états et l'accomplissement des requêtes sont alors effectués selon le style de réplication.

5.3.5.2 Mise en oeuvre des styles de réplication

Dans le paragraphe précédent, nous avons proposé les traitements généraux effectués par les intercepteurs au niveau du client et du serveur. Nous présentons ici les différents traitements effectués par les intercepteurs coté client et coté serveur en fonction du style de réplication.

Réplication sans état Ce style de réplication est le plus simple à mettre en oeuvre puisque le service répliqué est supposé sans état. L'intercepteur client utilise l'`IOGR` du service et invoque la requête en utilisant l'un des profils. En cas d'échec, un autre profil est utilisé. Comme le service est sans état, la duplication des requêtes ne pose pas de problème. La gestion de `LOCATION_FORWARD` demeure cependant nécessaire afin de garantir la continuité du service lorsque la composition du groupe d'objets change.

Styles de réplication passifs Dans le cas d'un style de réplication passif, l'intercepteur du client n'envoie la requête qu'à la réplique primaire. Si le primaire est en panne alors nous distinguons deux cas. Le premier cas se présente si la panne empêche le client de fournir le service attendu mais que le mécanisme de `LOCATION_FORWARD` fonctionne encore (par exemple si la réplique primaire est au courant qu'elle est isolée du groupe) alors ce mécanisme est utilisé pour la ré-émission de la requête. Dans le cas contraire, pour compléter l'invocation, une autre réplique est choisie au hasard en utilisant l'`IOGR`.

Au niveau des répliques, le mécanisme d'interception se charge de toutes les opérations nécessaires à la journalisation des requêtes et à la synchronisation d'états se fait comme prévu par la norme. Pour la réplication passive à froid, l'état du primaire est stocké sur un support non volatile. Afin d'assurer la reprise lorsqu'un secondaire est promu, l'ensemble des requêtes reçues par l'ancien primaire depuis la dernière journalisation est rejoué. La réplication passive à chaud se base sur les mêmes principes sauf que la synchronisation des états se fait directement au niveau des secondaires (sans passer par le support de stockage). Notons que la sérialisation des états du primaire et l'opération inverse consistant à retrouver l'état des répliques à partir d'un tableau d'octets sont de la responsabilité de l'utilisateur. L'utilisateur peut se baser dans certains cas simples sur la sérialisation définie par `Ada` [71, 67].

Styles de réplication actifs Ces styles de réplication consistent à traiter les requêtes par toutes les répliques. Au niveau du client, l'intercepteur envoie la requête à toutes les répliques. Nous choisissons le résultat de l'une des réponses comme résultat final. Pour les mesures de performances, nous choisissons la dernière. Ceci permettra de comparer le comportement de ce style avec le style de réplication active avec vote. Le mécanisme de vote consiste en effet à récupérer toutes les réponses des serveurs, à les analyser pour choisir la réponse la plus correcte en faisant l'hypothèse que la majorité des serveurs font leurs calculs correctement.

5.3.6 Construction d'applications basées sur FT CORBA

Nous nous intéressons ici à la construction et à la configuration d'applications tolérantes aux fautes, basés sur PolyORB et respectant le standard FT CORBA. Nous détaillons les éléments de configuration et d'assemblage permettant de construire les différents noeuds de l'application tolérante aux fautes. Nous commençons par décrire les différents éléments permettant de configurer l'instance de PolyORB. Nous décrivons ensuite les éléments de configuration nouveaux introduits par notre mise en oeuvre.

5.3.6.1 Configuration des services dans PolyORB

L'architecture de PolyORB lui permet de supporter un grand nombre de politiques d'exécution. Outre le choix des personnalités applicatives et protocolaires, il est également possible de choisir les politiques d'exécution des requêtes, le modèle de concurrence, les services permettant l'interaction avec l'environnement (gestion des paramètres de configuration, entrées/sorties, etc.), adaptateurs d'objets, politique d'exécution du μBroker, etc [63]. Le choix des différents services, leur assemblage et leur configuration se font grâce aux services d'initialisation et de gestion des paramètres décrits dans le paragraphe 5.2.2.4. La configuration de ces services doit se faire en fonction des exigences de l'application, mais également en fonction de la nature de l'environnement (fonctions supportées par le système opératoire, ressources disponibles, restrictions imposées par le domaine de l'application, etc.). Par exemple, certains services supposent l'existence d'un système de fichiers ou le support de la concurrence par l'exécutif.

5.3.6.2 Configuration des applications tolérantes aux fautes

L'application basée sur FT CORBA se compose de deux parties : l'infrastructure de tolérance aux fautes et le code applicatif. L'utilisateur dispose général d'une grande liberté pour écrire le code de l'application en fonction de ses besoins. L'infrastructure de tolérance aux fautes est définie par le fournisseur de l'intergiciel et doit par conséquent être flexible pour répondre efficacement aux différents besoins.

Extrait de code 5 Exemple de services nécessités par l'application tolérante aux fautes

```
with PolyORB.Setup.Base;
with PolyORB.Setup.OA.Basic_POA;
with PolyORB.Setup.IIOP;
with PolyORB.Setup.Access_Points.IIOP;
with PolyORB.Binding_Data.GIOP.IIOP;
with PolyORB.Binding_Data.GIOP;
with PolyORB.Binding_Data.GIOP.Multiple_Components;
with PolyORB.Binding_Data.GIOP.Fault_Tolerance;
with PolyORB.Utils.Logging.File;

package body PolyORB.Setup.FT_Base is

end PolyORB.Setup.FT_Base;
```

L'extrait de code 5 montre un exemple de configuration de base des services de tolérance aux fautes. Outre les services de base et le choix des personnalités CORBA et GIOP, nous remarquons les dépendances sur les paquetages gérant les profils MultipleComponents et permettant de construire les IOGR (dépendance sur PolyORB.Binding_Data.GIOP.Fault_Tolerance). Le paquetage PolyORB.Utils.Logging.File définit le support de stockage non stable requis par la réplication

passive à froid : la sauvegarde et la restitution des états des répliques se font vers et depuis des fichiers. L'ajout de cette dépendance suppose que le système opératoire dispose d'un système de fichiers accessible en utilisant les constructions offertes par `Ada`.

Les choix de conception que nous avons effectués sont très peu intrusifs. Nous minimisons à la fois l'impact sur l'intergiciel et sur le code de l'application. Pour passer d'un client "classique" à un client tolérant aux fautes, il "suffit" de choisir le ou les styles de réplication qui doivent être mis en oeuvre par les différents intercepteurs. Ce choix s'effectue en rajoutant une dépendance vers les paquetages qui implémentent les styles de réplication. L'extrait de code 6 montre un exemple de choix possible chargeant les intercepteurs pour tous les styles de réplication.

Extrait de code 6 Chargement des intercepteurs coté client

```
with PolyORB.Setup.FT_Base;
with PolyORB.FTCORBA_P.Client.Replication_Styles.Active;
with PolyORB.FTCORBA_P.Client.Replication_Styles.Active_With_Voting;
with PolyORB.FTCORBA_P.Client.Replication_Styles.Passive;
with PolyORB.FTCORBA_P.Client.Replication_Styles.Stateless;

package body PolyORB.Setup.FT_Client is

end PolyORB.Setup.FT_Client;
```

Au niveau du serveur, les services intergiciels utilisés sont également chargés grâce à un paquetage. L'extrait de code 7 montre un exemple de configuration possible. Notons que les logiques de réplication coté client et coté serveur sont nécessitées car les répliques agissent comme clients tolérants aux fautes lorsque le `ReplicationManager` est répliqué. Pour l'infrastructure de tolérance aux fautes, il n'est pas autorisé de charger les intercepteurs implantant les styles de réplication. Ces intercepteurs mettront par exemple en péril le bon fonctionnement `ReplicationManager` et les détecteurs de défaillances. Même si l'infrastructure de tolérance aux fautes est sujette aux défaillances et doit être répliquée, il ne faut pas se baser sur ces intercepteurs car ces derniers dépendent de cette infrastructure. Ce point n'est pas traité par la norme et en constitue une limitation.

Extrait de code 7 Chargement des intercepteurs coté serveur

```
with PolyORB.Setup.FT_Base;

with PolyORB.FTCORBA_P.Server;
--  Server side interceptor

with PolyORB.Setup.FT_Client;
--  This dependency is needed by the replicas because they can act
--  FT-Clients e.g. when invoking a replicated RM.

package body PolyORB.Setup.FT_Server is

end PolyORB.Setup.FT_Server;
```

Comme nous pouvons le constater à travers les différents exemples de configuration, les possibilités d'adaptation de l'intergiciel ne sont pas amoindries par l'ajout de la tolérance aux fautes (à part bien sûr l'utilisation des personnalités `CORBA` et `GIOP`). Le choix des autres services comme la concurrence, la gestion des paramètres et le stockage des données se font en toute liberté en réponse aux exigences et contraintes de l'application cible. Le prochain paragraphe

présente les limites de la norme et les perspectives d'évolution de la mise en oeuvre actuelle.

5.3.6.3 Limites et perspectives

La conception et la mise en oeuvre du service de tolérance aux fautes dans `PolyORB` ont permis une étude poussée de plusieurs notions théoriques mais aussi architecturales et pratiques autour de la tolérance aux fautes dans les systèmes distribués en général et de son support au niveau des intergiciels en particulier. Nous présentons certains éléments permettant de valider les différentes fonctionnalités et d'évaluer le comportement temporel de certaines configurations dans le chapitre suivant.

Tout au long de la conception et de la mise en oeuvre de ce service, nous avons constaté les limites suivantes : réplication de l'infrastructure de tolérance aux fautes, consommation des ressources et dépendance vis à vis de la programmation orientée objets. Ces limites peuvent empêcher l'utilisation de `FT CORBA` dans le cadre d'applications critiques.

L'utilisation de l'orienté objets pose un problème pour le déterminisme des applications critique (voir 1.5.2). Cette technologie supporte idéalement plusieurs concepts définis par les normes `CORBA` et `FT CORBA`. Le code de `PolyORB` se base sur plusieurs constructions de l'orienté objet et il est très difficile de se passer de l'orienté objet sans passer une étude architecturale profonde de la norme et de `PolyORB`.

Le nombre important et la complexité des modules constituant l'infrastructure de la tolérance aux fautes peuvent poser des problèmes de consommation des ressources. Ce problème peut être accentué par deux points selon la nature et les besoins de l'application. D'une part l'application peut être, elle même, assez gourmande en ressources et peut également nécessiter un degré de réplication important. D'autre part, il peut être nécessaire, pour atteindre des niveaux de sûreté de fonctionnement élevés, de répliquer l'infrastructure de tolérance aux fautes. Les mécanismes de configuration peuvent limiter cette consommation mais il serait utile de disposer d'une version légère de `FT CORBA` limitant l a taille de l'infrastructure de tolérance aux fautes.

Le dernier point concerne la réplication de l'infrastructure de tolérance aux fautes. L'élimination des points uniques de défaillances dans les applications tolérantes aux fautes est nécessaire. Ceci nécessite la réplication de cette infrastructure et notamment le `ReplicationManager`. La norme ne donne pas d'indications pour résoudre ce problème. Même s'il sort du cadre de nos travaux, la réplication de l'infrastructure tolérante aux fautes en constitue une perspective. Cette dernière peut se baser sur le service de consensus générique que nous avons conçu.

L'introduction du service de tolérance aux fautes ne réduit pas les propriétés d'adaptation de `PolyORB`. Elle les enrichit avec toutes les propriétés de tolérance aux fautes définies par la norme, et elle impose des contraintes supplémentaires lors du choix et de la gestion des dépendances des différents modules. L'utilisation de langages de description d'architecture comme `AADL` pour la configuration de l'intergiciel est un sujet de recherche important et fait l'objet de plusieurs thèses. Certaines ont déjà été soutenues ([125]), d'autres sont en cours. Nous revenons sur ce point dans la conclusion de ce mémoire.

5.3.7 Conclusion

Dans cette section, nous avons présenté les principaux éléments de mise en oeuvre d'un service de tolérance compatible avec `FT CORBA`. Nous avons également présenté l'impact de la mise en oeuvre de ce standard sur l'architecture schizophrène.

Nous avons proposé et réalisé une architecture basée sur les interceptions pour mettre en place les différents styles de réplication. Pour la détection de défaillances, nous avons généralisé un

patron de conception pour garantir la compatibilité `Ravenscar` tout en permettent la détection simultanée des défaillances de plusieurs répliques. L'architecture que nous avons mis en place facilite l'application et la modification des styles de réplication. Cette stratégie de mise en oeuvre n'affecte le code de l'application et l'adaptation de l'intergiciel que très légèrement. Nous avons montré que cet effet reste limité à travers plusieurs exemples de configuration. La section suivante traite la réalisation du service générique de consensus que nous avons proposé dans le chapitre 4.

5.4 Réalisation d'un service générique de consensus

La réalisation du service de consensus doit répondre aux objectifs d'efficacité et de compatibilité avec les contraintes de hautes intégrité que nous nous sommes fixés lors de la conception de ce service. Nous nous basons sur `PolyORB` comme concrétisation des différentes services intergiciels de base. Cette réutilisation concerne principalement les fonctionnalités permettant l'encodage et le transfert de données à travers un support de communications.

Dans un premier temps, nous nous intéressons à la projection des différents éléments architecturaux sur le langage `Ada`. Nous présentons ensuite la mise en oeuvre proposée pour les différents composants de ce service. Dans un second paragraphe, la gestion des évènements et leur propagation entre les composants sera détaillée. Nous présentons enfin deux exemples de configuration de ce service.

5.4.1 Composants du service du consensus

Le chapitre 4 propose un ensemble d'éléments architecturaux permettant la conception d'un service de consensus générique. Les services proposés suivent le patron de conception "stratégie". Dans un premier temps nous décrivons les choix d'implantation basée sur `Ada`. Dans un second temps nous décrivons les trois modules de ce service (consensus, détection de défaillances et transport).

5.4.1.1 Projection sur `Ada`

La projection de l'architecture définit d'une part les modules implantant les différents composants et d'autre part la mise en oeuvre du patron "stratégie". Pour maximiser le déterminisme, nous nous interdisons toute les constructions orientées objets. Cette contrainte est généralement appréciée dans le cadre du développement des applications critiques (voir paragraphe 1.5.2).

Les paquetages `Ada` reflètent fidèlement la structure d'un composant. Les fonctionnalités fournies de chaque module sont visibles dans les spécifications. Les fonctionnalités requises par chaque module sont initialisées durant la phase de l'élaboration grâce à l'ajout de dépendances sur les paquetages adéquats. Nous nous basons sur le service d'initialisation de `PolyORB` pour gérer la dépendance entre les modules.

La mise en oeuvre la plus répandue du patron "stratégie" consiste à définir un objet abstrait et de le dériver pour chaque stratégie. Lors de la dérivation, les méthodes abstraites sont surchargées. Le mécanisme du polymorphisme permet d'appliquer les traitements correspondants à la stratégie en appelant la méthode qui correspond au type de l'objet concret. Cette mise en oeuvre ne répond pas à notre objectif car d'une part nous nous interdisons l'orienté objet, et d'autre part, le polymorphisme est l'une des structures les plus coûteuses lors des analyses et de la certification des applications. Nous proposons une mise en oeuvre de ce patron basée sur les pointeurs sous programmes et les types paramétrés d'`Ada`.

Extrait de code 8 Exemple d'implémentation du patron Stratégie

```
1  package Strategies is
2     type Execute_Body is access procedure;
3     type Strategy_Type is (A, B, C);
4     type Strategy (Disc : Strategy_Type) is
          record
5        -- Common data
6        case Disc is
7           when A =>
8              null;  -- A data
9           when B =>
10             null;  -- B data
11          when C =>
12             null;  -- C data
13       end case;
14    end record;
15    type Strategies is
16      array (Strategy_Type'Range)
17      of Execute_Body;
18    type Context is record
19       Registered_strategies : Strategies :=
20         (others => null);
21       Current_Strategy : Strategy_Type;
22    end record;
23
24    procedure Execute (C : Context);
25    procedure Register_Strategy
26      (C  : in out Context;
27       ST : Strategy_Type;
28       E  : Execute_Body);
29
30    procedure Set_Current_Strategy
31      (C  : in out Context;
32       ST : Strategy_Type);
33    CC : Context;
34 end Strategies;
35
36 package body Strategies is
37    procedure Execute (C : Context) is
38       begin
39          C.Registered_strategies
40            (C.Current_Strategy).all;
41       end Execute;
42
43       procedure Set_Current_Strategy
44         (C  : in out Context;
45          ST : Strategy_Type) is
46       begin
47          C.Current_Strategy := ST;
48       end Set_Current_Strategy;
49
50       procedure Register_Strategy
51         (C  : in out Context;
52          ST : Strategy_Type;
53          E  : Execute_Body) is
54       begin
55          C.Registered_strategies (ST) := E;
56       end Register_Strategy;
57
58    procedure Execute_A is ..
59    procedure Execute_B is ..
60    procedure Execute_C is ..
61 begin
62    Register_Strategy
63      (CC, A, Execute_A'Access);
64    Register_Strategy
65      (CC, B, Execute_B'Access);
66    Register_Strategy
67      (CC, C, Execute_C'Access);
68 end Strategies;
69
70 Set_Current_Strategy (CC, B);
71 Execute (CC);
72 Set_Current_Strategy (CC, C);
73 Execute (CC);
74 Set_Current_Strategy (CC, A);
75 Execute (CC);
```

Les types paramétrés sont des structures (type record) comprenant une partie discriminante constituée par un ou plusieurs paramètres formels. La définition et l'accès aux différents champs du type peut se faire en fonction des valeurs de la partie discriminante. Les types paramétrés constituent dans plusieurs cas une alternative à l'extension de type par héritage. Un type dérivé peut contenir les champs qu'il hérite de son père dans sa partie fixe et ses champs propres dans la partie dépendante de la contrainte. Les pointeurs sur sous programmes permettent de définir l'ensemble des fonctions à appeler pour chaque stratégie concrète. L'extrait de code 8 présente une implémentation possible de ce patron. Le contexte d'exécution dispose d'une table associant le nom de chaque stratégie aux pointeurs sur les sous programmes à appeler. Outre Execute permettant l'accès aux services, nous proposons une méthode permettant d'enregistrer les stratégies concrètes (Register_Strategy) et une autre (Set_Current_Strategy) permettant de définir ou de changer la stratégie d'exécution courante. Les lignes 62 à 67 décrivent l'enregistrement des différentes stratégies. Cet enregistrement se fait pendant la phase d'élaboration et assure que les différentes stratégies sont enregistrées avant tout appel de service. Enfin les lignes 70 à 75 illustrent l'accès au service et le changement dynamique des stratégies.

