ÉTUDE DE L'INVOCATION ENTRE OBJETS DUPLIQUÉS DANS UN SYSTÈME RÉPARTI TOLÉRANT AUX FAUTES

THÈSE N° 1578 (1996)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Karim Riad MAZOUNI

DEA en systèmes informatiques, Université Pierre et Marie Curie (Paris VI)

originaire de Fribourg et de la Tour-de-Trême (FR)

acceptée sur proposition du jury:

Prof. A. Schiper, directeur de thèse
Dr D. Powell, corapporteur
Dr M. Shapiro, corapporteur
Prof, A. Strohmeier, corapporteur

Lausanne, EPFL
1996

Résumé

Cette thèse étudie les problèmes à résoudre afin de pouvoir utiliser la notion d'invocation, pour exprimer la communication entre des objets dupliqués, dans le contexte d'un système réparti tolérant aux fautes. L'objectif de cette étude est de définir des abstractions permettant *d'encapsuler la duplication*, c.-à-d. de voir la duplication comme une caractéristique interne d'un objet. Ainsi, la communication avec un objet (dupliqué ou non) se ferait toujours avec la même paradigme: l'invocation.

Contexte

Une invocation est une interaction de type "requête-réponse". Ce mode d'interactions correspond exactement au *modèle client-serveur* qui exprime la communication entre les services d'un système réparti. Pour cette raison, *l'approche orientée-objets* s'avère particulièrement adaptée à la conception d'un système reparti. Dès lors, un service peut être réalisé à l'aide d'un ou plusieurs objets, répartis sur des ordinateurs (ou *nœuds*) distincts. La mise en œuvre de la tolérance aux fautes d'un service consiste alors à dupliquer les objets réalisant ce service.

La plupart des travaux portant sur les objets dupliqués ne considèrent *que la duplication d'objets serveurs*. L'étude menée au cours de cette thèse se veut plus générale: les objets dupliqués peuvent être à la fois clients et serveurs. En outre, les stratégies de duplication des objets peuvent être différentes. Quatre stratégies de duplication sont étudiées: la duplication active, la duplication passive, la duplication semi-active, et la duplication coordinateur-cohorte.

Encapsulation de la duplication

Encapsuler la duplication consiste à encapsuler la pluralité et à encapsuler les stratégies de duplication. *Encapsuler la pluralité* consiste à cacher, aux autres objets du système, qu'un objet dupliqué est constitué par un ensemble de copies réparties sur plusieurs nœuds. *Encapsuler les stratégies de duplication* consiste à cacher, aux autres objets du système, le protocole qu'il faut utiliser pour communiquer avec les copies d'un objet dupliqué, afin que ces copies restent cohérentes.

Modélisation symétrique de l'invocation

La plupart des travaux décrits dans la littérature s'appuient sur une *modélisation asymétrique* de l'invocation entre objets dupliqués. Cette modélisation considère que la réponse est toujours transmise en suivant exactement le chemin inverse de la requête. Le principal inconvénient de cette modélisation est qu'elle ne permet pas de réaliser l'encapsulation de la duplication

d'un objet client. Cette étude propose donc une alternative, appelée *modélisation symétrique*, qui ne présente pas cet inconvénient. La modélisation symétrique considère la transmission de la requête et la transmission de la réponse comme deux instances du même problème: la transmission d'un message à un objet dupliqué.

L'analyse des problèmes liés à l'encapsulation de la duplication et la modélisation symétrique ont permis de construire une spécification de l'invocation entre objets dupliqués. Cette spécification est constituée par un ensemble de propriétes paramétrées par les stratégies de duplication des objets participant à l'invocation. Ces propriétes intègrent notamment la sémantique choisie pour exprimer les informations sur les défaillances. Cette sémantique correspond à l'abstraction de groupe augmentée de la communication vue-synchrone. Chaque objet dupliqué est modélisé à l'aide d'un groupe d'objets dont la composition varie en fonction des défaillances/redémarrages des copies.

Le service d'invocation N2M

Les résultats de cette étude ont conduit à la conception d'un service d'invocation pour objets dupliqués, appelé N2M (par référence à l'expression anglaise *n to m* caractérisant la communication entre n copies d'un client et m copies d'un serveur). Les objets utilisant le service N2M sont appelés *objets application*, par opposition aux *objets communication* qui réalisent le service N2M. Les objets communication prennent en charge tous les aspects relatifs à la duplication des objets application. Par conséquent, les objets application dupliqués communiquent à l'aide d'invocations, comme s'ils n'étaient pas dupliqués.

Pour construire cette abstraction, le service N2M définit deux catégories d'objets communication: les *encapsulateurs* et les *messagers*. Un objet application dupliqué est construit à partir d'un groupe d'encapsulateurs. À chaque copie de l'objet application dupliqué est associé un encapsulateur chargé de filtrer les invocations entrant ou sortant de la copie. Pour communiquer avec un objet application dupliqué O, un objet doit s'adresser au messager local de O. Sur chaque noeud, un objet application dupliqué est représenté par un messager chargé de transmettre les requêtes et les réponses vers les copies de l'objet application dupliqué, via leurs encapsulateurs respectifs.

L'originalité de ce modèle réside dans sa symétrie: au cours d'une invocation, il y a des messagers du serveur sur les nœuds des copies du client, et des messagers du client sur les nœuds des copies du serveur. Cette symétrie découle directement de la modélisation symétrique de l'invocation.

Une stratégie de duplication est mise en œuvre à l'aide d'une classe d'encapsulateurs et d'une classe de messagers. Ces classes permettent de dupliquer des objets selon la stratégie considérée tout en respectant le schéma de communication défini par la notion d'invocation. En d'autres termes, les classes d'objets communication (i.e. encapsulateurs et messagers) réalisent l'encapsulation de la duplication.

L'environnement GARF-v2

Le service N2M a été mis en œuvre dans le contexte du projet GARF (Génération Automatique d'applications Résistantes aux Fautes). L'objectif du projet GARF était la réalisation d'un environnement de programmation visant à faciliter le développement d'applications réparties tolérantes aux fautes. Le prototype de l'environnement a été réalisé en Smalltalk et il se base sur le logiciel ISIS pour les communications de groupe.

Abstract

This dissertation studies the problems to solve in order to use the invocation paradigm to express replicated object communication in fault-tolerant distributed systems. The ultimate goal is to define abstractions which achieve *replication encapsulation*, ie which give the illusion that replication is an internal property of objects. Thus, object communication could be always expressed using the invocation paradigm, whether objects are replicated or not.

Background

The invocation paradigm defines a "request-reply" communication model which matches exactly the *client-server model*. The latter is generally used to express service interactions in distributed systems. For this reason, the *object-oriented approach* is well suited to the design of distributed system services. A service can be implemented as a set of objects, located on remote computers (or *nodes*). Service fault-tolerance is achieved by replicating the objects which implement the service.