Cette implémentation évite les problèmes qui peuvent être posés par le polymorphisme. Toutes les instances du patron stratégie dans les différents du service du consensus seront développés selon cet idiome. Nous nous intéressons dans le reste de cette section à la mise en oeuvre de ces composants.

5.4.1.2 Composants de consensus et de détection de défaillances

Le composant du consensus présente une interface permettant de créer et de lancer des instances de consensus. A sa création, chaque instance est attachée à une partition. L'association entre les partitions et les instances permet aux différents algorithmes d'échanger des messages avec les autres membres du groupe. L'architecture de la mise en oeuvre de ce composant se base sur le mécanisme décrit dans le paragraphe précédent, sur la hiérarchie des paquetages et sur les mécanismes de paramétrage et d'initialisation de PolyORB.

Chaque algorithme est implémenté dans un sous paquetage de Consensus. Pour utiliser un algorithme, l'utilisateur ajoute une dépendance vers le paquetage lui correspondant. Le mécanisme d'initialisation de PolyORB assure que l'interface du composant principal dispose d'une concrétisation correspondant à l'algorithme choisi. La configuration des différents paramètres des algorithmes peut se faire grâce au mécanisme de gestion des paramètres de PolyORB. L'utilisateur peut charger plusieurs algorithmes et retarder le choix effectif des algorithmes ainsi que de leurs paramètres à la phase d'exécution. Le changement à la volée de l'algorithme est également possible par exemple lors du passage à un mode dégradé. Toutefois il nécessite un travail de synchronisation supplémentaire pour assurer l'utilisation du même algorithme par tous les participants. Notons qu'un changement de l'algorithme de consensus peut nécessiter le changement de l'algorithme de détection de défaillances et le chargement de nouveaux paramètres de configuration.

La mise en oeuvre effective d'un algorithme de consensus doit se baser sur les constructions et les interfaces fournies par l'intergiciel de base et par les autres composants du service. Par exemple, la création de tâches doit se faire en appelant le service de concurrence, l'attente sur plusieurs évènements soit se faire en utilisant l'interface du gestionnaire, etc. Nous avons mis en oeuvre quelques algorithmes de consensus se basant sur l'architecture que nous proposons.

Notre architecture supporte tout algorithme définissant un ensemble de messages, utilisant une ou plusieurs tâches échangeant ces messages, et nécessitant un ensemble de synchronisations (attente jusqu'à l'expiration d'un timeout, attente de l'arrivée d'un message, ou autre évènement

dans le système). Les différentes tâches s'exécutent au sein de partitions, chacune associée à un unique participant. Les participants sont identifiés par un entier. Cette identification facilite la mise en oeuvre des algorithmes et en particulier ceux qui nécessitent une relation d'ordre dans l'ensemble des identifiants des participants.

L'interface que propose le composant de détection de défaillances permet de créer une instance d'un détecteur de défaillances sur une partition donnée. Elle permet également de le lancer et de le stopper. L'arrêt du détecteur peut être nécessité en cas de changement d'algorithme ou simplement pour libérer des ressources qui ne sont plus utilisées.

L'architecture du composant de détection de défaillances se base sur les mêmes principes que celui du consensus. La seule différence entre les deux architectures est due à la propagation des évènements de suspicion. La suspicion d'un processus par un algorithme de détection donné est un évènement interne au composant de la détection de défaillances. Le composant de détection de défaillances (représentant le participant contexte dans le patron stratégie) fournit deux méthodes privées aux algorithmes pour leur permettre de mettre à jour la liste des suspects. Lorsque la liste de suspects est mise à jour, cette information est propagée en cas de besoin (si une entité applicative a demandé la notification de la suspicion d'un processus et si ce dernier est concerné par la mise à jour) au gestionnaire des évènements.

Grâce à notre architecture l'implémentation d'algorithmes de consensus comme le coordinateur tournant [22] ou celui proposé dans [85] ne nécessite que l'équivalent de 150 lignes de code. La mise en oeuvre d'algorithmes de détection de défaillances comme ceux proposés par ([22]) et par ([75] ne dépasse pas les 300 lignes. Une analyse plus complète de la distribution des lignes de code sera présentée dans le chapitre suivant (paragraphe 6.3.1).

5.4.1.3 Gestion des messages

Un extrait de l'interface de programmation de haut niveau utilisée pour l'échange des messages requis par les algorithmes de consensus et de détection de défaillances est présenté par 9. Le paquetage responsable de fournir ces abstractions se nomme Partition_Protocols. Il permet l'échange de messages entre les partitions logiques ou physiques.

La partie privée de ce paquetage illustre l'utilisation du service de contrôle de concurrence de PolyORB. Le type Runnable abstrait une unité d'exécution (par exemple une tâche Ada). Dans chaque partition, une unité d'exécution est dédiée pour la réception des messages entrants. Elle est également responsable de notifier le gestionnaire des évènements de l'arrivée de chaque message et de le stocker dans la liste de messages si ce message n'est pas consommé.

Grâce à cette interface, il est possible d'arrêter l'exécution des fonctionnalités de transport dans une partition. Il est également de possible de simuler des pertes de messages. Selon le taux de perte défini par l'utilisateur, un message peut être soit livré soit détruit. Il est donc possible de faire des tests poussés pour mesurer la réaction des différents algorithmes selon les taux de pertes de messages. Les taux de pertes peuvent également être définis dans le cadre d'un mode dégradé si les ressources de calcul sont limités ou si la consommation d'énergie est réduite.

L'interface fournit plusieurs fonctions de type Receive_Message. Chacune de ces fonctions permet la réception d'un message selon un certain nombre de critères. Chacune de ces fonctions peut être bloquante ou non. Dans le premier cas, elle bloque le fil d'exécution qui l'appelle jusqu'à la réception d'un message satisfaisant les critères. Dans le second cas elle vérifie la présence d'un tel message dans la queue. Un résultat nul peut être renvoyé si aucun message appartenant à la queue ne satisfait les conditions. Nous fournirons plus de détails sur notre mise en oeuvre basée sur les évènements de ces fonctions. Rappelons que ce composant est totalement indépendant du protocole effectif utilisé pour l'échange des données. Le bon fonctionnement de ce composant

Extrait de code 9 Transport des messages

```
package Partition_Protocols is

   function Create_Partition_Transport
     (Nid  : Process_Id) return Partition_Transport_Access;
   --  If the Partition_Transport exists return it else create a new one
   function Receive_Message
     (Self     : access Partition_Transport;
      Pattern  : Message_Pattern;
      Blocking : Boolean := False) return C_Message;
   function Receive_Any_Message
     (Self     : access Partition_Transport;
      Blocking : Boolean := False)
      return     C_Message;
   procedure Send_Message
     (Self : access Partition_Transport;
      M    : C_Message;
      To   : Process_Id);
   procedure Shutdown (Self : access Partition_Transport);
   procedure Simulate_Failure
     (Self : access Partition_Transport;
      TF   : Transport_Failure);
private
   type Partition_Transport is limited record
      --        [...]
      NId            : Process_Id;
      Lock           : PolyORB.Tasking.Mutexes.Mutex_Access;
      Message_Queue  : Message_Queues_Pkg.List;
   end record;
   type T_Runnable is new PolyORB.Tasking.Threads.Runnable with record
      T_Partition : Partition_Transport_Access;
   end record;
   procedure Run (R : access T_Runnable);
end Partition_Protocols;
```

Extrait de code 10 Gestion des évènements

```
package Event_Manager is

   type Event_Type is (Message_Arrival , Process_Suspicion , None);
   type Event (T : Event_Type := None) is record
      Partition : Process_Id := No_Id;
      case T is
         when Message_Arrival =>
            The_Message : C_Message;
         when Process_Suspicion =>
            The_Suspect : Process_Id := No_Id;
         when None =>
            null;
      end case;
   end record;
   type Event_Condition (T : Event_Type := None) is record
      Partition : Process_Id := No_Id;
      case T is
         when Message_Arrival =>
            Pattern : Message_Pattern;
         when Process_Suspicion =>
            Process : Process_Id := No_Id;
         when None =>
            null;
      end case;
   end record;
   type Event_Condition_Array is array (Integer range <>) of Event_Condition;

   function Wait_Event (ECA  : Event_Condition_Array) return Event;
   function Notify_Event (E : Event) return Boolean;
   function Satisfies_Condition
     (E : Event; C : Event_Condition) return Boolean;
end Event_Manager;
```

a été vérifié avec des protocoles de transport comme UDP, IIOP, DIOP ainsi qu'un protocole de transport se basant sur la mémoire partagée et qui sera présenté dans le chapitre suivant.

5.4.2 Gestion des évènements

Nous nous intéressons dans ce paragraphe à la définition et à la gestion des différents évènements échangés au sein du service du consensus. Le gestionnaire des évènements est implémenté par le paquetage Event_Manager. Dans ce paragraphe, nous nous intéressons principalement à son interface détaillée dans l'extrait de code 10. Cette interface permet aux différentes entités applicatives ou intergicielles de produire et de consommer les évènements. Nous nous intéressons également au filtrage des évènements et à la gestion d'évènements complexes du point de vue des producteurs et des consommateurs.

Nous utilisons le type paramétré (Event) pour représenter les évènements. Nous définissons deux types d'évènement correspondant respectivement aux arrivées de messages et aux suspicions de participants : Message_Arrival et Process_Suspicion. Pour permettre aux entités de définir les critères sur les évènements qu'elles souhaitent consommer, nous définissons également un troisième type paramétré (Event_Condition).

5.4.2.1 Production

Les producteurs notifient le gestionnaire de l'occurrence d'un évènement en utilisant Notify_Event. Cette fonction renvoie un booléen permettant de préciser si l'évènement a été consommé

ou non. Si un évènement n'est pas consommé, le producteur l'enregistre dans son état interne. Ce choix s'impose puisque le gestionnaire des évènements est considéré comme une machine sans états. Si l'évènement contenant un message n'est pas consommé, le message est stocké dans la queue des messages. Les évènements de suspicion non consommés ne posent pas de problème puisque la défaillance d'un processus est toujours visible depuis la liste des suspects exportée par le détecteur de défaillances.

5.4.2.2 Consommation

Les entités consommatrices commencent par vérifier si l'évènement qu'elles souhaitent consommer a déjà eu lieu en contactant directement les producteurs. Si cette vérification échoue alors le consommateur demande au gestionnaire des évènements de le notifier dès l'arrivée de cet évènement. La fonction Wait_Event est prévue à cet effet. Cette fonction est bloquante. Pour le gestionnaire des évènements, l'évènement est considéré comme consommé dès que l'entité appelante est débloquée.

Pour permettre l'évaluation d'un prédicat sur un évènement, nous nous basons sur la fonction Satisfies_Condition, elle évalue un évènement contre un critère. Par exemple, elle permet de préciser si un message est envoyé par processus particulier, le type des données transportées par ce message, etc. L'utilisation de cette fonction s'impose puisque même si l'évènement et la condition qui lui correspond n'ont pas le même type.

La fonction Wait_Event prend en paramètre un ensemble de conditions et renvoie un évènement satisfaisant l'un de ces critères. La définition de “liste de conditions” permet de gérer le cas des évènements complexes. Le gestionnaire des évènements décompose l'évènement complexe symbolisé par un ensemble de conditions, et attend la notification de l'occurrence d'un évènement satisfaisant l'une des conditions. Dans ce cas, cet évènement est re-envoyé comme résultat de Wait_Event.

5.4.2.3 Conclusion

La mise en oeuvre de la gestion des évènements permet une propagation fiable de l'information depuis les producteurs jusqu'aux consommateurs en passant par le gestionnaire des évènements. Nous avons prêté une très grande attention à la rapidité et au déterminisme des notifications. Nous avons opté pour un gestionnaire d'évènements très léger et comme dans tout le service du consensus, nous avons évité l'utilisation de l'orienté objet.

La gestion des évènements peut facilement être étendue afin de supporter la définition de nouveaux évènements mais aussi la définition de prédicats plus complexes sur les évènements. Notons que le simple prédicat dont nous avons eu besoin est l'occurrence de l'un évènement satisfaisant l'une des conditions définies dans une liste.

Dans le prochain paragraphe nous nous intéressons à plusieurs cas d'application du service du consensus dans plusieurs cas. Nous mettons en particulier l'accent sur les éléments de mise en oeuvre garantissant les propriétés de configurabilité et de généricité de ce service.

5.4.3 Construction d'applications basées sur le consensus

L'objectif de cette section est d'illustrer l'utilisation du service du consensus. Nous commençons par détailler deux interfaces permettant de gérer la configuration du groupe des participants d'une part, et d'abstraire les fonctionnalités du transport de données d'autre part. Nous montrons ensuite deux exemples de constructions d'applications basées sur ce service.

5.4.3.1 Transport de données

Le gestionnaire des message se base sur une interface fournissant un ensemble minimal de fonctionnalités assurant le transport effectif des données entre les partitions. Le diagramme UML de cette interface est proposé par la figure 4.6 page 91

La résolution de cette dépendance se fait d'une manière configurable grâce au patron stratégie permettant de définir et de choisir l'implémentation de cette interface qui satisfait le mieux aux besoins de l'application. Si d'application nécessite un haut niveau d'intégrité, la simplicité de l'interface permet de choisir un protocole transport adapté en termes d'utilisation des ressources et d'effort requis pour la validation.

Cette interface garantit la portabilité de tous les composants du service du consensus. Les concrétisations de cette interface peuvent se baser sur des protocoles de transport simples comme UDP ou alors utiliser toutes les fonctions de distribution d'un intergiciel .

Nous avons conçu un concrétisation de cette interface se basant sur le paquetage PolyORB.Partitions (extrait de code 11). Ce paquetage permet de passer par les services fondamentaux de la couche neutre pour assurer le transfert des données, permettant ainsi l'utilisation du service du consensus indépendamment du modèle de distribution. Cet extrait montre la définition d'une paritition comme un servant particulier. Bien entendu, il faut résoudre les différentes dépendances introduites par ce paquetage. En particulier, il faut fournir un adapteteur d'objets concrêt (par exemple le POA de CORBA), ainsi qu'une concrétisation du μBroker et des différents services abstraits de la couche neutre. Nous donnons un peu plus loin dans ce chapitre un exemple de configuration complet permettant d'exécuter le consensus en se basant sur la couche neutre de PolyORB (extrait de code 14).

5.4.3.2 Configurations locale et globale d'une partition

Les différents paramètres de configuration peuvent être classés en deux catégories : paramètres de configuration locaux propres aux partitions et paramètres globaux relatifs à la configuration de groupe. L'interface de configuration donnée par l'extrait 12 s'intéresse principalement à cette dernière catégorie de paramètres. Elle a été introduite pour permettre la configuration globale des différents paramètres des algorithmes mais également pour autoriser un changement dynamique de ces paramètres en cas de besoin.

Dans notre implémentation, les différents détails peuvent être récupérés d'une manière locale en analysant un fichier de configuration (statique) soit en ne se basant que sur un serveur d'initialisation comme décrit dans le paragraphe 4.2.4. Il est également possible de se baser sur ces deux modes à condition de séparer les paramètres globaux des paramètres locaux.

Pour permettre une configuration de groupe adaptée aux mécanismes de transport choisi, nous nous basons encore une fois sur le patron stratégie et sur le service d'initialisation de PolyORB. Une stratégie concrète de configuration consiste à implémenter deux méthodes permettant à tout processus de joindre un groupe de participants (Join) et d'accéder à la configuration du groupe (Get_Global_Config). Nous empêchons l'utilisation du service du consensus jusqu'à ce que la configuration du groupe soit effectuée. Cette configuration doit se faire en appelant les deux méthodes ci dessus. La première bloque le participant jusqu'à ce que tous les membres soient enregistrés. La seconde méthode permet à chaque module de transport d'associer l'identifiant du processus à une forme portable de son adresse (chaîne de caractère). Cette information est encodée et décodée par le module de transport bas niveau. Elle n'est utilisée que pour le transport effectif des données. La stratégie et les fichiers de configuration doivent alors être choisis selon le protocole de transport de données.