Most of the previous works about replicated objects consider *only server object replication*. This study is more general: both client and server can be replicated. Furthermore, replication policies of objects can be different. Four replication policies had been studied: active replication, passive replication, semi-active replication, and coordinator-cohort replication.

Replication encapsulation

Replication encapsulation means both plurality encapsulation and replication policy encapsulation. *Plurality encapsulation* consists in hiding from other objects, that a replicated object is actually a set of replicas located on several nodes. Replication policy encapsulation consists in hiding from other objects, the communication protocol to use in order to interact with object replicas without breaking their consistency.

The symmetric invocation model

Most of the related works are based on an *asymmetric invocation model*. In this model, the invocation reply follows exactly the reverse of the request communication path. The asymmetric invocation model can not be used to achieve replication encapsulation of client objects. This dissertation proposes a *symmetric invocation model* which solves this problem. The symmetric invocation model considers the request transmission and the reply transmission as two instances of the same problem: the transmission of a message to a replicated object.

Both the analysis of the replication encapsulation problem and the symmetric invocation model

were used to define a specification of replicated object invocation. This specification is a set of generic formal properties based on parameters which values depend on replication policies. The properties include object failure semantics which is expressed using the group paradigm and the view-synchronous communication paradigm. Every replicated object is built using an object group which membership changes whenever object replicas fail or restart.

The N2M invocation service

The main result of this study is the design[1] of N2M, an invocation service which supports replicated objects. Objects using N2M are called *application objects* whereas objects implementing N2M are called *communication objects*. Communication objects take care of every aspect related to application object replication. Thus, replicated application objects communicate using regular invocations, just as if they were not replicated.

There are actually two kinds of communication objects: *encapsulators* and *mailers*. Each replicated application object is built using an encapsulator group. Each application object replica is associated with a private encapsulator which acts as an invocation filter for this replica. To communicate with a replicated application object O, every object must interact with O's local mailer. On every node, a replicated application object is represented by a mailer which is responsible for transmitting requests and replies to the application object replicas.

The originality of this model is its symmetry: there are both mailers of the server on the client nodes, and mailers of the client on the server nodes. This symmetry is directly inherited from the symmetric invocation model.

Each replication policy is implemented using an encapsulator class and a mailer class. These classes replicate objects according to a specific replication policy, while respecting the invocation paradigm. In other words, communication object classes achieve replication encapsulation.

The GARF-v2 programming environment

The N2M service has been implemented in the context of the GARF project[2]. The GARF project aimed to provide a programming environment which faciltates the design of fault-tolerant distributed applications. The environment prototype was implemented in Smalltalk. It is based on the group communication layer provided by ISIS toolkit.

[1] this logo refers to the *n to m* expression which usually names the interaction between n client replicas and m server replicas.

[2] GARF is the french acronym for automatic generation of fault-tolerant applications.

à l'Autre, sans oublier sa sœur;

à Nedjma, lointaine mais omniprésente;

à Hélène et Roger, disparus dans la tourmente de l'année 1994.

Remerciements

Je tiens à remercier André Schiper pour la confiance qu'il m'a témoignée, tout au long de ma thèse, mais aussi au cours de toutes mes activités au sein du Laboratoire de Systèmes d'Exploitation. La liberté d'action qu'il m'a laissée, ainsi que ses conseils, m'ont été précieux.

Je tiens à remercier Messieurs Boi Faltings, David Powell, Marc Shapiro et Alfred Strohmeier, d'avoir accepté de composer mon jury de thèse, et de juger ce travail.

Je tiens à remercier Benoît Garbinato et Rachid Guerraoui pour leur collaboration fructueuse tout au long du projet GARF. Cette collaboration a grandement contribué à m'inspirer les réflexions décrites dans cette thèse.

Je tiens à remercier Benoît Garbinato (encore lui!) et François Pacull d'avoir pris le temps de relire scrupuleusement ce rapport. Je tiens aussi à remercier Kristine Verhamme, pour sa disponibilité et son assistance lorsqu'il s'agissait de décrypter les circulaires et autres textes administratifs.

Last but not least, un grand merci à Marielle, Nabila, Tariq, à mes parents, à mes amis (spécialement mes amis de 30 ans!) pour leur soutien constant tout au long de ce travail.

Table des matières

1 Introduction **1**

1.1 Contexte 1

1.2 Objectifs 2

1.3 Contributions 3

1.4 Organisation 4

2 La duplication dans les systèmes répartis **5**

2.1 Introduction 5

2.2 Tolérance aux fautes 6

2.2.1 Définitions 7

2.2.2 Enjeux de la tolérance aux fautes 10

2.2.3 Techniques de tolérance aux fautes 10

2.3 Tolérance aux fautes dans un système réparti 12

2.3.1 Modélisation d'un système réparti 12

2.3.2 Défaillances dans un système réparti 13

2.3.3 Mécanismes de base pour la tolérance aux fautes 14

2.3.4 Tolérance aux fautes par duplication dans un système réparti 18

2.4 Modèle client-serveur et duplication 19

2.4.1 Définitions 19

2.4.2 Duplication des clients et des serveurs 20

2.5 Stratégies de duplication 24

2.5.1 Notion de stratégie de duplication 24

2.5.2 Duplication active 24

2.5.3 Duplication passive 25

2.5.4 Duplication semi-active 27

2.5.5 Duplication coordinateur-cohorte 28

2.6 Exemples de mises en œuvre de la duplication 29

2.6.1 ISIS 30

2.6.2 CIRCUS 30
2.6.3 DELTA-4 31
2.7 Conclusion 32

3 Les objets dupliqués dans les systèmes répartis 35
3.1 Introduction 35
3.2 Définitions 37
3.3 Notion d'objet dupliqué 38
3.4 Vers une encapsulation de la pluralité 38
3.4.1 Synchroniseur d'objets 39
3.4.2 Mandataire 39
3.4.3 Objet dispersé 41
3.4.4 Troupeau d'objets 42
3.4.5 Communauté d'objets 43
3.5 Vers une encapsulation des stratégies de duplication 45
3.5.1 Objets fragmentés dans SOS 45
3.5.2 Groupes d'interfaces dans ANSA 46
3.5.3 Objets dupliqués dans ARJUNA 48
3.5.4 Systèmes à objets basés sur ISIS 50
3.6 Conclusion 52

4 L'invocation entre objets dupliqués: analyse et modélisation 53
4.1 Introduction 53
4.2 Hypothèses de base 54
4.2.1 Hypothèses relatives au système réparti 54
4.2.2 Hypothèses relatives aux objets 55
4.3 Encapsulation de la duplication 57
4.3.1 Étude de quelques exemples 57
4.3.2 Classification des problèmes rencontrés 61
4.3.3 Modélisation symétrique de l'invocation entre objets dupliqués 64
4.3.4 Spécification de l'invocation entre objets dupliqués 66
4.4 Prise en compte des défaillances 69
4.4.1 Expression des informations sur les défaillances 69
4.4.2 Étude de quelques exemples 72
4.4.3 Spécification étendue de l'invocation entre objets dupliqués 77
4.5 Application aux stratégies de duplication 81