Extrait de code 11 Concrétisation de l'interface transport par la couche neutre

```
package PolyORB.Partitions is

   type Partition is new PolyORB.Servants.Servant with private;
   type Partition_Access is access all Partition;

   function Execute_Servant
     (Self : access Partition;
      Msg  : Message'Class) return Message'Class;

   function Handle_Message
     (Self : access Partition;
      Msg  : Message'Class) return Message'Class;

   procedure Create_Partition
     (P       : in out Partition_Access;
      Id      : Integer;
      The_ORB : PolyORB.ORB.ORB_Access);

   procedure Run (The_Partition : access Partition);

   --------------------------------------------------------
   -- Primitives for message exchange across partitions --
   --------------------------------------------------------

   procedure Send_Message
     (M           : C_Message;
      Self        : access Partition;
      Destination : String);

   function Receive_Message
     (Self : access Partition) return C_Message;

   procedure Wait_Message (Self : access Partition);

   procedure Abort_Wait_Message
     (Self    : access Partition;
      Aborted : out Boolean);

end PolyORB.Partitions;
```

Extrait de code 12 Éléments de configuration des partitions

```
package Partition.Configuration_Iface is
   type Participant is record
      Stringified_Ref : String_Ptr;
   end record;
   type Participant_Array is array (Partition_Id'Range) of Participant;
   type Global_Config is record
      P_Count         : Integer;    --  Proces count
      C_P_Count       : Integer;    --  Correct process count
      D_Timeout       : Duration;   --  Default timeout
      D_Timeout_Incr  : Duration;   --  Default timeout incrementation
      Participants    : Participant_Array;
   end record;

   type Join_Body is access
     procedure (Id : Partition_Id; My_Ref : String);

   type Get_Global_Config_Body is access
     function return Global_Config;

   type Partition_Configuration_Interface_Type is record
      Join              : Join_Body;
      Get_Global_Config : Get_Global_Config_Body;
   end record;
end Partition.Configuration_Iface;
```

5.4.3.3 Assemblage du service du consensus

Pour construire une application basée sur le service su consensus, nous nous basons sur les mêmes mécanismes que ceux utilisés pour `FT CORBA`. Ces différents mécanismes d'assemblage et de paramétrage de composants permettent une grande adaptation aux différentes caractéristiques des applications cibles à un coût relativement faible. Dans les prochains paragraphes nous analysons deux exemples de configuration permettant l'utilisation d'un service de consensus dans deux contextes différents. Le premier se base sur une configuration statique et sur `UDP` comme protocole de transport. Le second utilise un serveur de configuration et se base sur les services fondamentaux de la couche neutre pour les échanges des données.

5.4.3.4 Exemple 1 : Application embarquée

Nous commençons par décrire une configuration qui vise à répondre à des besoins de déterminisme et de faible consommation de ressources. Cette configuration est adaptée aux environnements embarqués avec un exécutif léger supportant le profil `Ravenscar`. Ce premier exemple d'assemblage se base sur une configuration statique. Les paramètres de configuration sont récupérés à partir d'un paquetage PolyORB.Parameters.Default. Ceci évite le problème potentiel du manque de support des entrées/sorties pour certains systèmes opératoires. Une description entière de la configuration est montrée dans l'extrait de code 13. Deux algorithmes de consensus et de détection de défaillances sont chargés, l'application pourra sélectionner n'importe quelle combinaison de ces algorithmes si la condition de compatibilité est satisfaite. Enfin, nous sélectionnons un modèle de concurrence compatible avec le profil `Ravenscar`.

5.4.3.5 Exemple 2 : Intégration dans l'architecture schizophrène

Le deuxième exemple de configuration permet à l'application d'utiliser les services de l'architecture schizophrène pour lancer des instances de consensus. Cette configuration est présen-

Extrait de code 13 Assemblage d'un service de consensus basé sur `UDP`

```
-- Transport initialization
with Partition.Transport.UDP;
with Partition.Configuration.Static;

-- Consensus algorithms
with Consensus.Rotating_Coordinator;
with Consensus.Perpetual_Accuracy;

-- Failure detection algorithms
with Failure_Detector.Larrea;
with Failure_Detector.Chandra;

-- PolyORB Setup
with PolyORB.Setup.Tasking.Ravenscar;
with PolyORB.ORB_Controller.Workers;
with PolyORB.ORB.Thread_Pool;
with PolyORB.Log.Stderr;
with PolyORB.Setup.Base;
with PolyORB.Parameters.Default;

package body Consensus_Setup is

end Consensus_Setup;
```

Extrait de code 14 Assemblage d'un service de consensus basé sur les services fondamentaux de `PolyORB`

```
-- Transport initialization
with Partition.Transport.PNL;
with Partition.Configuration.CORBA_Server;

-- Consensus algorithms
with Consensus.Rotating_Coordinator;
with Consensus.Perpetual_Accuracy;

-- Failure detection algorithms
with Failure_Detector.Larrea;
with Failure_Detector.Chandra;

-- PolyORB Setup
with PolyORB.Setup.Tasking.Full_Tasking;
with PolyORB.ORB_Controller.Workers;
with PolyORB.ORB.Thread_Pool;
with PolyORB.Log.Stderr;
with PolyORB.Setup.Base;
with PolyORB.Parameters.Default;

-- POA, GIOP/IIOP/DIOP
with PolyORB.Setup.OA.Basic_POA;
with PolyORB.Binding_Data.GIOP;
with PolyORB.Setup.IIOP;
with PolyORB.Setup.DIOP;
with PolyORB.Binding_Data.GIOP.IIOP;
with PolyORB.Binding_Data.GIOP.DIOP;

package body Consensus_Setup is

end Consensus_Setup;
```

tée dans l'extrait 14. Elle diffère de la configuration du paragraphe par les aspects suivants. D'abord, la configuration des partitions est effectuée grâce à un serveur CORBA. Ensuite, nous réalisons l'interface du transport en se basant sur la couche neutre de PolyORB. Le paquetage Partition.Transport.PNL implémente les quatre méthodes de cette interface en assurant la conversion des messages utilisés par les différents algorithmes depuis et vers des requêtes qui sont gérées par les différents services de la couche neutre de PolyORB. Nous sélectionnons ensuite les composants permettant d'initialiser le POA qui gère le cycle de vie des participants ainsi que d'autres composants permettant la gestion des objets de liaison et l'utilisation des protocoles IIOP ou DIOP pour l'échange des données. Notons enfin l'absence de restrictions sur la bibliothèque de concurrence (Full_Tasking).

Notons que dans les deux cas de configuration que nous avons présentés, le code de l'application reste pratiquement le même, ainsi que celui mettant en oeuvre les algorithmes de consensus. Les changements significatifs sont plutôt au niveau dans les caractéristiques de l'application et de son comportement temporel. Ces deux exemples illustrent un aspect important d'adaptation et de configuration de notre service de consensus. Ils valident également les différents choix architecturaux et de mise en oeuvre que nous effectuons. Une évaluation plus complète de ce service sera proposée dans le prochain chapitre.

5.4.4 Conclusion

Cette section a présenté les principaux éléments permettant une mise en oeuvre efficace de l'architecture que nous avons proposés dans le chapitre 4. Nous avons commencé par présenter une implémentation possible du patron de conception "stratégie" évitant le polymorphisme qui pose de sérieux problèmes lors de l'analyse des systèmes critiques. Nous avons ensuite décrit les principaux éléments de mise en oeuvre des différents composants. Nous notons l'utilisation intensive du patron stratégie. Nous nous sommes en effet basés sur ce patron dans les composants de consensus et de détection des défaillances pour permettre le support flexible de différents algorithmes. Ce patron a également été utilisé pour permettre la mise en oeuvre des interfaces de transport de bas niveau et de configuration selon plusieurs stratégies. L'architecture de notre mise en oeuvre assure l'application du principe de séparation des préoccupations notamment grâce à un ensemble de règles de visibilité strictes et à ce patron de conception. Notons que ce patron permet une re-configuration dynamique des services. Cette possibilité est intéressante pour la mise en oeuvre de mode de fonctionnement dégradé généralement nécessité par les systèmes tolérants aux fautes. C'est une perspective possible de nos travaux.

Concernant la propagation des évènements entre les composants du service du consensus, nous avons présenté une implémentation possible d'un gestionnaire d'évènements. Pour optimiser son comportement temporel, ce gestionnaire ne stocke pas d'évènements. Il les propage dès qu'ils sont signalés aux consommateurs intéressés. L'interface qu'il propose aux consommateurs ne souffre pas de complexité. Ces derniers définissent un ensemble de critères sur les évènements et peuvent se bloquer jusqu'à l'occurrence d'un évènement correspondant à l'un des critères. Dans le cadre des différentes implémentations, nous n'avons pas eu besoin de définir des évènements complexes autre que l'exemple cité ci-dessus. Nous pensons que nous avons déjà les structures de données permettant de gérer d'autres évènements complexes définis en connectant des évènements simples avec des opérateurs binaires sur les booléens (par exemple occurrence de deux évènements mais pas d'un troisième).

Nous avons ensuite présenté deux cas d'utilisation du composant du consensus. Ces deux cas montrent les larges capacités d'adaptation et de réutilisation présentés par ce service. La généricité et le comportement temporel seront amplement discutés dans le prochain chapitre.

5.5 Conclusion

Dans ce chapitre, nous avons détaillé les différents travaux de mise en oeuvre d'un service de tolérance aux fautes et d'un ensemble de composants fournissant l'abstraction du consensus. Dans un premier temps, nous avons présenté l'intergiciel `PolyORB` dont nous avons enrichi l'architecture pour supporter `FT CORBA` et dont nous avons réutilisé certains composants pour mettre en oeuvre le service du consensus. Nous avons également présenté le langage de programmation `Ada`95 en mettant l'accent sur ses avantages au niveau du génie logiciel ainsi qu'au niveau du support des besoins d'applications temps réel ou critiques.

Ensuite, nous avons présenté les principaux composants et modules que nous avons mis en place pour implanter `FT CORBA` en se pliant aux contraintes de l'architecture schizophrène. Cet effort nous a permis d'avoir une architecture claire basée sur le concept d'intercepteurs. Nous nous sommes particulièrement intéressé à la configuration des différents styles de réplication et proposé une mise en oeuvre basée sur le service de consensus.

Enfin, nous avons présenté les différents choix de mise en ouvre pour le service de consensus en détaillant les principaux composants et les mécanismes de gestion des évènements. Nous avons également illustré la généricité de notre service en détaillant deux exemples d'assemblages adaptés à deux cas d'utilisation différents.

Dans le chapitre suivant, nous nous intéressons à une évaluation plus précise du comportement temporel des différents composants des services de tolérance aux fautes et de consensus.

Chapitre 6

Validation et mesures

Contents

6.1 Introduction 141
6.2 Évaluation de la mise en oeuvre de `FT CORBA` 142
6.2.1 Gestion des groupes de répliques 142
6.2.2 Comportement temporel des styles de réplication 143
6.2.3 Détection et notification des défaillances 146
6.2.4 Conclusion 148
6.3 Évaluation du service générique de consensus 149
6.3.1 Analyse du code source 149
6.3.2 Mesures de performances 149
6.3.3 Conclusion 155
6.4 Conclusion 156

6.1 Introduction

L'objectif de ce chapitre est d'évaluer les différentes choix que nous avons faits pour concevoir et réaliser les services de tolérance aux fautes et de consensus. Les architectures de ces différents services ont été détaillées dans la deuxième partie de ce manuscrit, alors que les détails de mise en oeuvre ont fait l'objet du chapitre précédent.

Pour la validation du service de tolérance aux fautes compatible avec `FT CORBA`, nous proposons un ensemble de tests validant la construction des `IOGR` et la mise en oeuvre des différents styles de réplication. Nous effectuons ensuite des mesures permettant d'évaluer le coût de la réplication et les performances de la détection et la notification de défaillance.

Pour évaluer le service de consensus que nous avons proposé, nous effectuons plusieurs mesures correspondant à différents scénarios d'utilisation. Nous traitons en particulier une configuration minimaliste minimisant les sources de non déterminisme et à une autre compatible avec l'architecture schizophrène. Nous analysons également la répartition du code utilisé pour la mise en oeuvre de ce service. Cette analyse fournit des arguments supplémentaires quant à la validité de l'architecture et de sa capacité de supporter de nouveaux algorithmes et protocoles pour le consensus, la détection de défaillances et le transport des données.

Sauf indication contraire, les différentes mesures sont réalisées en utilisant une machine disposant d'un processeur P4 avec 1 GB de RAM. Cette machine supporte le système d'exploitation Linux (noyau 2.6.12-9-386). Les schémas de déploiement diffèrent selon les tests et seront détaillés

lors de la description des scénarios. Quand c'est possible, nous préférons des schémas de déploiement simples n'utilisant qu'une machine. Ce choix évite plusieurs problèmes qui se posent lors des mesures de performances dans des environnements distribués, notamment l'absence d'horloge globale et la nécessité de synchronisations supplémentaires qui en résultent.

6.2 Évaluation de la mise en oeuvre de FT CORBA

Cette section complète l'évaluation de notre mise en ouvre de FT CORBA basée sur l'architecture schizophrène. Nous commençons par présenter certains tests avant d'évaluer le comportement temporel d'une application tolérante aux fautes supportant plusieurs styles de réplication. Nous nous intéressons également aux performances de la détection et de la notification des défaillances. Les performances de nos mises en oeuvre seront comparées à d'autres implémentations de FT CORBA.

6.2.1 Gestion des groupes de répliques

Cette section présente deux tests de validation montrant le bon fonctionnement de deux fonctionnalités importantes de l'intergiciel. Nous nous intéressons tout d'abord à la construction des IOGR permettant de référencer les groupes de répliques et d'atteindre les différentes répliques au sein d'un groupe. Ensuite nous proposons un scénario pour valider le comportement des différents intercepteurs en cas de défaillances et en cas de fonctionnement normal.

6.2.1.1 Gestion des références sur les groupes d'objets

Pour valider le support des IOGR que nous avons introduit dans PolyORB, nous avons développé un test de non régression utilisant l'ensemble des fonctions que nous avons décrites ci dessus. L'extrait 15 décrit le scénario d'exécution et le résultat obtenu en utilisant les références de deux répliques. Ce test montre que l'interface utilisée pour la gestion et la construction des références sur les groupes d'objets fonctionne correctement.

Extrait de code 15 Validation de la gestion des IOGR

```
==> Begin test IOGR Generation and Manipulation <==
Create an empty IOGR (internals)............................: PASSED
Create an empty IOGR........................................: PASSED
Add a first profile to the IOGR (internals).................: PASSED
Add a first profile to the IOGR.............................: PASSED
Set the first profile as primary (internals)................: PASSED
Set the first profile as primary............................: PASSED
Add a second Profile (internals)............................: PASSED
Add a second Profile........................................: PASSED
Set the second profile as primary (internals)...............: PASSED
Set the second profile as primary...........................: PASSED
Remove the second member and reset the first as primary (int: PASSED
Remove the second member and reset the first as primary.....: PASSED
Change the Object Group Reference version (internals).......: PASSED
Change the Object Group Reference version...................: PASSED
Test IOGR generation from a PolyORB reference...............: PASSED
END TESTS...................................................: PASSED
```

Pour vérifier la possibilité de contacter les groupes d'objets sans passer par le gestionnaire de réplication, nous avons développé un programme simple (merge_iors). Ce programme permet de construire une IOGR à partir d'un ensemble d'IOR. L'interopérabilité des IOGR construites peut

alors vérifiée en demandant à un client d'appeler une méthode distante en utilisant cette IOGR. Nous avons ensuite vérifié que les références construites comme l'exige la norme en utilisant deux analyseurs d'IOR, le parseur de TAO (catior) d'ILU [15]. La validité des IOGR produites par le ReplicationManager découle du bon fonctionnement de l'interface ci dessus.

6.2.1.2 Mise en oeuvre des styles de réplication de FT CORBA

Afin de vérifier le bon comportement des différentes fonctionnalités utilisées pour la mise en oeuvre des styles de réplication, nous proposons ce second test de non régression. Il s'agit d'une application tolérante aux fautes se basant sur un gestionnaire de réplication, un serveur hébergeant plusieurs groupes d'objets, ainsi qu'un client effectuant des appels de méthodes distants. Le serveur prend deux paramètres : le nombre de répliques et le style de réplication. Il construit un groupe d'objet pour chaque style de réplication. Tous les groupes d'objets se composent du même nombre de répliques saisi par l'utilisateur. Chaque groupe d'objets est ensuite enregistré auprès du ReplicationManager. Pour faciliter les analyses à posteriori, nous identifions chaque réplique par un entier permettant de déterminer le groupe auquel elle appartient ainsi que sa position dans le groupe. Un compteur symbolise l'état de chaque réplique, il est incrémenté d'une unité à chaque invocation d'une méthode pouvant être appelée par le client distant.

Le client prend deux paramètres définissant les groupes d'objets auxquels les requêtes devront être envoyées ainsi que le nombre des invocations à effectuer pour chaque style de réplication. Nous définissons deux scénarios d'invocation. Le première se fait sans défaillances, le seconde se fait en simulant des défaillances au niveau des répliques. Pour les styles de réplication passifs nous simulons la défaillance du primaire. Deux entiers définissant les itérations avant lesquelles des défaillances sont simulées sont enfin choisis au hasard. Ces deux entiers sont enregistrés pour permettre, en cas de besoin, une reproduction fidèle du scénario d'exécution.

Pour des raisons de lisibilité, et même si l'application gère le style de réplication sans état, le scénario que nous proposons se limite aux styles de réplication avec état. Nous vérifions le bon fonctionnement des différents composants de tolérance aux fautes en identifiant la réplique ayant effectivement répondu à la requête. Les mécanismes de synchronisation d'états sont testés en faisant la différence entre la valeur du compteur que maintient le groupe d'objets entre deux itérations successives.

Le résultat de l'exécution du scénario est montré dans l'extrait 16. Nous pouvons constater que tous les styles de réplication sont correctement implantés et que le service répliqué fonctionne correctement en conditions normales, avec une ou deux défaillances. Sur ce scénario, les défaillances ont lieu avant la première et la troisième itération. Nous constatons également que que la détection et la notification des défaillances se produisent correctement, que le gestionnaire de réplication re-configure le groupe et que le mécanisme de location forwarding est correctement implémenté dans GIOP et correctement utilisé dans les intercepteurs. La prochaine section présente le comportement temporel des différents styles de réplication.

6.2.2 Comportement temporel des styles de réplication

Pour jauger le comportement temporel de notre implémentation, nous évaluons les temps de réponses de bout en bout d'une application compatible avec FT CORBA. L'application témoin que nous avons retenue pour ces mesures est semblable à celle décrite ci dessus.

Nous contrôlons explicitement la formation et l'appartenance aux différents groupes d'objets (style d'appartenance MEMB_APP_CTRL). Ce contrôle explicite limite l'activité et donc la perte

[15] http ://www2.parc.com/istl/projects/ILU/

Extrait de code 16 Validation de la mise en oeuvre des styles de réplication

```
==> Begin test FT CORBA Replication styles <==
start test for : L_COLD_PASSIVE............................: PASSED
     State consistency tested, no failures.................: PASSED
      Failure simulated at iteration # 1...................: PASSED
      Location Forwarding..................................: PASSED
      Failure simulated at iteration # 3...................: PASSED
      Location Forwarding..................................: PASSED
     State consistency under failures......................: PASSED
start test for : L_WARM_PASSIVE............................: PASSED
     State consistency tested, no failures.................: PASSED
      Failure simulated at iteration # 1...................: PASSED
      Location Forwarding..................................: PASSED
      Failure simulated at iteration # 3...................: PASSED
      Location Forwarding..................................: PASSED
     State consistency under failures......................: PASSED
start test for : L_ACTIVE..................................: PASSED
     State consistency tested, no failures.................: PASSED
      Failure simulated at iteration # 1...................: PASSED
      Location Forwarding..................................: PASSED
      Failure simulated at iteration # 3...................: PASSED
      Location Forwarding..................................: PASSED
     State consistency under failures......................: PASSED
start test for : L_ACTIVE_WITH_VOTING......................: PASSED
     State consistency tested, no failures.................: PASSED
      Failure simulated at iteration # 1...................: PASSED
      Location Forwarding..................................: PASSED
      Failure simulated at iteration # 3...................: PASSED
      Location Forwarding..................................: PASSED
     State consistency under failures......................: PASSED
END TESTS..................................................: PASSED
```

de temps pouvant être occasionnée par l'infrastructure de tolérance aux fautes.

L'application témoin définit une méthode "echostring" prenant et renvoyant une chaîne de type CORBA.String comprenant 4 caractères. L'invocation de cette méthode modifie l'état de chaque réplique en incrémentant un compteur. L'invocation de cette méthode sur un groupe d'objets nécessite donc une synchronisation des états des répliques selon le style de réplication du groupe. La latence des primitives utilisées pour vérifier l'état des répliques n'est pas prise en compte par nos mesures. Le `ReplicationManager` gère simultanément quatre groupes d'objets, chacun de ces groupes permet de tester l'un des styles de réplication avec état de `FT CORBA`. Les mesures de performances présentent les latences de bout en bout pour chaque style de réplication. Le choix de déploiement sur une seule machine s'effectue sans perte de généralité, puisque les appels distants passent par l'`ORB` et par toute la pile protocolaire au niveau des différentes entités de l'application. Les deux paragraphes suivants présentent deux ensembles de mesures. Le premier a pour objectif de comparer les latences des invocations selon le style de réplication. Le second paragraphe présente le comportement des différents intercepteurs lorsque le degré de réplication (c'est à dire le nombre de répliques dans le groupe) évolue.

6.2.2.1 Distribution des latences

La figure 6.1 montre la latence de 1000 invocations distance sur des groupes d'objets de trois répliques ayant chacun un style de réplication différent. Pour chaque style de réplication, le graphique donne deux informations : une indication sur la latence moyenne des invocations ainsi que la distribution des différentes latences autour de cette moyenne.

Le style de réplication active présente les meilleures performances pour cette configuration.

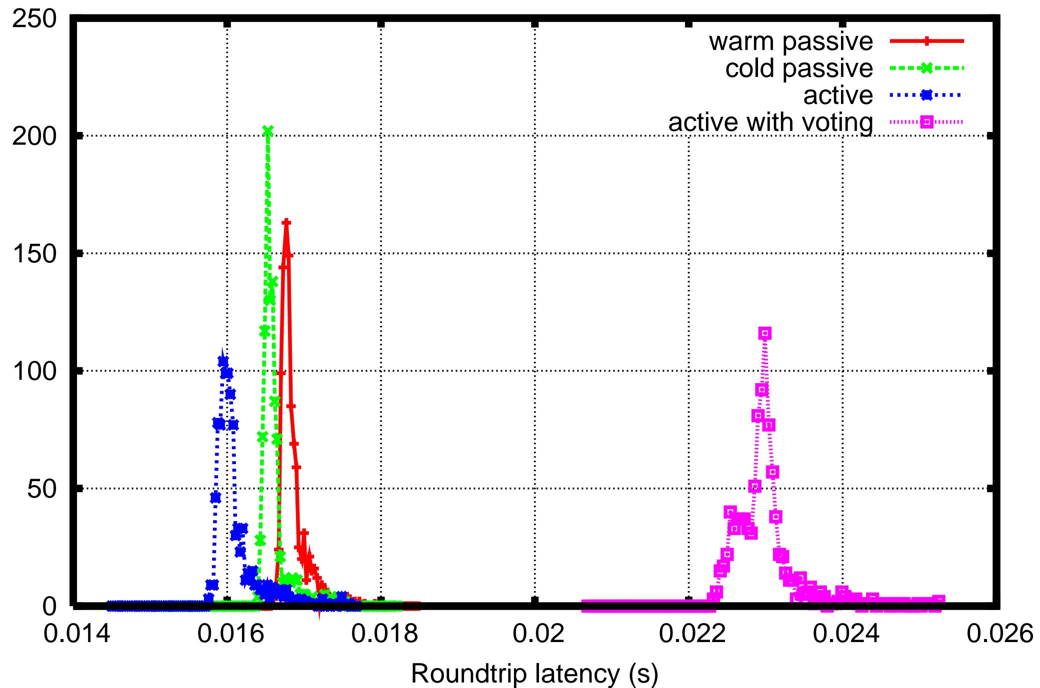

FIG. 6.1 – Distributions des latences de bout en bout pour les styles de réplication avec état

Il est suivi respectivement par les styles COLD_PASSIVE, WARM_PASSIVE et ACTIVE_WITH_VOTING. Le style de réplication active avec vote, comme prévu, présente les latences les plus importantes avec une moyenne de 23ms. Ceci est dû au traitement additionnel fait au niveau des intercepteurs pour organiser le vote. Notons que pour le style de réplication active, l'intercepteur client attend toutes les réponses à ses requêtes temporaires. Plusieurs implémentations préfèrent renvoyer la réponse la plus rapide. Pour nos mesures nous avons fait en sorte que l'intercepteur renvoie la dernière, ce choix nous permet de déterminer le coût introduit par l'organisation du vote. Le temps additionnel introduit par le vote peut être expliqué en grande partie par la manipulation du type CORBA.Any qui prend beaucoup de temps. Notre proposition admet un comportement temporel équivalent ou meilleur que plusieurs implantations de FT CORBA proposées par [8] et par [88].

6.2.2.2 Les latences en fonction du degré de réplication

La figure 6.2 présente des latences moyennes sur 30 invocations. L'application que nous testons est la même que celle utilisée dans les mesures précédentes (figure 6.1). Pour ces mesures, nous faisons évoluer le degré de réplication. Les résultats montrent des latences de plus en plus importantes au fur et à mesure que le degré de réplication augmente. Ce phénomène peut s'expliquer par l'augmentation du nombre de messages échangés. Nous remarquons que cette évolution est plus importante pour les styles de réplication actifs. Ceci est du aux délais supplémentaires d'attente et de collection des réponses ainsi qu'a l'organisation des votes (pour le style ACTIVE_WITH_VOTING).

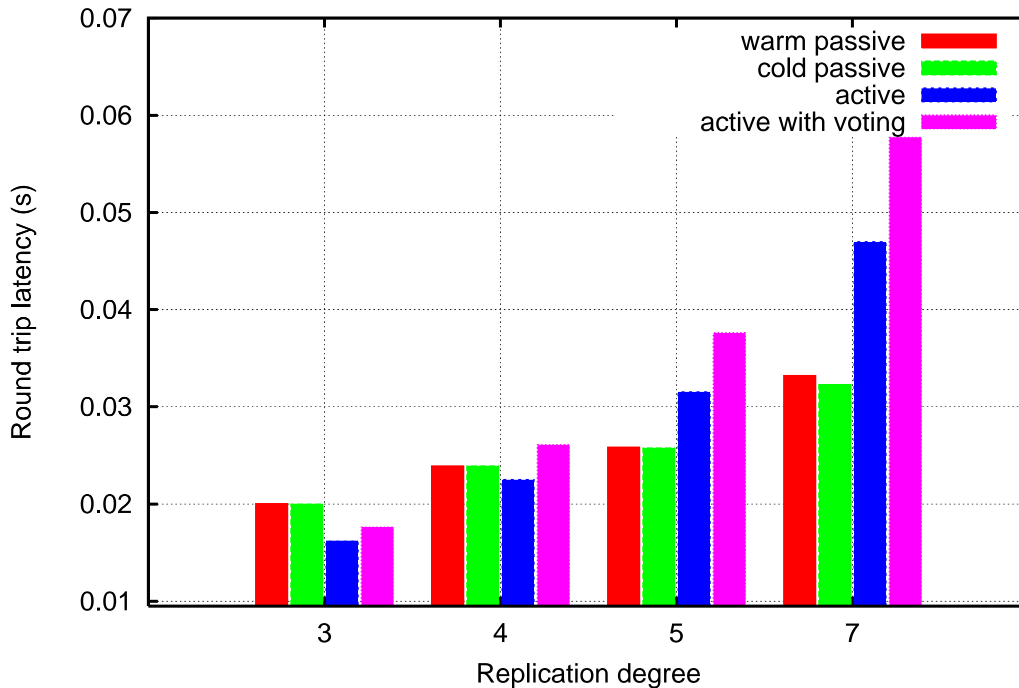

Fig. 6.2 – Les latences en fonction du degré de réplication

6.2.3 Détection et notification des défaillances

Pour évaluer le comportement temporel du détecteur de défaillance que nous avons mis en place, nous avons utilisé la boucle locale de l'`ORB` pour assurer les échanges de requêtes. Ce choix a au moins deux avantages. D'une part, il facilite le déploiement. D'autre part, il empêche les distorsions des mesures potentiellement occasionnées par le comportement du réseau. Nous pouvons alors nous concentrer exclusivement sur les performances de la détection et la notification des défaillances. Selon le standard, la détection et la notification des défaillances font intervenir trois processus. Le détecteur de défaillances, le notificateur des défaillances et le gestionnaire de réplication.

Le détecteur des défaillances est en charge de la surveillance des répliques, et plus généralement de tout objet qui implémente l'interface PullMonitorale. Le notificateur des défaillances (`FaultNotifier`) se charge de la réception et de l'acheminement des rapports de défaillances jusqu'au gestionnaire de réplication qui prend les mesures nécessaires en fonction des propriétés des groupes des répliques défaillantes. Le temps que nous mesurons est celui qui sépare l'occurrence de la défaillance de l'arrivée du rapport au `ReplicationManager`.

L'extrait de code 17 présente l'interface des objets à surveiller pour ce scénario de test. Ces objets implémentent la méthode obligatoire *is_ alive* mais également trois autres primitives pouvant être invoquées pour simuler des défaillances. shutdown fait en sorte que *is_ alive* renvoie FALSE. delay_is_alive permet de retarder les réponses aux invocations d'une durée suffisamment importante pour dépasser les timeouts. La troisième méthode fait en sorte que les appels à *is_ alive* génèrent une exception.

Après la création d'un `FaultDetector`, un `FaultNotifier` et d'un `ReplicationManager`, les différentes connexions entre ces trois éléments sont établies selon la norme. Nous nous basons

Extrait de code 17 Réplique héritant de PullMonitorable

```
import ::FT;

interface Echo: ::FT::PullMonitorable
{
    string echoString (in string Mesg);

    void shutdown ();

    void delay_is_alive ();

    void crash ();
};
```

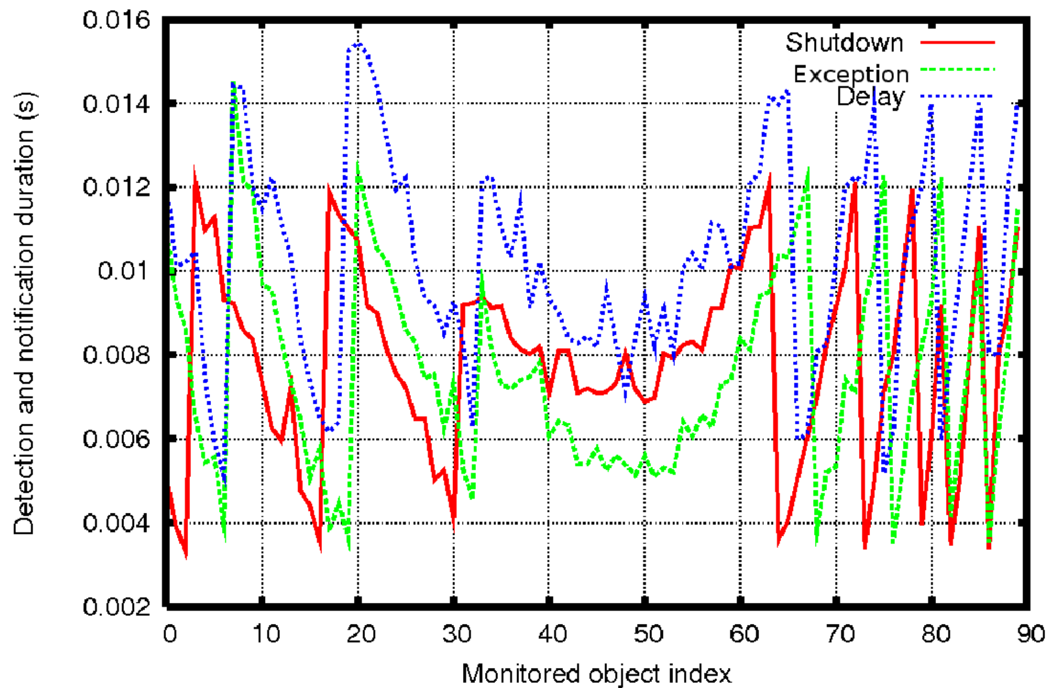

Fig. 6.3 – Comportement temporel de la détection et de la notification des défaillances

sur le service de notification pour la propagation des données entre ces trois composants. Un ensemble de d'objets `CORBA` héritant de l'interface ci-dessus sont également créés. Le détecteur des défaillances commence la surveillance simultanée des objets précédemment créés. Selon le scénario, après une certaine durée, une des trois méthodes décrites ci-dessus est invoquée pour simuler une défaillance. L'invocation de cette méthode est immédiatement signalée au `ReplicationManager` comme date de défaillance. Si l'infrastructure de tolérance fonctionne correctement alors cette défaillance est détectée et ensuite propagée jusqu'au `ReplicationManager` sous forme d'un rapport d'erreurs créé par le détecteur et relayé par le notificateur. Le `ReplicationManager` enregistre la date de l'arrivée du rapport. Il est alors possible de mesurer le temps de détection de la défaillance de la réplique en faisant la différence entre les deux dates. Ce temps est équivalent à la somme du temps de détection et du temps de propagation du rapport d'erreur, les temps de création et de gestion des rapports étant négligeables.