4.5.1 Duplication active . . . 81
4.5.2 Duplication passive . . . 84
4.5.3 Duplication semi-active . . . 87
4.5.4 Duplication coordinateur-cohorte . . . 90
4.5.5 Synthèse . . . 92
4.6 Conclusion . . . 93

5 N2M, un service d'invocation pour objets dupliqués 95
5.1 Introduction . . . 95
5.2 Représentants symétriques . . . 96
5.2.1 Notion de représentants symétriques . . . 96
5.2.2 Rôle des représentants symétriques . . . 98
5.2.3 Représentants vs mandataires . . . 99
5.3 Encapsulateurs et messagers . . . 101
5.3.1 Modélisation des représentants symétriques . . . 101
5.3.2 Notion d'encapsulateur . . . 103
5.3.3 Notion de messager . . . 104
5.4 Encapsulation de la pluralité dans le service N2M . . . 104
5.4.1 Introduction . . . 104
5.4.2 Structure des objets application dupliqués . . . 105
5.4.3 Désignation des objets application dupliqués . . . 106
5.4.4 Synthèse . . . 108
5.5 Encapsulation des stratégies de duplication dans le service N2M . . . 108
5.5.1 Introduction . . . 109
5.5.2 Désignation des messages liés aux invocations . . . 109
5.5.3 Choix des multicasts . . . 110
5.5.4 Pré-filtrage . . . 111
5.5.5 Post-filtrage . . . 116
5.5.6 Pré-filtrage vs post-filtrage . . . 117
5.5.7 Synthèse . . . 118
5.6 Application aux stratégies de duplication . . . 119
5.6.1 Duplication active . . . 119
5.6.2 Duplication passive . . . 119
5.6.3 Duplication semi-active . . . 119
5.6.4 Duplication coordinateur-cohorte . . . 120
5.7 Conclusion . . . 120

6 Mise en œuvre du service N2M dans l'environnement GARF-v2 **121**
6.1 Introduction 121
6.2 Présentation du projet GARF 122
6.2.1 Objectifs 122
6.2.2 Démarche et résultats 122
6.3 Architecture de l'environnement GARF-v2 124
6.4 Couche Application 125
6.5 Couche Runtime 127
6.6 Couche Communication 129
6.6.1 Mise en œuvre des objets communication 129
6.6.2 Gestion des invocations 131
6.7 Couche Réseau 132
6.7.1 Pilotage d'un nœud 133
6.7.2 Interface avec ISIS 135
6.8 Conclusion 136

7 Conclusions **139**
7.1 Bilan 139
7.2 Perspectives 140

Liste des figures **141**

Liste des tableaux **145**

Bibliographie **147**

A Récapitulation des 16 cas possibles **153**

B Curriculum Vitae **159**

Chapitre 1

Introduction

Ce premier chapitre situe brièvement le contexte de ce travail de thèse, énonce les objectifs visés, présente les contributions apportées, et présente l'organisation de ce rapport.

1.1 Contexte

Le contexte de ce travail de thèse comporte deux volets: les systèmes répartis tolérants aux fautes et l'approche orientée-objets.

Les systèmes répartis tolérants aux fautes

L'avènement des systèmes informatiques répartis est certainement l'un des faits les plus marquants de l'histoire de l'informatique de ces dix dernières années. Un système informatique réparti permet à ses utilisateurs d'accéder de façon *uniforme* aux ressources informatiques (i.e. processeur, mémoire, logiciels, périphériques, ...) de plusieurs ordinateurs interconnectés par un réseau de communication. Les utilisateurs d'un système réparti ont ainsi l'impression de travailler sur une machine unique.

Outre les avantages induits par le partage des ressources informatiques, un système réparti est un support idéal pour construire un système informatique tolérant aux fautes. Dans un système centralisé, la défaillance du seul ordinateur constituant le système paralyse tous les services dépendant de cet ordinateur. Dans ce cas, l'usager de ces services doit attendre le redémarrage du système. Dans un système réparti, la défaillance de l'un des ordinateurs du système n'affecte pas nécessairement les autres ordinateurs du système.

Par conséquent, il est possible d'améliorer la disponibilité d'un service en *dupliquant* ce service sur plusieurs ordinateurs du système. La duplication d'un service consiste à faire en sorte que plusieurs copies du même service s'exécutent simultanément sur des ordinateurs distincts. La défaillance d'un ordinateur interrompt uniquement la copie du service qui s'exécutait sur cette machine. Les autres copies du service sont toujours disponibles.

L'approche orientée-objets

Développée initialement dans le contexte de la programmation, l'approche orientée-objets s'est généralisée, ces dix dernières années, à de nombreux secteurs de l'informatique, comme les bases

de données ou les systèmes répartis.

L'approche orientée-objets consiste à voir un programme informatique comme un ensemble d'objets communiquant selon le modèle client-serveur. Un objet est un composant logiciel qui possède un état et qui définit un ensemble de services (ou opérations) s'appliquant sur cet état. Un objet joue le rôle de serveur lorsqu'il met ses services à la disposition d'autres objets. Un objet joue le rôle de client lorsqu'il utilise les services proposés par un ou plusieurs serveurs. Pour communiquer avec un serveur, un client envoie une requête de service à ce serveur. Celui-ci traite la requête et communique le résultat du traitement au client. Cette forme de communication est généralement appelée invocation.

L'approche orientée-objets est particulièrement adaptée à la conception d'un système réparti. Le modèle client-serveur exprime parfaitement la relation entre les services fournis par un système réparti et les usagers (ou clients) de ces services. Ainsi, chaque service d'un système réparti peut être réalisé à l'aide d'un ou plusieurs objets s'exécutant sur un ou plusieurs ordinateurs constituant le système.

La mise en œuvre de la tolérance aux fautes consiste alors à dupliquer les objets réalisant les services du système réparti. Mais la duplication des objets complique les interactions entre les objets constituant le système réparti. C'est justement sur ce point, l'invocation entre objets dupliqués, que porte ce travail de thèse.

1.2 Objectifs

Les objectifs de ce travail de thèse sont:

1. l'identification et l'analyse des problèmes à résoudre pour permettre à des objets dupliqués de communiquer en utilisant la notion d'invocation;
2. la conception et la réalisation de mécanismes logiciels répondant aux problèmes identifiés précédemment.

Ces objectifs ont été motivés par un double constat:

- la notion de stratégie de duplication entre en conflit avec la notion d'invocation;
- les interférences entre les stratégies de duplication des objets peuvent provoquer des incohérences.