La figure 6.3 montre le temps de détection et de propagation des défaillances de 90 objets surveillés avec un intervalle de surveillance (monitoring interval) de 10 milli-secondes et un timeout de 3 milli-secondes. Les 90 défaillances sont détectées dans tous les cas, quelque soit la nature de la défaillance. La propriété de complétude est satisfaite par ce détecteur pour ce test. Les temps de détection sont, naturellement, plus grands que le timeout, ce qui prouve l'absence de faux positifs ce qui assure la propriété de complétude du détecteur. Notons que le détecteur réagit de la même manière aux trois types de défaillances que nous proposons pour ce test, ce qui prouve une certaine robustesse pour notre détecteur de défaillances. L'infrastructure de détection et de notification de défaillances que nous proposons exhibe des performances totalement acceptables en comparaison avec [130] et à [78].

6.2.4 Conclusion

Dans cette section, nous avons proposé un ensemble de tests et de mesures qui nous ont permis de vérifier la qualité de l'intégration de d'un service de tolérance aux fautes dans l'intergiciel schizophrène `PolyORB`. Au niveau des performances, notre mise en oeuvre des différents styles de réplication présente des performances égales ou meilleures que les implantations de cette norme proposées par [8] et par [88]. Pour l'infrastructure de détection et de notification de défaillances, notre proposition exhibe des performances totalement acceptables en comparaison avec [130] et à [78]. L'intégration de `FT CORBA` dans l'architecture schizophrène a été donc réussie tant au niveau architectural, qu'au niveau du comportement temporel. Notons que ce comportement temporel, même s'il est déjà satisfaisant, peut être sujet à un ensemble d'améliorations puisque aucune piste d'optimisation n'a été explorée. Notons que l'architecture de ce service nous a permis de développer une application témoin mettant en oeuvre simultanément tous les styles de réplication. Le nombre de répliques simultanément surveillées par le détecteur des défaillances est également important, il montre des capacités de passage à l'échelle ainsi qu'une réaction uniforme à plusieurs types de défaillances. Les différents scénarios de tests montrent que l'intégration de la tolérance aux fautes dans l'architecture schizophrène s'est effectuée avec succès tant sur le plan architectural que sur les plans performance et passage à l'échelle.

6.3 Évaluation du service générique de consensus

Pour évaluer l'architecture et la mise en oeuvre de notre service du consensus nous avons effectué deux types d'évaluations complémentaires. D'une part, nous avons effectué plusieurs mesures de performances en variant plusieurs paramètres : algorithmes, protocoles de transport de données et plates-formes de déploiement. D'autre part, nous avons étudié la distribution du code entre les différents éléments architecturaux de notre conception pour évaluer la validité et l'extensibilité de notre proposition.

6.3.1 Analyse du code source

L'analyse du code source fournit non seulement des informations sur les éléments importants d'une architecture mais aussi des indications précieuses sur l'effort d'implémentation nécessaire pour supporter de nouvelles fonctionnalités.

Dans notre architecture, les composants pouvant être facilement être réutilisés sont naturellement les composants que nous avons repris de `PolyORB` mais aussi les différentes abstractions de haut niveau pour le transport de messages, le consensus et la détection de défaillances.

Nous analysons une instance de notre architecture comprenant deux algorithmes de consensus : le coordinateur tournant de Chandra ainsi qu'un autre algorithme proposé dans [85]. Nous nous basons également sur les deux détecteurs de défaillances de classe $\diamond S$ ([22]) et $\diamond P$ ([75]). Ces algorithmes sont assez connus dans la littérature. En pratique, le choix des algorithmes dépend des applications et des environnements de déploiement. En ce qui nous concerne, le nombre des algorithmes importe peu dès lors que nous montrons que l'architecture que nous proposons ne va pas influencer le comportement des algorithmes supportés et qu'elle pourra facilement supporter de nouvels algorithmes.
L'analyse de cette instance est effectuée grâce au logiciel SLOCCount[16] grâce auquel nous mesurons le nombre de ligne de code non commentées de notre implémentation. Les différentes mesures sont fournies par la table 6.1.

Cette table montre que la majorité ($\approx 89\%$) du code utilisé dans cette architecture provient de `PolyORB`, du composant du transport (transport et représentation des messages) et d'autres fonctionnalités utilitaires (gestion des listes de participants, etc). La mise en oeuvre des composants de consensus et de détection de défaillances n'a pas nécessité un effort de codage important. La partie du code désignée comme "générique" pour chaque composant correspond au code fournissant l'interface de chaque composant. En dessous de cette donnée nous trouvons la taille du code nécessité pour implémenter chaque algorithme.

Cette analyse montre clairement que la majorité du code nécessité pour concrétiser notre architecture est générique et/ou très fortement réutilisable. L'effort de codage des algorithmes est réduit au strict minimum. Par conséquent, notre architecture supporte facilement l'ajout de nouveaux algorithmes. Le code nécessaire pour mettre en oeuvre ces algorithmes sera simple, facilement traçable et en cas de besoin facilement certifiable.

6.3.2 Mesures de performances

Généralement, l'évaluation des performances d'un algorithme de consensus ou de détection de défaillances se fait théoriquement en analysant leurs différentes complexités. Le coût des échanges

[16] `http://www.dwheeler.com/sloccount/`

[17] La taille de la couche neutre de `PolyORB` est d'à peu près 40000 SLOCs, nous en utilisons effectivement 5000

PolyORB	Consensus		Failure detection		Transport	Other utilities
SLOCs	Algo	SLOCs	Algo	SLOCs	SLOCs	SLOCs
≈ 5000[17]	generic	144	generic	333	833	2200
	Chandra[22]	273	Chandra $\diamond S$ [22]	170		
	Mostefaoui[85]	273	Larrea $\diamond P$ [75]	279		

TAB. 6.1 – SLOCs des modules du service du consensus

des données est traditionnellement approché par le nombre total des messages nécessités. Le coût des calculs est également approché par le nombre d'étapes nécessaires pour atteindre le consensus.

Même si ces différents paramètres fournissent des indications de valeur, ils ne sont pas suffisamment précis dans le cadre d'applications présentant des exigences fortes en performances et en déterminisme. Implémenter et tester un algorithme de consensus au dessus d'un système distribué est généralement difficile à cause des efforts requis pour mettre en oeuvre les primitives d'échange de données de contrôle de concurrence. Dans notre cas, ces efforts ont été minimisés grâce à l'architecture claire de notre service ainsi qu'à la réutilisation de plusieurs composants préexistants fournis par `PolyORB`.

6.3.2.1 Configuration de base

Nous avons choisi l'algorithme du coordinateur tournant pour cette évaluation. Cet algorithme a été proposé dans [22]. Cet algorithme nécessite un détecteur de défaillances inévitablement fort (classe $\diamond S$). L'application de test consiste en un nombre variable de participants chacun exécutant cet algorithme ainsi qu'un algorithme de détection de défaillances compatible. Après une phase d'initialisation pendant laquelle les partitions sont créées et les algorithmes de consensus et de détection de défaillances chargés, les processus lancent 100 instances de consensus en proposant à chaque fois un entier tiré au sort. Nous lançons les différentes partitions au sein d'un seul programme `Ada`. Ce choix permet de simplifier le protocole de mesures en évitant par exemple le problème de synchronisation des horloges. Il permet également de tester l'isolation entre les partitions. Par exemple, les instances de consensus et de détection de défaillances sur une partition ne doivent pas présenter de mauvais comportements par exemple en récupérant des données ou des ressources de calcul appartenant à une autre partition.

Nous présentons ci dessous deux ensembles de mesures utilisant l'algorithme du coordinateur tournant de Chandra avec deux détecteurs de défaillances différents. Ces deux tests utilisent `UDP` comme protocole de transport de messages. Le composant du consensus implémente l'algorithme du coordinateur tournant alors que le composant de détection de défaillances fournit deux algorithmes de détection de défaillances de classes $\diamond S$ et $\diamond P$. Ces deux algorithmes sont respectivement proposés par Chandra dans [22] et par Larrea dans [75].

La figure 6.4 présente les résultats d'une première expérience utilisant un détecteur de défaillances de classe $\diamond S$. Nous définissons trois participants au début du test. Une défaillance est simulée au niveau de la partition du coordinateur en arrêtant le détecteur des défaillances et le gestionnaire des messages. Les trois processus présentent une durée de convergence d'à peu près $1ms$. Nous notons également une bonne distribution des mesures autour de la moyenne. Un pic est enregistré pour le 50 tour, il est dû au fait que les processus ne peuvent décider de la défaillance du coordinateur jusqu'à la réception de cette information depuis chaque détecteur local (utilisant un `timeout` de $100ms$). Le pic mesuré est d'à peu près $208ms$. Après le cinquantième tour, nous enregistrons une légère augmentation des temps de convergence. Cette augmentation s'explique par deux phénomènes qui se compensent : le temps écoulé pour définir un nouveau

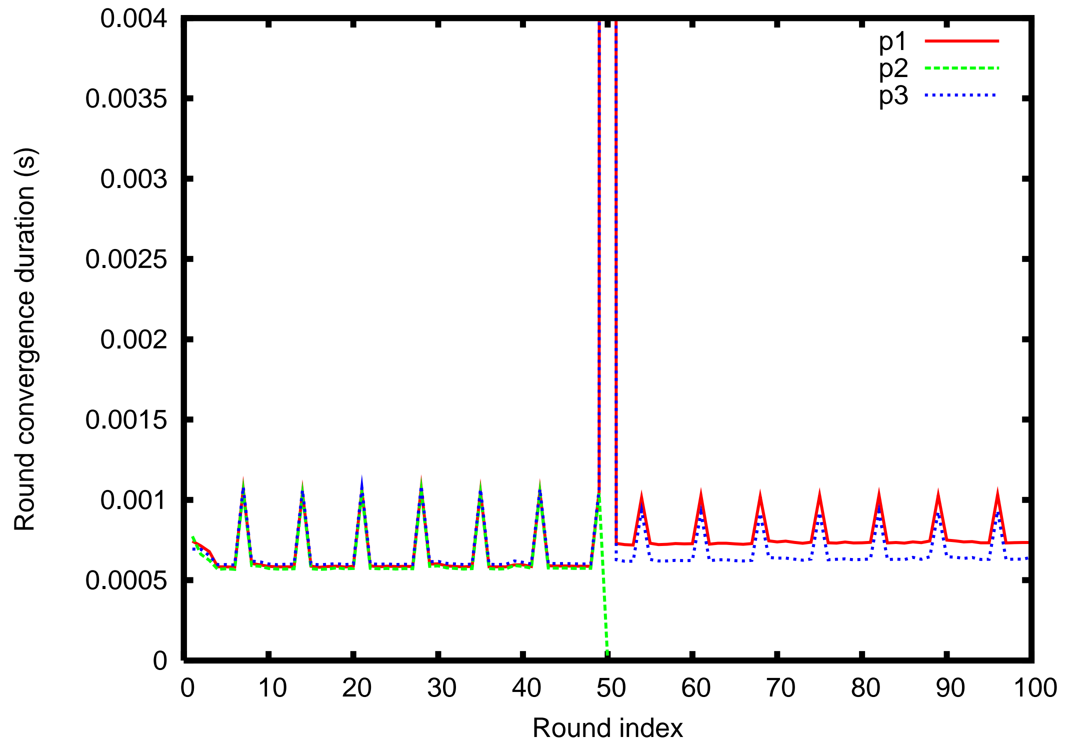

FIG. 6.4 – Coordinateur tournant avec un détecteur de défaillances de classe $\diamond S$

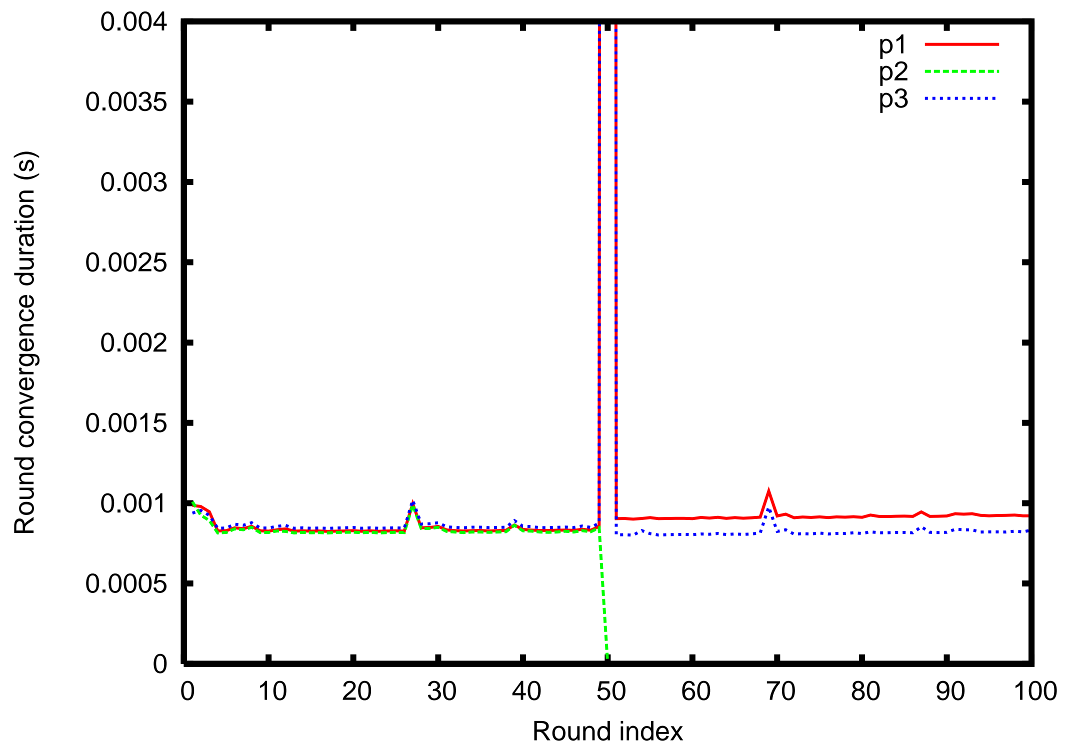

FIG. 6.5 – Coordinateur tournant avec un détecteur de défaillances de classe $\diamond P$

coordinateur et la libération de ressources additionnelles après la simulation du crash dans la partition du coordinateur.

La figure 6.5 présente les résultats d'une seconde expérience similaire à la première mais cette fois en utilisant un second détecteur de défaillances (classe $\diamond P$). Notons que le code de l'application test est le même et qu'entre les tests nous n'avons changé qu'un seul paramètre dans la ligne de commande. Le timeout pour le détecteur de défaillances n'a pas également pas été modifié. Le comportement temporel est similaire même si nous notons que la moyenne des temps d'invocations a légèrement augmenté. Cette augmentation s'explique par les traitements additionnels effectués au niveau du second détecteur de défaillances ainsi que par la taille des messages échangés (certains messages transportent en effet des listes entières de suspects). Notons enfin que les timeouts peuvent être réduits mais que cette réduction sera au profit de la disponibilité des ressources de calcul et de transport de données et peuvent mettre en péril certaines propriétés du détecteur de défaillances.

Les deux expériences ci dessus montrent un comportement temporel stable et correspond au comportement théorique des différents algorithmes utilisés. Ces tests nous permettent d'avoir un minimum de confiance en l'architecture et la mise en oeuvre des différents composants du service du consensus.

6.3.2.2 Compatibilité avec l'architecture schizophrène

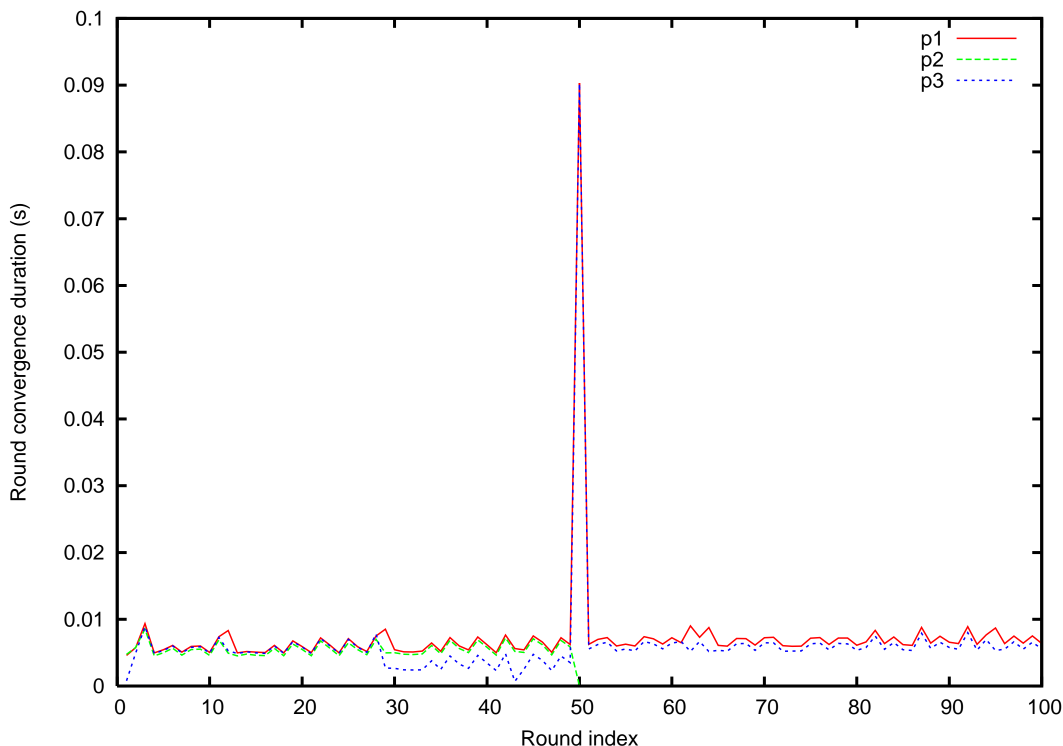

FIG. 6.6 – Comportement temporel du protocole de consensus en utilisant `IIOP`

L'architecture de notre service du consensus ainsi que le travail d'intégration au sein de l'architecture schizophrène ont permis de facilement mettre en oeuvre le test ci dessous. Nous utilisons les algorithmes du coordinateur tournant ainsi que le détecteur de défaillances de classe $\diamond S$ du précédent test.