La stratégie de duplication d'un objet définit comment les copies de cet objet reçoivent, traitent et répondent à une requête. Cette notion entre en conflit avec la notion d'invocation qui définit un modèle d'interaction client-serveur général et indépendant des caractéristiques des objets. L'invocation d'un objet consiste à envoyer une requête à l'objet puis à recevoir une réponse correspondant au traitement de la requête par l'objet.

Par exemple, la stratégie de duplication active confère un rôle identique à toutes les copies d'un objet dupliqué. Chaque copie reçoit, traite et répond à toutes les requêtes. Soit S un objet dupliqué activement. Lorsqu'un objet client C invoque S, C doit envoyer la requête à toutes les copies de S. De plus, le client C doit savoir interpréter les multiples réponses qu'il reçoit pour la même requête, puisque chaque copie de S répond à C.

Par conséquent, le client C doit connaître la stratégie de duplication du serveur S pour pouvoir communiquer avec lui. Cet exemple montre que le modèle général de l'invocation n'est plus respecté lorsque l'on considère des objets dupliqués.

Si le client C est aussi dupliqué activement, *chaque* copie de C envoie une requête à toutes les copies de S. Ainsi, chaque copie de S reçoit et traite la même requête plusieurs fois, au risque de rendre son état incohérent si le traitement n'est pas idempotent. Cet exemple illustre comment les interférences entre les stratégies de duplication respectives d'un client et d'un serveur dupliqués risquent de provoquer des incohérences.

1.3 Contributions

Les principales contributions de cette thèse sont:

1. l'analyse et la modélisation des problèmes posés par l'invocation entre objets dupliqués;

2. la conception et la réalisation d'un service d'invocation répondant à ces problèmes.

Analyse et modélisation des problèmes

L'étude décrite au chapitre 4 présente une modélisation générique des problèmes posés par l'invocation entre objets dupliqués. Cette modélisation est générique dans le sens ou elle dépend de paramètres déterminés par les stratégies de duplication des objets impliqués dans l'invocation. Quatre stratégies de duplication[1] fréquemment référencées dans la littérature fournissent les exemples servant à illustrer cette modélisation.

L'originalité de cette modélisation réside dans le fait qu'elle considère l'invocation comme un problème symétrique: la transmission de la requête et la transmission de la réponse sont vues comme deux instances du même problème.

Conception et réalisation d'un service d'invocation

Le chapitre 5 décrit un service d'invocation pour objets dupliqués appelé[2] N2M. Le service N2M est basé sur un modèle de communication original constitué par deux catégories d'objets: les encapsulateurs et les messagers. Un encapsulateur a pour rôle de filtrer les requêtes et les réponses redondantes. Un messager a pour rôle de transmettre les requêtes et les réponses.

Une stratégie de duplication est mise en œuvre à l'aide d'une classe d'encapsulateurs et d'une classe de messagers. Ces classes permettent de dupliquer des objets selon la stratégie considérée tout en respectant le schéma de communication défini par la notion d'invocation. En d'autres termes, les classes d'objets communication (i.e. encapsulateurs et messagers) réalisent l'encapsulation de la duplication.

[1] Il s'agit de la duplication active, passive, semi-active et coordinateur-cohorte.

[2] Ce logo fait référence à l'expression "n to m" qui caractérise la communication entre n copies d'un client dupliqué et m copies d'un serveur dupliqué.

1.4 Organisation

- Le chapitre 2 présente la notion de duplication et montre comment elle a été utilisée par d'autres auteurs pour réaliser la tolérance aux fautes dans un système réparti.
- Le chapitre 3 présente la notion d'objets dupliqués et montre son utilisation dans le contexte de travaux décrits dans la littérature.
- Le chapitre 4 analyse les problèmes posés par l'invocation entre objets dupliqués et propose une modélisation de ces problèmes.
- Le chapitre 5 décrit N2M, un service d'invocation pour objets dupliqués intégrant les solutions envisagées pour résoudre les problèmes décrits au chapitre précédent.
- Le chapitre 6 présente la mise en œuvre de N2M dans le contexte de l'environnement GARF.
- Le chapitre 7 conclut ce rapport en soulignant les perspectives futures de ce travail.

Chapitre 2

La duplication dans les systèmes répartis

2.1 Introduction

Ce chapitre montre comment la *duplication* est utilisée dans les *systèmes répartis* pour mettre en œuvre la *tolérance aux fautes*. Son objectif est de rappeler les principaux résultats publiés dans la littérature concernant la duplication.

Présentation générale

Intuitivement, la tolérance aux fautes d'un système informatique est sa capacité à continuer à fonctionner malgré les défaillances matérielles et logicielles[1] pouvant affecter les composants du système. Une méthode efficace pour réaliser la tolérance aux fautes consiste à introduire de la redondance dans le système. Certains composants matériels et logiciels sont alors dupliqués: ils existent en plusieurs exemplaires dans le système.

Un système réparti est constitué par un ensemble de processus (ou composants logiciels) s'exécutant sur des ordinateurs distincts et interconnectés par un réseau de communication. Dans un système réparti tolérant aux fautes, la duplication concerne essentiellement le logiciel. Les composants logiciels critiques du système sont dupliqués: ils existent en plusieurs exemplaires et chaque exemplaire d'un même composant s'exécute sur un ordinateur distinct. Par ce biais, on cherche à augmenter la probabilité qu'un composant soit opérationnel malgré les défaillances.

La mise en œuvre de la duplication dans un système réparti s'appuie sur des mécanismes logiciels. Ces derniers sont basés sur des algorithmes complexes qui visent deux objectifs antagonistes: améliorer la disponibilité du système (propriété de vivacité) tout en garantissant sa cohérence (propriété de sûreté). La difficulté principale de ces algorithmes réside dans l'impossibilité d'observer un état global du système. Cette impossibilité provient de l'incertitude introduite par l'existence de délais de communication de durées imprévisibles, et par l'occurence de défaillances. Informellement, cette incertitude correspond à ne pas pouvoir distinguer un composant logiciel défaillant qui n'est plus en mesure de répondre à une requête, d'un composant opérationnel dont la réponse est encore en transit sur le réseau. Construire un état global dans le contexte de la

[1] La cause d'une défaillance est une faute (c.-à-d. un "bug", une panne matérielle, ...). Des définitions plus rigoureuses sont données dans la suite du chapitre.

duplication, correspondrait, par exemple, à établir la liste des copies opérationnelles de chaque composant, ainsi que la liste des requêtes reçues par chaque copie.

Malgré ces difficultés, les systèmes répartis sont adaptés à la construction de systèmes informatiques tolérants aux fautes. La redondance matérielle intrinsèque d'un système réparti fournit un support adéquat à la duplication de composants logiciels. En effet, la défaillance d'un ordinateur (ou d'une ligne de communication) d'un système réparti n'entraine pas la défaillance de tout le système. Cette sémantique de défaillance *partielle* est un avantage certain des systèmes répartis sur les systèmes centralisés. Pour ces derniers, la défaillance de l'unique ordinateur du système est fatale. Cet avantage est renforcé si les ordinateurs composant le système réparti sont éloignés géographiquement: le système peut alors survivre à des catastrophes majeures entraînant la destruction physique d'un ordinateur du système. Cependant, la sémantique de défaillance partielle doit être explicitement prise en compte lors de la conception des composants logiciels d'un système réparti. Un composant logiciel doit savoir réagir correctement à la défaillance d'un autre composant du système. En d'autres termes, tolérance aux fautes et répartition sont intimement liées: d'une part, un système tolérant aux fautes doit être réparti; d'autre part, un système réparti doit inclure des mécanismes de tolérance aux fautes.

La duplication a aussi été utilisée dans certains systèmes centralisés pour assurer la tolérance aux fautes. Dans ces systèmes, la duplication est essentiellement matérielle. Les composants matériels du système (processeur, mémoire centrale, disque, bus, ...) existent en plusieurs exemplaires. Une électronique spécialisée détecte la défaillance éventuelle d'un composant matériel, remplace automatiquement le composant défaillant par l'une de ses copies et le déconnecte du reste du système. Depuis 1976, l'entreprise Tandem domine[2] le marché des systèmes centralisés tolérants aux fautes, notamment grâce au succès du système Tandem NonStop [Bartlett 87].

L'utilisation de mécanismes logiciels pour mettre en œuvre la duplication garantit l'indépendance de ces mécanismes par rapport au matériel. Cette indépendance présente de nombreux avantages. Premièrement, il n'est pas nécessaire de disposer de matériel spécialisé souvent très coûteux. Il suffit de construire le système en utilisant du matériel "classique" dont le prix ne cesse de baisser. Par ailleurs, les ordinateurs composant le système ne doivent pas être nécessairement identiques (hétérogénéité). Enfin, l'évolution inévitable du matériel dans le temps ne demande pas systématiquement de concevoir à nouveau les mécanismes logiciels réalisant la duplication.

Organisation

La section 2.2 présente la notion de tolérance aux fautes. La section 2.3 montre comment la duplication permet d'assurer la tolérance aux fautes dans un système réparti. La section 2.4 décrit l'intégration de la duplication dans le modèle client-serveur. La section 2.5 présente brièvement les quatre stratégies de duplication qui seront étudiées dans les chapitres suivants. La section 2.6 donne quelques exemples d'utilisation de la duplication dans les systèmes répartis. La section 2.7 conclut ce chapitre.

2.2 Tolérance aux fautes

Cette section définit la notion de tolérance aux fautes, en souligne les enjeux et présente les différentes techniques (notamment la duplication) permettant d'assurer la tolérance aux fautes.

[2]Selon [OFTA 94], Tandem détenait 70% du marché européen en 1989!

2.2.1 Définitions

La terminologie présentée dans cette section est inspirée de [Powell 91a, Laprie 92, OFTA 94]. La traduction anglaise de chaque terme défini est donnée entre parenthèses pour permettre au lecteur de faire le lien avec la terminologie utilisée dans la communauté anglophone.

La tolérance aux fautes s'inscrit dans le contexte plus large de la sûreté de fonctionnement. La **sûreté de fonctionnement** (*dependability*) d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre. Le service délivré par un système est son comportement tel qu'il est perçu par ses utilisateurs. Un utilisateur est un autre système (humain ou physique) qui interagit avec le système considéré.

Attributs de la sûreté de fonctionnement

La sûreté de fonctionnement d'un système peut être abordée sous des angles différents, selon les fonctions que remplit le système et selon le domaine d'applications auquel il est destiné. Ces points de vue correspondent à des attributs de la sûreté de fonctionnement, sur lesquels les concepteurs du système peuvent mettre plus ou moins l'accent. Ces attributs sont:

- la **disponibilité** (*availability*) qui définit le fait d'être prêt à l'utilisation;
- la **fiabilité** (*reliability*) qui définit la continuité du service;
- la **sécurité** (*safety*) qui définit la non-occurence de défaillances catastrophiques;
- la **confidentialité** (*privacy*) qui définit l'évitement de divulgations non autorisées de l'information;
- l'**intégrité** (*integrity*) qui définit l'évitement d'altérations non désirées de l'information;
- la **maintenabilité** (*maintenability*) qui définit l'aptitude aux réparations et aux évolutions.

Les attributs de la sûreté de fonctionnement permettent d'exprimer les propriétés attendues du système et d'apprécier la qualité du service délivré. Cette qualité est déterminée par l'efficacité des moyens assurant la sûreté de fonctionnement sur les entraves à la sûreté de fonctionnement.

Entraves à la sûreté de fonctionnement

Quelque soit l'attribut considéré, mettre en œuvre la sûreté de fonctionnement d'un système correspond à lutter contre les défaillances du système. Une **défaillance** (*failure*) survient lorsque le service délivré par le système ne correspond plus à sa spécification. La spécification du service correspond à la description du service que les utilisateurs sont en mesure d'attendre du système. Une **erreur** (*error*) est une anomalie affectant l'état du système et suceptible d'entraîner une défaillance. En d'autres termes, une erreur ne provoque pas systématiquement une défaillance: le système peut continuer à délivrer un service correct malgré un certain nombre d'erreurs affectant son état. Une **faute** (*fault*) est la cause d'une erreur. En résumé, une faute est un événement provoquant une erreur: une altération de l'état du système. Une erreur peut provoquer une défaillance: une altération du service délivré par le système.

La distinction entre faute, erreur et défaillance n'est pas absolue. Elle dépend principalement de la position du point d'observation par rapport aux frontières du système observé. Par exemple, la défaillance d'un sous-système constitue une faute pour le système englobant. La figure 2.1