Pour ce test nous proposons trois participants se basant sur la personnalité CORBA de PolyORB. Un serveur CORBA est utilisé pour la phase d'initialisation. Chaque processus invoque une méthode distante au niveau de ce serveur pour s'enregistrer et pour obtenir les informations nécessaires concernant la configuration du groupe. A la fin de cette phase d'initialisation, le serveur est automatiquement arrêté. Les participants peuvent maintenant participer aux différentes instances de consensus. Les échanges de messages se font maintenant grâce au protocole IIOP.

Chaque participant lance 100 instances de consensus, pour chacune de ces instances, il propose une valeur tirée au sort et enregistre cette valeur, la valeur résultante du consensus ainsi que la durée écoulée pour obtenir le consensus. Comme pour le précédent test, une défaillance est simulée au niveau du coordinateur au cinquantième tour. Les différentes données enregistrées par chacun des participants permet, d'une part la vérification ultérieure des propriétés de terminaison, d'accord et de terminaison et, d'autre part d'accomplir les mesures de performances.

Les résultats de cette expérience sont présentés par la figure 6.6. Ils indiquent une durée moyenne de $10ms$ ainsi qu'un pic de $90ms$ au niveau du cinquantième tour correspondant à la défaillance du coordinateur. Ces résultats montrent un comportement similaire à celui de l'expérience précédente même si nous enregistrons une durée de convergence moyenne et une distribution autour de la moyenne moins bonnes qu'avec UDP. Les résultats de ce test sont totalement attendus connaissant le comportement temporel d'IIOP. Bien entendu, ce test ne sert pas à montrer le comportement temporel de notre service du consensus mais plutôt à prouver que son architecture lui permet de supporter facilement plusieurs protocoles.

6.3.2.3 Configuration déterministe

Pour illustrer le support d'environnements de déploiement exigeants (c'est à dire de type embarqué et/ou temps réel), nous avons développés des tests pour les exécutifs temps réel ORK[18] et MaRTE OS[19]. La conception et le développement de l'application test se sont faites selon la méthodologie suivante :

1. Choix et configuration des services intergiciels de base
2. Instantiation des composants génériques du service du consensus
3. Compilation de l'application en utilisant l'exécutif approprié
4. Lancement des applications et analyses des résultats

Le choix et paramétrage des services intergiciels de base ont été discutés lors de la description de la méthodologie d'assemblage des différents composants de ce service (paragraphe 5.4.3.3). Pour nos tests nous avons utilisé un assemblage similaire a celui de l'application embarquée présentée en (5.4.3.4).

Comme l'exécutif ORK ne supporte pas les entrées/sorties, nous avons développés un protocole simple basé sur la notion de boites aux lettres et se basant sur le service de concurrence de PolyORB. Ce protocole fournit une instanciation de la classe Transport présentée lors de la conception du service de consensus (chapitre 4). L'initialisation de ce service se fait grâce à une configuration statique associant à chaque participant une adresse lui permettant de recevoir les données qui lui sont envoyées par les autres participants.

La production du code de l'application dépend de l'architecture du de déploiement et de l'exécutif à utiliser. Le support de l'exécutif ORK par notre application s'est basé sur un ensemble de travaux que nous avons assurés durant cette thèse. En effet, nous avons au préalable effectué

[18] http://www.dit.upm.es/
[19] http://marte.unican.es

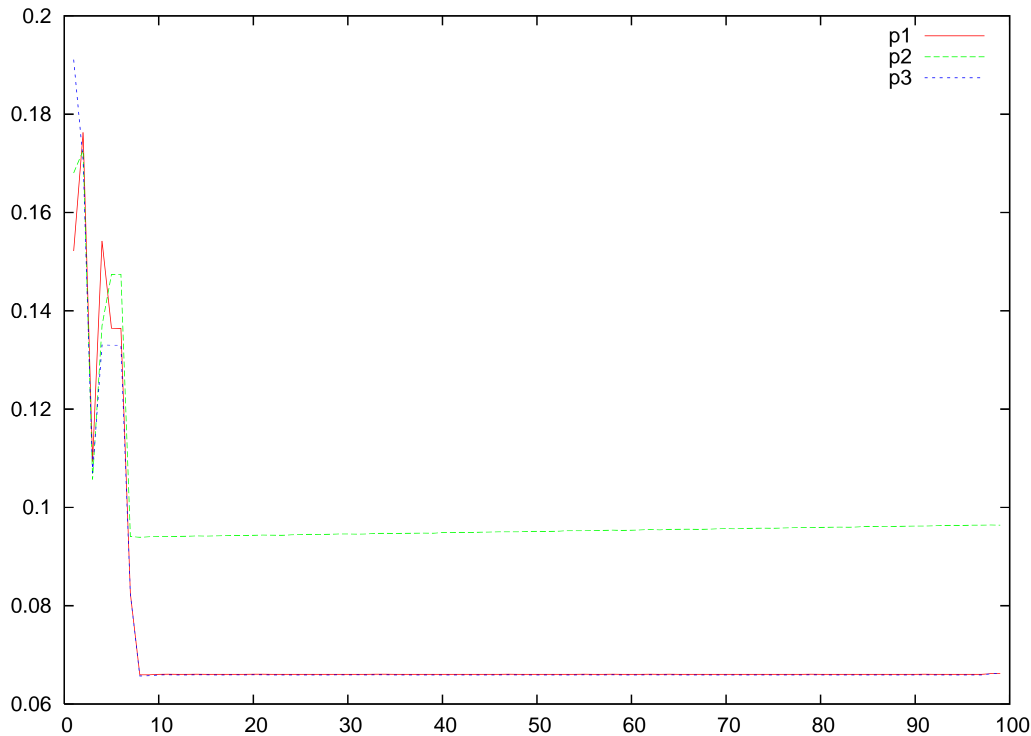

FIG. 6.7 – Comportement temporel d'un protocole de consensus se basant sur l'exécutif ORK

plusieurs tests et produit des patches assurant non seulement la compatibilité du code de PolyORB avec le profil Ravenscar, mais également le bon fonctionnement d'applications basées sur PolyORB et utilisant l'exécutif ORK. En particulier nous nous sommes assurées du bon fonctionnement d'applications basées sur CORBA et GIOP sur ORK.

Le support de MaRTE OS comme exécutif s'est inspiré des travaux que nous avons effectué pour le support de ORK. Les majeures différences se situent au niveau de la chaîne de compilation et de l'utilisation de primitives spécifiques pour l'écriture sur la ligne série. Même si ce n'est pas le premier objectif, le support de MaRTE OS est intéressant pour tester le comportement de l'application lorsque le profil Full_Tasking est choisi. En effet ORK est conçu pour supporter exclusivement les applications compatibles avec le profil Ravenscar.

Pour tester les différentes applications produites, nous nous sommes principalement basés sur le simulateur Bochs[20]. Le code généré à la sortie des différentes chaînes de compilations définies par les deux exécutifs fonctionne directement sur les architectures x86 comme par exemple un ordinateur de bureau. L'utilisation de Bochs nous a permis de gagner du temps et d'éviter la mobilisation de matériel additionnel.

Les figures 6.7 et 6.8 montrent le comportement temporel de l'algorithme de consensus de Chandra dans le cas sans défaillances en utilisant respectivement les exécutifs ORK et MaRTE OS. Ces mesures sont obtenues grâce au simulateur Bochs. Le comportement temporel des deux applications est similaire, cependant les performances sont meilleures dans le cas de ORK. Cela peut être expliqué par la légèreté de cet exécutif. Notons que Bochs simule une machine disposant d'un processeur effectuant 10000000 instructions par seconde et disposant de 512 Méga octets de RAM.

[20]http://bochs.sourceforge.net

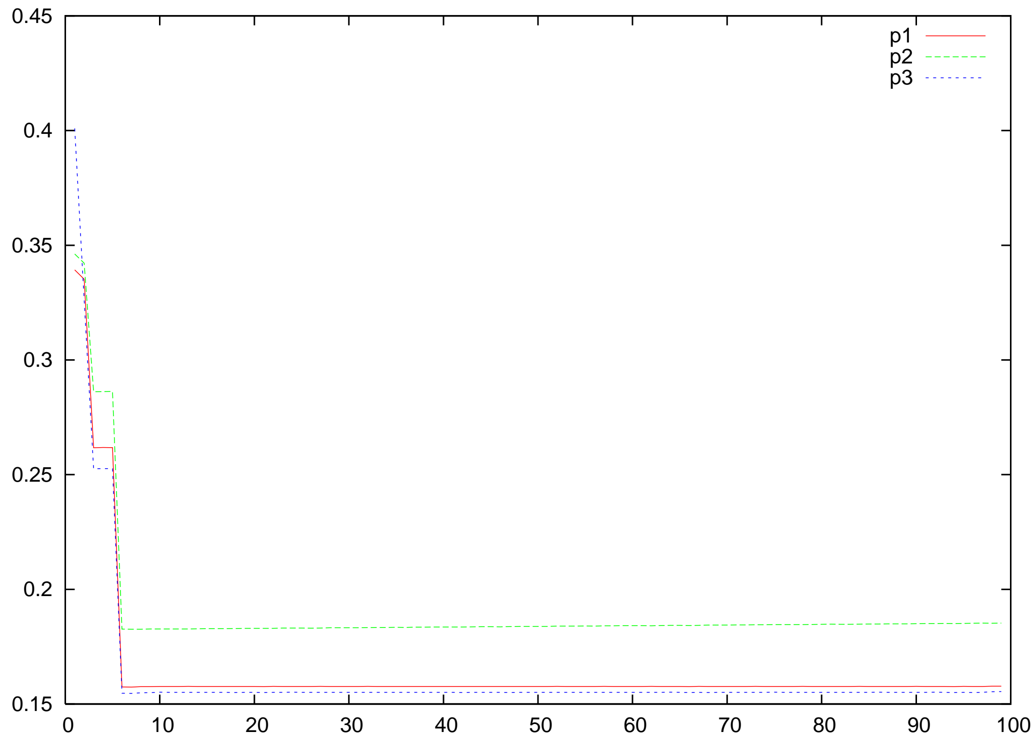

Fig. 6.8 – Comportement temporel d'un protocole de consensus se basant sur l'exécutif `MaRTE OS`

Les différentes mesures que nous avons effectué montrent un bon comportement temporel du service de consensus et des différents services intergiciels de base que nous avons utilisés (notamment, le service de concurrence compatible avec `Ravenscar` et le protocole de communication asynchrone). Une mesure séparée du comportement temporel de ces deux services permettra d'évaluer l'impact de ces différents services sur les performances globales de l'application. Hormis les premiers rounds de consensus, le comportement temporel des deux applications est suffisamment stable et présente une dispersion quasi nulle ce qui permet de garantir des propriétés et facilite les analyses d'ordonnancement.

6.3.3 Conclusion

Pour évaluer l'architecture et la mise en oeuvre du service du consensus, nous avons proposé deux types de mesures. La première consiste à analyser la répartition des lignes de codes nécessaires pour la réalisation des différents composants de ce service. Cette analyse a montré que la majorité du code a été introduite dans le cadre d'une réutilisation à partir d'un intergiciel préexistant ou dans le cadre de la mise en oeuvre de composants génériques. L'effort de codage nécessaire pour la mise en oeuvre de nouveaux algorithmes ou pour supporter de nouveaux protocoles de transport de données et minime.

Nous avons ensuite procédé à plusieurs mesures de performances permettant l'évaluation du comportement temporel de trois configurations correspondant à trois cas d'utilisation différents. Pour chacune de ces configurations nous avons utilisé un protocole différent pour assurer les échanges de données. Ces mesures montrent une très bonne distribution du temps de réponses autour de la moyenne. Ces trois configuration différentes montrent la capacité de notre composant

à supporter divers besoins provenant d'applications appartenant à plusieurs domaines. Nous supportons en particulier les besoins en déterminisme des applications les plus critiques. Le service de consensus s'intègre parfaitement dans l'architecture schizophrène ce qui lui assure le support et l'interopérabilité avec différents modèles de distributions et en particulier le standard `CORBA`. Le comportement général des différentes applications est similaire, il montre un comportement stable de l'architecture et notamment une très grande efficacité de l'infrastructure de gestion des évènements. La conception et la mise en oeuvre de scénarios de tests compatibles avec des exigences strictes tel que le support de `Ravenscar` et l'utilisation d'exécutifs dédiés montrent, outre le comportement temporel stable, une bonne portabilité de notre service. La facilité de configuration et du déploiement des applications dans les cas les plus exigeants prouvent la justesse des choix de conception de notre service de consensus.

6.4 Conclusion

Dans ce chapitre, nous avons proposé un ensemble de tests de validation et de mesures de performances évaluant le comportement et les différentes propriétés temporelles et architecturales des composants que nous avons conçus dans le cadre de cette thèse.

Nous avons étudié à travers plusieurs cas d'utilisation et mesures de performances le comportement des intercepteurs, des détecteurs de défaillances ainsi que celui de l'infrastructure de tolérance aux fautes. Ces mesures valident les différents choix que nous avons adoptés pour ajouter le support de la tolérance aux fautes à l'architecture schizophrène.

Les propriétés du service du consensus ont également été mises en évidence. Nous avons procédé à une analyse du code de notre mise en oeuvre. Cette analyse montre une grand pourcentage de la partie générique de l'architecture et donne une indication sur le coût du support de nouveaux algorithmes est relativement faible. Les différents cas d'utilisation montre la configurabilité de ce service et les différentes mesures de performances montrent sa capacité à supporter des contraintes temporelles fortes venant d'applications exigeantes en qualité et même critiques.

Le principe de séparation des préoccupations, les études architecturales, la définition et l'utilisation des abstractions permettent donc de réconcilier les besoins de configurabilité et d'adaptation aux exigences de qualité comme la sûreté de fonctionnement et le comportement temporel stable.

Chapitre 7

Conclusions et perspectives

La réconciliation des besoins de limitation des coûts et des exigences de qualité présentés par des applications à forts besoins en sûreté de fonctionnement a été notre principal objectif dans cette thèse. Les technologies intergicielles contribuent à la limitation des coûts de conception et développement d'applications appartenant à plusieurs domaines. Nous avons étudié et proposé deux services intergiciels généralement indispensables dans le cadre du développement des applications à forts besoins en sûreté de fonctionnement : la tolérance aux fautes et le consensus. Dans ce chapitre, nous revenons sur nos contributions et présentons les perspectives de cette thèse.

7.1 Contributions

L'architecture schizophrène fournit une réponse efficace à deux problèmes essentiels : la réduction des coûts et le support de certaines exigences de qualité comme la compatibilité avec le profil de restrictions `Ravenscar` et le support de la vérification formelle. Ces différentes exigences sont satisfaites simultanément.

Il fallait renforcer cette architecture avec des services de tolérance aux fautes et de consensus, requis par plusieurs applications tolérantes aux fautes et/ou critiques. Nous rappelons ici nos principales réalisations et contributions. Conscients de l'important rôle de l'architecture de l'intergiciel pour la réconciliation des besoins et la résolution des compromis posés par les applications, nous nous sommes intéressés aux abstractions et services intergiciels permettant de supporter la tolérance aux fautes et le consensus. Nous avons systématiquement appliqué le principe de séparation des préoccupations et nous nous sommes basé sur plusieurs patrons de conception lors de nos propositions d'architecture. Les principaux résultats de cette thèse sont :

Intégration de `FT CORBA` dans l'architecture schizophrène

Nous avons proposé une architecture permettant de renforcer l'architecture schizophrène avec un service de tolérance aux fautes compatible avec `FT CORBA`. Contrairement à plusieurs architectures intergicielles tolérantes aux fautes, l'architecture que nous proposons réconcilie les aspects de flexibilité, de performances et de configurabilité. Ce service présente les avantages et les caractéristiques suivants :

- *Préservation des propriétés de l'architecture schizophrène.* Pour supporter les différents styles de réplication, nous avons proposé une solution transparente et non intrusive basée sur des intercepteurs proches des intercepteurs portables de `CORBA` mais évitant plusieurs problèmes posés par ces derniers.

- *Transparence et configurabilité de la tolérance aux fautes.* Nos intercepteurs sont appliqués d'une manière transparente à l'ORB et à l'application. La définition d'une classe d'intercepteurs par style de réplication ainsi que l'architecture modulaire que nous avons mis en place permettent une grande flexibilité lors du choix du style de réplication. Notre architecture facilite le passage d'une application "classique" à une application tolérante aux fautes. Au niveau du client, il suffit de charger l'intercepteur responsable du style de réplication choisi. Côté serveur, il faut un peu plus de travail pour mettre en place le groupe d'objets et concrétiser les interfaces obligatoires de la norme nécessaires à la détection des défaillances et à la gestion des états. Les propriétés de tolérance aux fautes sont facilement ajustables grâce à la définition de fichiers et de paquetages de configuration.
- *Compatibilité avec le standard.* La mise en oeuvre que avons proposé présente la plus grande majorité des fonctions définies par le standard. Les contrats IDL de ce service n'ont pas subi de modifications. Nous avons également vérifié la compatibilité des IOGR avec la norme. La propagation des informations sur les défaillances des processus s'est également faite selon les directives de la norme.
- *Flexibilité de l'architecture résultante.* L'introduction transparente de la tolérance aux fautes augmente les possibilités de configuration des applications. Toutes les propriétés définies par le standard ont été supportées sans compromettre les avantages de l'architecture schizophrène. Par exemple, l'utilisateur a toujours la liberté de choisir un modèle de concurrence restreint à Ravenscar ou non. Les exigences de cohérence lors des choix des configuration sont également plus nombreux. Par exemple le détecteur de défaillances ne peut pas être instancié en se basant sur une unique tâche (profil de concurrence No_Tasking).
- *Détection et notification des défaillances.* Nous avons proposé une détection de défaillances se basant uniquement sur la couche neutre de l'architecture schizophrène et compatible avec les restrictions Ravenscar. Le grand nombre de répliques pouvant être simultanément surveillés et le comportent équivalent en réaction à plusieurs types de défaillances, et le support de plusieurs modèles de concurrence ont été les principaux résultats pour ces composants.
- *Isolation du comportement.* Les intercepteurs que nous avons proposé isolent tout les aspects comportementaux liés aux traitements des requêtes, les techniques utilisées pour la vérification du μBroker peuvent être également appliquées pour valider le comportement des implantations des différents styles de réplication.
- *Comportement temporel.* Les différents mesures de performances que avons effectué montrent que même si notre implantation a besoin d'une phase d'optimisation elle exhibe des performances comparables ou meilleurs à d'autres implémentations de ce standard.

Service générique du consensus

Le second axe de recherche de cette thèse a consisté à concevoir et à mettre en oeuvre un service de consensus dont nous avons maximisé la généricité et les possibilités de configuration et optimisé le comportement temporel. Ci dessous nos principales contributions.

- *Architecture modulaire.* Après avoir argumenté le choix des modèles de calcul et de défaillances nous avons proposé une architecture modulaire se basant sur trois composants pour le consensus, la détection des défaillances et la gestion des messages. Nous nous sommes basés sur plusieurs patrons de conception pour définir les différents éléments de cette architecture. Pour optimiser le comportement temporel de ce service, nous avons accordé une importance particulière à l'efficacité des interactions. Le choix d'une architecture orientée évènements a permis, outre l'optimisation de ces interactions, d'assurer un

couplage faible entre les composants augmentant la flexibilité de l'architecture.

- *Rôle de l'intergiciel lors du support des algorithmes.* Nous nous sommes intéressés au rôle de l'intergiciel comme support à l'exécution des algorithmes de consensus et de détection de défaillances. La définition des abstractions requises par le service de consensus augmente en effet sa généricité puisque il suffit de fournir une mise en oeuvre de ces abstractions pour pouvoir l'utiliser. Les choix et le paramétrage des mises en ouvre de ces différentes abstractions doit par contre se faire en fonction des exigences non fonctionnelles de l'application finale.
- *Intégration dans l'architecture schizophrène.* L'intégration de ce service dans l'architecture schizophrène s'est faite d'une manière efficace grâce à l'utilisation des services canoniques de l'architecture schizophrène et du μ`Broker` pour satisfaire les différents besoins du service du consensus. L'utilisation des services de l'architecture schizophrène n'est pas nécessaire au fonctionnement de ce service, ce dernier peut, en effet, être directement utilisé par les applications. Nous pensons que se service peut être facilement intégré dans d'autres architectures intergicielles.
- *Dimensions de configuration et processus d'assemblage.* L'intégration dans l'architecture schizophrène augmente les possibilités de configuration de ce service, principalement au niveau du transport des données. Les dimensions de configuration de ce service on été décrites et un processus de configuration et d'assemblage prenant en compte les différents besoins d'applications susceptibles d'utiliser notre service a été explicité.
- *Compatibilité avec les profils de restriction.* La comptabilité avec les restrictions requises par les applications critiques a été l'un de nos principales préoccupations lors de la proposition de ce service. La compatibilité avec le profil `Ravenscar` a été obtenue grâce à la définitions des interactions avec les services intergiciels de base. Nous avons également évité l'utilisation de l'orienté objets lors de la mise en oeuvre de ce service. Nous nous sommes basés sur les types paramétrés d'`Ada` comme alternative. En particulier nous avons proposé une implantation originale du patron "stratégie" évitant le polymorphisme.
- *Analyse du code source et mesures de performances.* Les mesures de performance de plusieurs cas d'utilisation de ce service montrent des temps de réponse assez courts et bien distribués autour de la moyenne.

Conclusions

Les différents résultats et contributions présentés si dessus montrent que l'application du principe de séparation des préoccupations, la définition et l'utilisation systématique des abstractions, des patrons de conceptions et la maîtrises des interactions entre les modules permettent de répondre à des besoins difficilement conciliables comme la réduction des coûts, le comportement temporel stable, et la sûreté de fonctionnement. Nous avons fourni à travers l'étude de deux exemples de services nécessaires à plusieurs applications tolérantes aux fautes et/ou critiques des éléments de réponse à cette problématique. Le service de tolérance aux fautes que nous avons intégré dans l'architecture schizophrène propose des indications utiles pour l'intégration de la tolérance aux fautes sans violer les propriétés de l'architecture initiale. Le service de consensus que nous avons conçu propose une étude poussée du support de cette abstraction par les intergiciels depuis les premières phases de conception et jusqu'aux dernières phases de mise en oeuvre et d'implémentation.

7.2 Perspectives

Les différents travaux que nous avons menés dans le cadre de cette thèse ouvrent plusieurs perspectives. Certains de ces développements ont déjà été entamés. Le service de consensus peut être la base de de travaux d'évaluation du comportement temporel des algorithmes de consensus et de détection de défaillances. Les services intergiciels peuvent être étendus pour supporter d'autres algorithmes. La mise en oeuvre peut être le sujet de plusieurs optimisations de performances. Les possibilités de configuration de l'architecture schizophrène et les nouveaux paramètres introduits par les différents composants que nous introduisons dans cette thèse accentuent le besoin d'automatisation de la configuration et de l'assemblage des applications basées sur les technologies intergicielles.

Support d'algorithmes par le service de consensus

Lors de la conception de notre service de consensus nous avons supposé un modèle de calcul asynchrone avec arrêt sur défaillances. Selon le l'environnement de déploiement de l'application, il doit être possible d'instancier les services de l'intergiciel pour supporter des modèles plus exigeants.

L'évaluation du comportement temporel des algorithmes de consensus et de détection des défaillances se fait généralement d'une manière sommaire et peu précise. Dans la littérature, nous ne trouvons que quelques travaux fournissant des indications précises sur le comportement des algorithmes. Des méthodes comme la simulation sont généralement préférées aux mesures de performances à cause de la difficulté de la mise en oeuvre des algorithmes et des protocoles de mesures dans les environnements distribués. La généricité du service de consensus lui permet de supporter de nouveaux algorithmes avec très peu d'efforts. Une perspective de ce travail serait d'étendre l'ensemble des algorithmes implantés et de dresser une étude comparative des comportements temporels des algorithmes.

Évaluation des propriétés comportementales

La modélisation formelle permet de s'assurer ou d'estimer certaines propriétés et caractéristiques de l'application modélisée. L'utilisation des réseaux de Petri pour la vérification des propriétés comme l'absence d'interbloages et de famine dans les instances d'intergiciels et l'isolation du comportement des différents styles de réplication dans les intercepteurs dédiés encourage l'exploration de mener une vérification comportementale des différentes mises en ouvre des styles de réplication. Notons que malgré les tests et les mesures de performances, les preuves formelles permettent d'augmenter le degré de confiance qu'on peut placer dans un intergiciel.

Modélisation formelle et configuration automatique

La flexibilité et la configurabilité ont toujours figuré parmi les objectifs de conception des différents produits. Cet objectif a été atteint, les paramètres de configuration sont plus nombreux et les exigences de cohérence plus forts. La modélisation architecturale se basant sur des langages de description d'architectures est une extension naturelle de nos travaux. La définition de processus de configuration automatisés partant des besoins des applications et permettant de générer automatiquement l'application finale est une perspective intéressante. Elle permet d'assurer des objectifs d'optimisations importants mais également une traçabilité entre l'expression des besoins et le produit final et garantir automatiquement les propriétés requises. Cette

modélisation architecturale peut être couplée avec la modélisation comportementale pour une meilleure estimation des caractéristiques des produits finaux.

Table des figures

1.1 Application distribuée basée sur un intergiciel 10
1.2 Réplication active 17
1.3 Réplication passive 17
1.4 Triple Modular Redundancy 20

2.1 Intergiciel orienté messages 33
2.2 De l'appel local à l'appel distant 34
2.3 Interactions entre les objets distribués 35
2.4 Le patron Broker [47] 36

3.1 Architecture schizophrène 62
3.2 Services fondamentaux de distribution 62
3.3 Architecture de FT CORBA 67
3.4 Positions des intercepteurs dans l'architecture 72
3.5 Détection et notification des défaillances 76

4.1 Architecture générale 83
4.2 Patron de conception Stratégie 87
4.3 Patron de conception Pont 87
4.4 Interface du composant du consensus 89
4.5 Interfaces du détecteur des défaillances 90
4.6 Interfaces pour la gestion des messages 91
4.7 Architecture dirigée par les évènements 95
4.8 Intégration des composants dans l'architecture schizophrène 100
4.9 Configuration des composants intergiciels 103

5.1 Composants de PolyORB 113
5.2 Impact de FT CORBA sur l'architecture de PolyORB 115
5.3 Structure d'une IOGR 116

6.1 Distributions des latences de bout en bout pour les styles de réplication avec état 145
6.2 Les latences en fonction du degré de réplication 146
6.3 Comportement temporel de la détection et de la notification des défaillances . . . 147
6.4 Coordinateur tournant avec un détecteur de défaillances de classe $\diamond S$ 151
6.5 Coordinateur tournant avec un détecteur de défaillances de classe $\diamond P$ 151
6.6 Comportement temporel du protocole de consensus en utilisant IIOP 152
6.7 Comportement temporel d'un protocole de consensus se basant sur l'exécutif ORK 154

6.8 Comportement temporel d'un protocole de consensus se basant sur l'exécutif `MaRTE OS` 155

Liste des tableaux

1.1 Classes des détecteurs de défaillances 23
1.2 Niveaux de criticité définis par `DO-178B` 25

6.1 SLOCs des modules du service du consensus 150

Bibliographie

[1] T. Abdelzaher, S. Dawson, W.-C. Feng, F. Jahanian, S. Johnson, A. Mehra, T. Mitton, A. Shaikh, K. Shin, Z. Wang, H. Zou, M. Bjorkland, and P. Marron. ARMADA middleware and communication services. *Real-Time Syst.*, 16(2-3) :127–153, 1999.

[2] M. K. Aguilera, G. Le Lann, and S. Toueg. On the impact of fast failure detectors on real-time fault-tolerant systems. In *DISC '02 : Proceedings of the 16th International Conference on Distributed Computing*, pages 354–370, London, UK, 2002. Springer-Verlag.

[3] J. Alves-Foss, W. S. Harrison, P. Oman, and C. Taylor. The MILS Architecture for High-Assurance Embedded Systems. *international journal of embedded systems*, 2005.

[4] J. Aspnes. Fast deterministic consensus in a noisy environment. *Journal of Algorithms*, 45(1) :16–39, Oct. 2002.

[5] J. Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2–3) :165–175, Sept. 2003.

[6] Ö. Babaoglu, A. Bartoli, V. Maverick, S. Patarin, J. Vuckovic, and H. Wu. A framework for prototyping J2EE replication algorithms. In *On the Move to Meaningful Internet Systems 2004 : CoopIS, DOA, and ODBASE, OTM Confederated International Conferences, Agia Napa, Cyprus, October 25-29, 2004, Proceedings, Part II.*

[7] S. Baker. A2A, B2B-now we need M2M (middleware to middleware) technology. In *DOA '01 : Proceedings of the Third International Symposium on Distributed Objects and Applications*, page 5, Washington, DC, USA, 2001. IEEE Computer Society.

[8] R. Baldoni and C. Marchetti. Three-tier replication for FT-CORBA" infrastructures. *Softw. Pract. Exper.*, 33(8) :767–797, 2003.

[9] M. Barborak, A. Dahbura, and M. Malek. The consensus problem in fault-tolerant computing. *ACM Comput. Surv.*, 25(2) :171–220, 1993.

[10] M. T. Bennani. *Tolérance aux fautes dans les systèmes répartis à base d'intergiciels réflexifs standards.* PhD thesis, Institut National des Sciences Appliquées de Toulouse, 2005.

[11] A. Bessani, J. Fraga, and L. Lung. Extending the UMIOP specification for reliable multicast in CORBA. In *OTM Conferences (1)*, pages 662–679, 2005.

[12] K. Birman and R. Cooper. The ISIS project : Real experience with a fault tolerant programming system. *SIGOPS Oper. Syst. Rev.*, 25(2) :103–107, 1991.

[13] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1) :39–59, 1984.

[14] B. Blakeley, H. Harris, and R. Lewis. *Messaging and queueing using the MQI.* McGraw-Hill, Inc., New York, NY, USA, 1995.

[15] A. Bondavalli, I. Mura, and I. Majzik. Automatic dependability analysis for supporting design decisions in UML. In *HASE '99 : The 4th IEEE International Symposium on High-Assurance Systems Engineering*, page 64, Washington, DC, USA, 1999. IEEE Computer Society.

[16] S. Bouchenak, S. Krakowiak, and N. de Palma. Tolérance aux fautes dans les grappes d'applications internet. In *RENPAR'16 / CFSE'4 / SympAAA'2005 / Journées Composants*, 2005.

[17] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems, 2003.

[18] A. Burns and A. J. Wellings. Restricted tasking models. In *IRTAW '97 : Proceedings of the eighth international workshop on Real-Time Ada*, pages 27–32, New York, NY, USA, 1997. ACM.

[19] F. Buschmann, K. Henney, and D. Schmidt. *Pattern-Oriented Software Architecture – Volume 4 – A Pattern Language for Distributed Computing.* Wiley & Sons, New York, NY, USA, 2007.

[20] B. Carré and J. Garnsworthy. SPARK, an annotated ada subset for safety-critical programming. In *TRI-Ada '90 : Proceedings of the conference on TRI-ADA '90*, pages 392–402, New York, NY, USA, 1990. ACM.

[21] CEP. Complex event processing applications, products, research, and developments in event processing. `http://www.complexevents.com`, 2007.

[22] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2) :225–267, 1996.

[23] K. M. Chandy and R. Schulte. What is Event Driven Architecture (EDA) and Why Does it Matter ?, 2007.

[24] D. Chemouil. The design of spacecraft on-board software. In *B 2007 : Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17-19, 2007, Proceedings*, page 3, 2007.

[25] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Trans. Comput.*, 51(1) :13–32, 2002.

[26] C. Colket. Ada semantic interface specification (asis) : frequently asked questions. *Ada Lett.*, XV(4) :50–63, 1995.

[27] C. Comar, R. Dewar, and G. Dismukes. Certification and object orientation : The new ada answer. In *Conference ERTS'06*, Toulouse, France, Jan. 2006.

[28] A. Corsaro, D. Schmidt, R. Klefstad, and C. O'Ryan. Virtual component : a design pattern for memory-constrained embedded applications, 2002.

[29] F. Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2) :56–78, 1991.

[30] J. da Silva Fraga, F. Siqueira, and F. Favarim. An adaptive fault-tolerant component model. In *9th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003 Fall), 1-3 October 2003, Anacapri (Capri Island), Italy*, pages 179–186, 2003.

[31] A. M. Déplanche, P. Y. Théaudière, and Y. Trinquet. Implementing a semi-active replication strategy in CHORUS/classix, a distributed real-time executive. In *SRDS '99 : Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, page 90, Washington, DC, USA, 1999. IEEE Computer Society.

[32] B. Dobbing and A. Burns. The Ravenscar tasking profile for high integrity real-time programs. In *Proceedings of SigAda'98*, Washington, DC, USA, Nov. 1998.

[33] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1) :77–97, 1987.