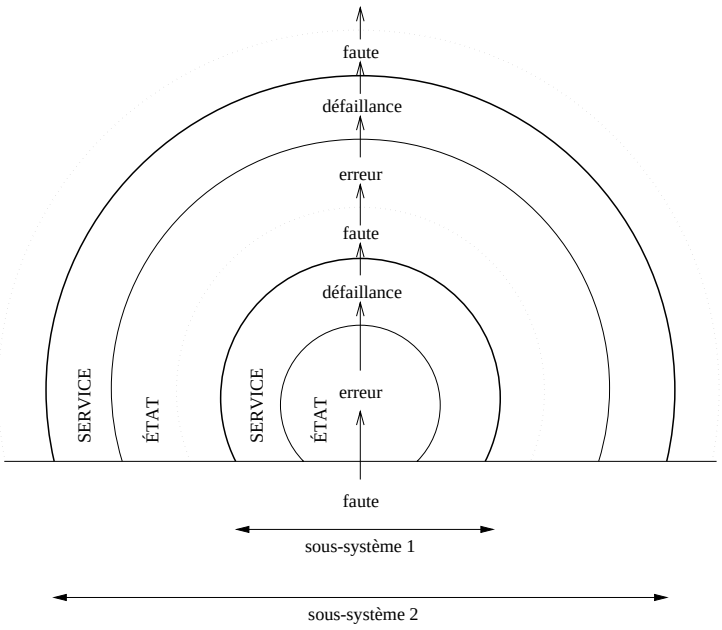

Figure 2.1: Faute, erreur et défaillance

illustre cet exemple. Elle représente un système constitué par plusieurs sous-systèmes imbriqués. Chaque sous-système est composé de deux couches schématisées par deux demi-cercles concentriques: une couche inférieure représentant l'état du sous-système (demi-cercle en trait fin) et une couche supérieure représentant le service fourni par le sous-système (demi-cercle en trait épais). De plus, une couche d'interface (demi-cercle en pointillé) sépare un sous-système du sous-système englobant. Sur la figure 2.1, deux sous-systèmes sont représentés. Une erreur dans l'état du sous-système 1 entraîne une défaillance du service fourni par ce sous-système. Cette défaillance provoque une faute dans la couche d'interface. Cette faute entraîne une erreur dans l'état du sous-système 2 qui provoque à son tour une défaillance du service de ce sous-système. La couche d'interface est en réalité une séparation fictive introduite pour les besoins de l'exposé. Elle permet d'illustrer la relativité des points de vue: la défaillance du sous-système 1 est perçue comme une faute par le sous-système 2.

Les défaillances pouvant affecter un système informatique sont variées. L'élaboration d'une stratégie efficace de lutte contre les défaillances nécessite une caractérisation précise des défaillances à combattre. Dans ce but, plusieurs auteurs comme [Cristian 91, OFTA 94] ont identifié des classes de défaillances. Cette classification s'appuie sur les deux dimensions qui fondent la spécification du service délivré par un système informatique: le *temps* et la *valeur*. La dimension temps permet de spécifier des conditions temporelles (par exemple un délai) sur la délivrance du service. La dimension valeur permet de spécifier des conditions sur la valeur de la réponse résultant de la délivrance du service.

La figure 2.2 présente les classes de défaillances les plus fréquentes. Lorsque la valeur du service

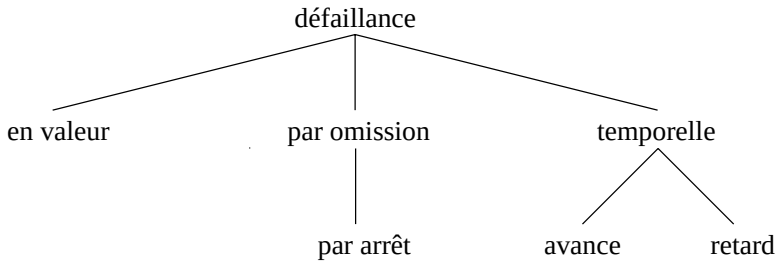

Figure 2.2: Classes de défaillances

délivré ne satisfait plus les conditions de la spécification, le système exhibe une défaillance **en valeur** (*value failure*). Lorsque les conditions temporelles ne sont plus satisfaites (le système répond trop vite (avance) ou trop tard (retard)), le système exhibe une défaillance **temporelle** (*timing failure*). Lorsque le système omet de délivrer le service demandé, il exhibe une défaillance **par omission** (*omission failure*). Si l'omission devient permanente (le système ne répond plus), le système exhibe une défaillance **par arrêt** (*crash failure*). Si le système exhibe chacun de ces comportements de façon imprévisible, il est affecté par des défaillances **arbitraires** (*arbitrary failures*) ou **byzantines** (*byzantine failures*).

Les défaillances sont causées par des fautes. La multitude des fautes potentielles rend difficile l'identification précise de l'origine d'une défaillance. Certains auteurs ont défini des critères de classification des fautes. Ces critères sont la nature, l'origine et la persistance temporelle d'une faute. La **nature** (*nature*) d'une faute permet de distinguer les fautes accidentelles des fautes intentionnelles. L'**origine** (*origin*) d'une faute permet de distinguer les fautes physiques des fautes humaines, les fautes internes au système des fautes externes, et les fautes de conception des fautes opérationnelles survenant lors de l'exploitation du système. Enfin, la **persistance temporelle** (*temporal persistence*) permet de distinguer les fautes permanentes des fautes temporaires.

Moyens permettant d'assurer la sûreté de fonctionnement

Plusieurs moyens sont généralement combinés pour mettre en œuvre la sûreté de fonctionnement d'un système:

- la **prévention des fautes** (*fault prevention*) consiste à empêcher l'occurence de fautes;
- la **tolérance aux fautes** (*fault tolerance*) consiste à délivrer un service correct en dépit de l'occurence de fautes;
- l'**élimination des fautes** (*fault removal*) consiste à réduire le nombre et la sévérité des fautes dans le but de les éliminer du système;
- la **prévision des fautes** (*fault forecasting*) consiste à estimer le nombre de fautes courantes et futures ainsi que leurs conséquences.

La suite de cette thèse se concentre sur l'*un* de ces moyens: la tolérance aux fautes.

2.2.2 Enjeux de la tolérance aux fautes

La tolérance aux fautes est loin d'être une préoccupation marginale. Elle correspond à une réalité économique certaine. Quelques chiffres tirés de [OFTA 94] permettent de s'en convaincre. En France, les défaillances informatiques coûtent annuellement plus de 2,5 milliards de francs suisses, soit près de 5% du chiffre d'affaires de toute l'industrie informatique française. Aux Etats-Unis, les défaillances causées uniquement par des fautes accidentelles (par opposition aux fautes intentionelles dues à la piraterie informatique) coûtent annuellement environ 4,5 milliards de francs suisses. Selon une étude commandée en 1993 par la société américaine Stratus, le coût moyen d'une défaillance d'un système transactionnel est de 500 000 francs suisses! Dans le milieu boursier et bancaire, une défaillance de ce type coûte en moyenne, 220 000 francs suisses à l'heure!

Pourtant, l'utilisation de techniques visant à assurer la tolérance aux fautes est loin d'être généralisée. En 1991, l'informatique tolérante aux fautes ne représentait que 4% du marché informatique européen. Mais à la même date, cette part du marché dépassait les 50% de croissance annuelle. A l'échelle mondiale, la vente de systèmes informatiques tolérants aux fautes totalisait 2.25 milliards de francs suisses sur les 45 milliards de francs suisses que représentaient les ventes mondiales de systèmes informatiques, soit environ 5%. Les constructeurs de systèmes informatiques tolérants aux fautes sont encore peu nombreux. Quatre entreprises se partagent inéquitablement le marché mondial: Tandem domine nettement avec 63% des ventes, suivi de Stratus avec 21%, de DEC avec 10% et de Sequoia avec 6%.

Outre ces considérations économiques, la généralisation de l'informatique à quasiment tous les secteurs de la société contemporaine rend encore plus urgent la nécessité de pouvoir compter sur les systèmes informatiques. Les défaillances des systèmes informatiques peuvent conduire à de véritables catastrophes surtout dans des domaines sensibles comme le transport aérien ou le contrôle de processus industriels. Il est vrai que dans ces domaines des techniques de tolérance aux fautes sont utilisées depuis de nombreuses années déjà.

Cependant, l'évolution actuelle des systèmes informatiques voit la généralisation d'une informatique répartie pour laquelle la tolérance aux fautes est encore loin d'être complètement maîtrisée. De nombreux résultats de recherche ont été publiés, mais jusqu'à présent, peu d'entre eux ont été utilisés pour construire des systèmes informatiques réels (par opposition aux prototypes de recherche). Pourtant, la tolérance aux fautes est un besoin intrinsèque pour les systèmes répartis. La multiplicité des composants d'un système réparti accroit la probabilité qu'un composant défaille. A contrario, la répartition du système augmente la probabilité que le système continue à être opérationel malgré la défaillance de l'un de ses composants.

2.2.3 Techniques de tolérance aux fautes

La tolérance aux fautes est mise en œuvre par la combinaison de deux techniques:

- le **traitement de faute** (*fault treatment*) qui vise à éviter qu'une faute survenue ne se reproduise;

- le **traitement d'erreur** (*error processing*) qui vise à éliminer une erreur avant qu'elle ne provoque une défaillance.

Traitement de faute

L'objectif du traitement de faute est d'éviter qu'une faute survenue ne se reproduise. Pour celà, on procède au **diagnostic de faute** (*fault diagnosis*) qui vise à identifier précisément la faute (nature, origine, composants responsables,...). Puis, on procède à la **passivation** (*fault passivation*) de la faute qui consiste généralement à neutraliser les composants fautifs en les excluant du système (reconfiguration).

Traitement d'erreur

L'objectif du traitement d'erreur est d'éliminer une erreur affectant le système afin d'éviter qu'elle n'entraîne une défaillance. Le traitement d'erreur peut revêtir deux formes:

- le **recouvrement d'erreur** (*error recovery*) qui consiste à remplacer l'état erroné du système par un état correct;
- la **compensation d'erreur** (*error compensation*) qui consiste à compter sur la redondance présente dans le système pour que celui-ci continue à délivrer un service correct malgré un état erroné.

Recouvrement d'erreur Pour réaliser le recouvrement d'erreur, deux méthodes sont possibles:

- la **reprise** (*backward recovery*) qui consiste à remplacer l'état erroné par un état correct dans lequel le système était avant l'occurence de l'erreur;
- la **poursuite** (*forward recovery*) qui consiste à remplacer l'état erroné par un nouvel état correct construit à partir de l'état erroné.

La reprise demande d'établir au préalable des sauvegardes régulières de l'état du système appelées **points de reprise** (*recovery points*). Pour transformer un état erroné en un état correct, on réinitialise l'état du système à partir du dernier point de reprise. La poursuite consiste à construire un nouvel état correct en appliquant sur l'état erroné des traitements spécifiques visant à annuler ou à minimiser la partie erronée de l'état du système. On permet ainsi au système de continuer à délivrer son service, éventuellement en mode dégradé: certaines fonctions du système non vitales ne sont alors plus assurées.

Compensation d'erreur La compensation d'erreur s'appuie sur la redondance présente dans le système. Elle peut prendre deux formes:

- la **détection et compensation d'erreur** (*error detection and compensation*);
- le **masquage d'erreur** (*error masking*).

La détection et compensation d'erreur consiste à ne réagir qu'après avoir détecté une erreur. Le composant erroné est alors remplacé par un composant correct équivalent. Par contre, le masquage d'erreur consiste en une compensation d'erreur systématique, effectuée en permanence, même en l'absence d'erreur. Plusieurs composants réalisent systématiquement la même fonction de façon à ce qu'une erreur provoquant la défaillance de l'un d'entre eux soit sans conséquence sur la disponibilité de la fonction concernée.

2.3 Tolérance aux fautes dans un système réparti

Cette section décrit la tolérance aux fautes dans un système réparti. La section 2.3.1 présente la notion de système réparti. La section 2.3.2 décrit les défaillances les plus souvent considérées dans un système réparti. La section 2.3.3 présente les mécanismes de base sur lesquels s'appuient les algorithmes réalisant la tolérance aux fautes. Enfin, la section 2.3.4 montre comment la duplication permet d'atteindre la tolérance aux fautes dans un système réparti.

2.3.1 Modélisation d'un système réparti

La modélisation d'un **système réparti** (*distributed system*) s'articule autour de la définition de trois niveaux de modélisation: le modèle physique, le modèle logique et le modèle temporel.

Modèle physique

Le **modèle physique** (*physical model*) d'un système réparti décrit les composants physiques du système. Il consiste en un ensemble d'ordinateurs autonomes (appelés aussi **nœuds** (*nodes*)), géographiquement dispersés et interconnectés par un réseau de lignes de communication. Chaque nœud est composé essentiellement par un processeur, une horloge, une mémoire centrale (volatile), une mémoire secondaire (non volatile) et une interface raccordée au réseau de communication. L'ensemble est coordonné par logiciel (système d'exploitation intégrant une couche de communication). Les composants d'un nœud sont utilisés exclusivement par le nœud auxquels ils appartiennent. En d'autres termes, les nœuds ne partagent *aucun* composant et ils communiquent *exclusivement* à travers le réseau de communication.

Par conséquent, il n'existe pas d'état global qui soit stocké dans une zone mémoire accessible à tous les nœuds. Il n'existe pas non plus d'horloge commune qui puisse délivrer un temps global permettant de dater les phases de l'exécution de tous les processeurs du système.

Modèle logique

Le **modèle logique** (*logical model*) décrit un système réparti du point de vue des programmes qui s'exécutent sur le système. Ce point de vue est aussi appelé modèle d'exécution. Le modèle logique consiste en un ensemble de processus s'exécutant de manière concurrente et coopérant pour réaliser une tâche commune. Un **processus** (*process*) représente l'exécution d'un programme sur un nœud du système. C'est une entité active dont le comportement est défini par le code du programme et dont l'état est défini à tout instant par le contenu de la mémoire correspondant aux variables du programme. Les processus coopèrent en échangeant des messages. Un **message** (*message*) est un ensemble d'informations qu'un processus désire communiquer à un ou plusieurs processus du système. Cet échange de messages permet aux processus de prendre des décisions concernant l'état (décentralisé) du système. Un message est transmis d'un processus à l'autre par le biais d'un **canal de communication** (*communication channel*). Un canal représente une connexion logique entre deux processus. La notion de canal permet de s'abstraire de la topologie du réseau de communication. En d'autres termes, tant que la communication est possible entre un nœud n_1 et un nœud n_2, il existe un canal entre tout processus s'exécutant sur n_1 et tout processus s'exécutant sur n_2.