[34] B. Dumant, F. Horn, F. D. Tran, and J.-B. Stefani. Jonathan : an open distributed processing environment in java. *Distributed Systems Engineering*, 6(1) :3–12, 1999.

[35] R. C. Durst, G. J. Miller, and E. J. Travis. TCP extensions for space communications. *Wirel. Netw.*, 3(5) :389–403, 1997.

[36] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2) :288–323, 1988.

[37] X. Défago. *Agreement-Related Problems : From Semi-Passive Replication to Totally Ordered Broadcast*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 2000.

[38] G. T. Edwards, G. Deng, D. C. Schmidt, A. S. Gokhale, and B. Natarajan. Model-driven configuration and deployment of component middleware publish/subscribe services. In *GPCE*, pages 337–360, 2004.

[39] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2) :114–131, 2003.

[40] S. Evangelista, C. Kaiser, J. F. Pradat-Peyre, and P. Rousseau. Verifying linear time temporal logic properties of concurrent ada programs with quasar. In *SigAda '03 : Proceedings of the 2003 annual ACM SIGAda international conference on Ada*, pages 17–24, New York, NY, USA, 2003. ACM.

[41] P. Ezhilchelvan, A. Mostefaoui, and M. Raynal. Randomized multivalued consensus. In *ISORC '01 : Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, page 195, Washington, DC, USA, 2001. IEEE Computer Society.

[42] J.-C. Fabre and T. Pérennou. Friends - A flexible architecture for implementing fault tolerant and secure distributed applications. In *EDCC-2 : Proceedings of the Second European Dependable Computing Conference on Dependable Computing*, pages 3–20, London, UK, 1996. Springer-Verlag.

[43] P. Felber. *The CORBA object group service*. PhD thesis, Lausanne, 1998.

[44] P. Felber and P. Narasimhan. Experiences, strategies, and challenges in building fault-tolerant CORBA systems. *IEEE Transactions on Computers*, 53(5) :497–511, 2004.

[45] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2) :374–382, 1985.

[46] C. Flaviu and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. Parallel Distrib. Syst.*, 10(6) :642–657, 1999.

[47] C. Francu. An advanced communication toolkit for implementing the broker pattern. In *ICDCS '99 : Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 458, Washington, DC, USA, 1999. IEEE Computer Society.

[48] R. Friedman and E. Hadad. FTS : A high-performance CORBA fault-tolerance service. In *WORDS '02 : Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, page 61, Washington, DC, USA, 2002. IEEE Computer Society.

[49] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1994.

[50] B. Garbinato and R. Guerraoui. Using the strategy design pattern to compose reliable distributed protocols. In *COOTS'97 : Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 17–17, Berkeley, CA, USA, 1997. USENIX Association.

[51] F. C. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1) :1–26, 1999.

[52] R. Guerraoui and A. Schiper. Consensus : The big misunderstanding. In *FTDCS '97 : Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '97)*, page 183, Washington, DC, USA, 1997. IEEE Computer Society.

[53] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Trans. Softw. Eng.*, 27(1) :29–41, 2001.

[54] F. Guidec, Y. Maho, and L. Courtrai. A java middleware platform for resource-aware distributed applications, 2003.

[55] A. Hall and R. Chapman. Correctness by construction : Developing a commercial secure system. *IEEE Softw.*, 19(1) :18–25, 2002.

[56] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena : an integrated development, analysis, and verification environment for component-based systems. In *ICSE '03 : Proceedings of the 25th International Conference on Software Engineering*, pages 160–173, Washington, DC, USA, 2003. IEEE Computer Society.

[57] D. A. Haverkamp and R. J. Richards. Towards safety critical middleware for avionics applications. In *LCN '02 : Proceedings of the 27th Annual IEEE Conference on Local Computer Networks*, page 0716, Washington, DC, USA, 2002. IEEE Computer Society.

[58] N. Hayashibara, P. Urbán, A. Schiper, and T. Katayama. Performance comparison between the Paxos and Chandra-Toueg Consensus algorithms. Technical Report IC/2002/61, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, August 2002.

[59] M. Henning. A new approach to object-oriented middleware. *IEEE Internet Computing*, 8(1) :66–75, 2004.

[60] M. Henning, M. Spruiell, D. Boone, B. Eagles, B. Foucher, M. Laukien, M. Newhook, and B. Normier. *Distributed Programming with Ice, revision 3.2*. ZeroC, Inc. http ://www.zeroc.com, 2007.

[61] J. F. Hermant and G. Le Lann. Fast asynchronous uniform consensus in real-time distributed systems. *IEEE Trans. Comput.*, 51(8) :931–944, 2002.

[62] M. Hiller. Software fault tolerance techniques from a realtime systems point of view : An overviewtechnical report no, 1998.

[63] J. Hugues. *Architecture et Services des Intergiciels Temps Réel*. Thèse de doctorat, ENST, 2005.

[64] J. Hugues, Y. Thierry-Mieg, F. Kordon, L. Pautet, S. Baarir, and T. Vergnaud. On the Formal Verification of Middleware Behavioral Properties. In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, volume ENTCS 133, pages 139–157, Linz, Austria, Sept. 2004. Elsevier.

[65] M. Ibrahim and O. Etzion. Workshop on event driven architecture. In *OOPSLA '06 : Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 624–624, New York, NY, USA, 2006. ACM Press.

[66] IEEE. *1996 (ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1 : System Application : Program Interface (API) [C Language].* 1996.

[67] I. Inc. Ada 9x reference manual, 1995.

[68] IONA and Isis. An introduction to orbix+isis, IONA technologies and isis distributed systems. Technical report, 1994.

[69] Z. T. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant. Chameleon : A software infrastructure for adaptive fault tolerance. *IEEE Trans. Parallel Distrib. Syst.*, 10(6) :560–579, 1999.

[70] Y. Kermarrec, L. Nana, and L. Pautet. GNATDIST : a configuration language for distributed ada 95 applications. In *TRI-Ada '96 : Proceedings of the conference on TRI-Ada '96*, pages 63–72, New York, NY, USA, 1996. ACM.

[71] J. Kienzle and A. Romanovsky. On persistent and reliable streaming in ada. In H. B. Keller and E. Plöderer, editors, *International Conference on Reliable Software Technologies - Ada-Europe'2000, Potsdam, Germany, June 26-30, 2000*, number 1845, pages 82–95, 2000.

[72] K. H. K. Kim. Object-oriented real-time distributed programming and support middleware. In *ICPADS '00 : Proceedings of the Seventh International Conference on Parallel and Distributed Systems (ICPADS'00)*, page 10, Washington, DC, USA, 2000. IEEE Computer Society.

[73] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and M. Ahamad. Efficient implementations of java remote method invocation (RMI). In *COOTS'98 : Proceedings of the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 2–2, Berkeley, CA, USA, 1998. USENIX Association.

[74] J.-C. Laprie and B. Randell. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1) :11–33, 2004. Fellow-Algirdas Avizienis and Senior Member-Carl Landwehr.

[75] M. Larrea, A. Fernndez, and S. Arvalo. the impossibility of implementing perpetual failure detectors in partially synchronous systems, 2001.

[76] G. Le Lann. On real-time and non real-time distributed computing. In *WDAG '95 : Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 51–70, London, UK, 1995. Springer-Verlag.

[77] G. Le Lann. Asynchrony and real-time dependable computing. *words*, 00 :18, 2003.

[78] L. C. Lung, F. Favarim, G. T. Santos, M. C. D. C. Schmidt, and F. Buschmann. An infrastructure for adaptive fault tolerance on FT-CORBA. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 504–511, 2006.

[79] S. Maffeis. A flexible system design to support object-groups and object-oriented distributed programming. Technical Report IFI TR 94.02, Univ. of Zurich, Apr. 1994.

[80] S. Maffeis. The object group design pattern. Technical report, Ithaca, NY, USA, 1996.

[81] I. Majzik and G. Huszerl. Towards dependability modeling of FT-CORBA architectures. In *EDCC-4 : Proceedings of the 4th European Dependable Computing Conference on Dependable Computing*, pages 121–139, London, UK, 2002. Springer-Verlag.

[82] I. Majzik and G. Huszerl. Towards dependability modeling of FT-CORBA architectures. In *EDCC-4 : Proceedings of the 4th European Dependable Computing Conference on Dependable Computing*, pages 121–139, London, UK, 2002. Springer-Verlag.

[83] C. Marchetti, L. Verde, and R. Baldoni. CORBA request portable interceptors : A performance analysis. In *DOA*, page 208, 2001.

[84] Microsoft. Strategies for fault-tolerant computing. *White Paper*, 2003.

[85] A. Mostefaoui and M. Raynal. Consensus based on failure detectors with a perpetual accuracy property. In *IPDPS '00 : Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 514, Washington, DC, USA, 2000. IEEE Computer Society.

[86] P. Narasimhan, T. Dumitras, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava. MEAD : support for real-time fault-tolerant CORBA. *Concurrency - Practice and Experience*, 17(12) :1527–1545, 2005.

[87] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Eternal : a component-based framework for transparent fault-tolerant CORBA. *Softw. Pract. Exper.*, 32(8) :771–788, 2002.

[88] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Shmidt. DOORS : Towards high-performance fault tolerant CORBA. *Proceedings of the 2nd Intenational Symposium on Distributed Objects and Applications (Antwerpen,Belgium) pp. 39-48*, 2000.

[89] B. Natarajan, A. S. Gokhale, S. Yajnik, and D. C. Schmidt. Applying patterns to improve the performance of fault tolerant CORBA. In *HiPC '00 : Proceedings of the 7th International Conference on High Performance Computing*, pages 107–120, London, UK, 2000. Springer-Verlag.

[90] Object Management Group. *Data Distribution Service for Real-time Systems Specification, version 1.0.* OMG, Mar. 2004. OMG Technical Document.

[91] OMG. *Common Object Request Broker Architecture : Core Specification, Version 3.0.3.* OMG, Mar. 2004. OMG Technical Document formal/04-03-12.

[92] OMG. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, Version 3.0.3.* OMG, June 2004. OMG Adopted Specification ptc/2004-06-01.

[93] L. Pautet. *Intergiciels schizophrènes : une solution à l'interopérabilité entre modèles de répartition*. Habilitation à diriger des recherches, Université Pierre et Marie Curie, 2001.

[94] L. Pautet and S. Tardieu. GLADE : A framework for building large object-oriented real-time distributed systems. In *ISORC '00 : Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 244, Washington, DC, USA, 2000. IEEE Computer Society.

[95] P. Plancke, P. David, C. Plummer, and et al. Standards for On Board Data Systems : An Updated View. In *ESA Special Publication*, volume 602 of *ESA Special Publication*, Aug. 2005.

[96] A. Polze. Building blocks for achieving quality of service with commercial off-the-shelf (COTS) middleware. Technical Report CMU/SEI-99-TR-001, 1999.

[97] A. Polze and M. Malek. Network computing with sonic. *J. Syst. Archit.*, 44(3-4) :169–187, 1997.

[98] A. Polze, J. Schwarz, and M. Malek. Automatic generation of fault-tolerant CORBA-services. In *TOOLS '00 : Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, page 205, Washington, DC, USA, 2000. IEEE Computer Society.

[99] D. Powell. Failure mode assumptions and assumption coverage. In D. K. Pradhan, editor, *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing (FTCS '92)*, pages 386–395, Boston, MA, 1992. IEEE Computer Society Press.

[100] T. Quinot. *Conception et réalisation d'un intergiciel schizophrène pour la mise en oeuvre de systèmes répartis interopérables.* Thèse de doctorat, Université Pierre et Marie Curie, 2003.

[101] V. Quéma and L. Bellissard. Configuration de middleware dirigée par les applications. In *Journées sur les Systèmes à Composants Adaptables et Extensibles*, Grenoble, France, 2002.

[102] K. Raman, Y. Zhang, M. Panahi, J. A. Colmenares, R. Klefstad, and T. Harmon. RTZen : Highly predictable, real-time java middleware for distributed and embedded systems, . In G. Alonso, editor, *Middleware*, volume 3790 of *Lecture Notes in Computer Science*, pages 225–248. Springer, 2005.

[103] B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surv.*, 10(2) :123–165, 1978.

[104] H. P. Reiser, U. Bartlang, and F. J. Hauck. A reconfigurable system architecture for consensus-based group communication. In *International Conference on Parallel and Distributed Computing Systems, PDCS 2005, November 14-16, 2005, Phoenix, AZ, USA*, pages 680–686, 2005.

[105] Y. J. Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. Sanders, R. E. Schantz, and M. Seri. AQuA : An adaptive architecture that provides dependable distributed objects. *IEEE Trans. Comput.*, 52(1) :31–50, 2003.

[106] R. V. Renesse, K. P. Birman, B. B. Glade, K. Guo, M. Hayden, T. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus : A flexible group communications system. Technical Report TR95-1500, 1995.

[107] M. Rodriguez, J.-C. Fabre, and J. Arlat. Wrapping real-time systems from temporal logic specifications. In *EDCC-4 : Proceedings of the 4th European Dependable Computing Conference on Dependable Computing*, pages 253–270, London, UK, 2002. Springer-Verlag.

[108] RTCA. Software considerations in airborne systems and equipment certification. DO-178B / ED-12B, 1992.

[109] SAE. *Architecture Analysis & Design Language (AS5506)*, sept. 2004. available at `http://www.sae.org`.

[110] D. Schmidt and C. Cleeland. Applying patterns to develop extensible and maintainable ORB middleware. *Communications of the ACM, CACM*, 40(12), 1997.

[111] D. C. Schmidt. Middleware for real-time and embedded systems. *Commun. ACM*, 45(6) :43–48, 2002.

[112] D. C. Schmidt and F. Buschmann. Patterns, frameworks, and middleware : their synergistic relationships. In *ICSE '03 : Proceedings of the 25th International Conference on Software Engineering*, pages 694–704, Washington, DC, USA, 2003. IEEE Computer Society.

[113] E. Schneider, F. Picioroagă, and U. Brinkschulte. Dynamic reconfiguration through osa+, a real-time middleware. In *DSM '04 : Proceedings of the 1st international doctoral symposium on Middleware*, pages 319–323, New York, NY, USA, 2004. ACM.

[114] F. B. Schneider and L. Lamport. Paradigms for distributed programs. In *Distributed Systems : Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985 Munich*, pages 431–480, London, UK, 1985. Springer-Verlag.

[115] N. Sergent. Performance evaluation of a consensus algorithm with petri nets. In *PNPM '97 : Proceedings of the 6th International Workshop on Petri Nets and Performance Models*, page 143, Washington, DC, USA, 1997. IEEE Computer Society.

[116] N. Sergent, X. Défago, and A. Schiper. Impact of a failure detection mechanism on the performance of consensus. In *PRDC '01 : Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing*, page 137, Washington, DC, USA, 2001. IEEE Computer Society.

[117] L. Sha, R. Rajkumar, and M. Gagliardi. A software architecture for dependable and evolvable industrial computing systems. Technical report, Software Engineering Institute, Carnegie Mellon, 1995.

[118] E. Shokri and H. Hecht. Matching software fault tolerance with application needs. In *HASE '98 : The 3rd IEEE International Symposium on High-Assurance Systems Engineering*, page 248, Washington, DC, USA, 1998. IEEE Computer Society.

[119] A. Singhai, A. Sane, and R. H. Campbell. Quarterware for middleware. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 1998.

[120] A. Singhai, A. Sane, and R. H. Campbell. Reflective ORBs : Supporting robust, time-critical distribution. In *ECOOP '97 : Proceedings of the Workshops on Object-Oriented Technology*, pages 55–61, London, UK, 1998. Springer-Verlag.

[121] J. M. Spivey. *Understanding Z : A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Jan. 1988.

[122] R. Stroud, I. Welch, J. Warne, and P. Ryan. A qualitative analysis of the intrusion-tolerance capabilities of the MAFTIA architecture. In *DSN '04 : Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 453, Washington, DC, USA, 2004. IEEE Computer Society.

[123] V. Subramonian, G. Xing, C. D. Gill, C. Lu, and R. Cytron. Middleware specialization for memory-constrained networked embedded systems. In *RTAS '04 : Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, page 306, Washington, DC, USA, 2004. IEEE Computer Society.

[124] M. Swanson, L. Stoller, T. Critchlow, and R. Kessler. The design of the schizophrenic workstation system. pages 291–306, 1993.

[125] T. Vergnaud. *Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique d'applications formellement vérifiées*. Thèse de doctorat, ENST, 2006.

[126] T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon. 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)*, volume LNCS 3063, pages 106–119, Palma de Mallorca, Spain, June 2004. Springer Verlag.

[127] T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon. Rapid Development Methodology for Customized Middleware. In *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*, pages 111–117, Montreal, Canada, June 2005. IEEE.

[128] S. Vinoski. Chain of responsibility. *IEEE Internet Computing*, 6(6) :80–83, 2002.

[129] S. Vinoski. An overview of middleware. In *Reliable Software Technologies - Ada-Europe 2004, 9th Ada-Europe International Conference on Reliable Software Technologies, Palma de Mallorca, Spain, June 14-18*, pages 35–51, 2004.

[130] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. Design and implementation of a pluggable fault-tolerant CORBA infrastructure. *Cluster Computing*, 7(4) :317–330, 2004.