Modèle temporel

Le **modèle temporel** (*timing model*) décrit les hypothèses temporelles sur les délais de communication et sur les délais de traitement dans un système réparti. On distingue généralement deux approches: l'approche synchrone et l'approche asynchrone. L'approche **synchrone** (*synchronous systems*) consiste à supposer que [Hadzilacos 93]:

- il existe une borne supérieure connue au délai de transmission d'un message. Ce délai comprend le temps nécessaire à l'émission, la transmission et la réception d'un message.
- Chaque processus a une horloge logique et la dérive de cette horloge par rapport au temps réel a une borne supérieure connue.
- Il existe une borne inférieure et une borne supérieure connues au temps nécessaire à un processus pour exécuter une instruction de son programme.

L'existence de ces bornes permettent de définir la notion de **délai de garde** (*timeout*) qui correspond au délai maximum au bout duquel un message doit être acquitté par le destinataire. Si au terme du délai de garde, l'acquittement ne parvient pas à l'expéditeur, une défaillance du destinataire ou du réseau est survenue.

L'approche **asynchrone** (*asynchronous systems*) ne fait *aucune* hypothèse temporelle. Aucune des bornes décrites précédemment n'est supposée exister. Cette approche est plus générale et plus réaliste. En pratique, les variations de la charge des nœuds et du réseau rendent difficiles l'établissement de bornes temporelles. Au meilleur des cas, ces bornes peuvent être fixées de manière probabiliste.

À mi-chemin entre l'approche synchrone et l'approche asynchrone, de nombreux autres modèles temporels ont été définis dans la littérature. Par exemple, les auteurs de [Dwork 88] considèrent que le délai de transmission d'un message est borné, mais que la valeur de cette borne est inconnue.

2.3.2 Défaillances dans un système réparti

Le modèle logique d'un système réparti intègre la prise en compte des défaillances. Un processus, comme un canal, peut défaillir à tout moment. La défaillance d'un canal ou d'un processus modélise une défaillance affectant un nœud ou une ligne du réseau. Cette défaillance d'un composant physique est causée par une faute matérielle ou logicielle.

Une défaillance de processus ou de canal est caractérisée uniquement par sa classe (voir section 2.2.1), indépendemment de la faute (pas toujours facile à identifier) ayant causé la défaillance. En outre, le caractère décentralisé d'un système réparti permet de supposer que les composants du système défaillent indépendemment les uns des autres. Cette hypothèse est valable aussi bien pour le modèle physique que pour le modèle logique.

Afin de clarifier la notion de défaillance pour un processus et pour un canal, il est utile de définir précisément, pour chaque classe de défaillances, la manifestation d'un comportement défaillant. Un processus exhibe:

- une défaillance en valeur si une valeur calculée au cours de son exécution n'appartient plus au domaine des valeurs attendues;

- une défaillance par arrêt s'il s'arrête prématurément de façon définitive;
- une défaillance par omission s'il omet d'envoyer ou de recevoir un message;
- une défaillance temporelle s'il ne respecte pas les bornes temporelles sur son temps d'exécution;
- un défaillance arbitraire s'il a un comportement imprévisible (par ex. action non prévue par le programme).

Un canal exhibe:

- une défaillance par arrêt s'il cesse définitivement de transmettre les messages qui lui sont confiés;
- une défaillance par omission s'il omet de transporter un message;
- une défaillance temporelle si le temps de transmission d'un message ne respecte pas les bornes temporelles;
- une défaillance arbitraire s'il a un comportement imprévisible (par ex. génération d'un message erroné).

Une défaillance par arrêt est un cas particulier de défaillance par omission. De même, toute défaillance par omission peut être vue comme un cas particulier de défaillance temporelle. En outre, les défaillances temporelles n'ont de sens que dans le contexte d'une approche synchrone.

2.3.3 Mécanismes de base pour la tolérance aux fautes

Cette section présente quelques mécanismes particulièrement utiles pour la mise en œuvre de la tolérance aux fautes dans un système réparti.

Détecteur de défaillances

Un système réparti tolérant aux fautes comprend un mécanisme chargé de détecter les défaillances des processus et des canaux. Ce mécanisme diffère généralement d'un système réparti à l'autre selon le modèle temporel du système et les classes de défaillances considérées.

Approche synchrone Dans un système synchrone, les défaillances sont détectées à l'aide d'un délai de garde. Lorsqu'un processus p_1 envoie un message m à un processus p_2, le processus p_1 s'attend à recevoir, de la part de p_2, une confirmation de la réception de m, avant l'écoulement d'un délai δ. Si p_1 ne reçoit pas cette confirmation alors que δ est écoulé, une défaillance temporelle est survenue. Cette information n'est pas exploitable sans hypothèses plus restrictives sur les classes de défaillances considérées pour chaque composant logique du système (i.e. processus et canal). Dans l'exemple précédent, la défaillance temporelle peut être aussi bien une véritable défaillance temporelle (le canal ou le processus p_2 n'a pas respecté le délai de traitement), une défaillance par omission (le canal ou le processus p_2 a perdu le message m), ou une défaillance par arrêt (le canal ou le processus p_2 s'est arrêté prématurément). Par conséquent, si l'on suppose que les seules défaillances considérées sont des défaillances par arrêt

affectant les processus, p_1 interprète la défaillance qu'il détecte comme l'arrêt prématuré du processus p_2.

Approche asynchrone Dans un système asynchrone, il n'est pas possible de détecter les défaillances avec un délai de garde, même en ne considérant que les défaillances par arrêt de processus. Dans l'exemple précédent, comme les délais de communication ne sont pas bornés, il est impossible de distinguer le cas où p_2 est défaillant, du cas où la confirmation envoyée par p_2 est encore en transit sur le canal. Pour contourner cette impossibilité, des auteurs [Chandra 94] ont défini le concept de détecteur de défaillances.

Un détecteur de défaillances est chargé d'informer les processus opérationnels (appelés aussi **processus corrects** (*correct processes*)) des défaillances de leurs pairs. Les processus sont supposés n'être affectés que par des défaillances par arrêt. Les canaux sont supposés **fiables** (*reliable channels*): lorsque p_1 envoie un message m à p_2, m finit toujours par arriver à destination sauf si p_1 ou p_2 défaille. Cette hypothèse revient à considérer que les canaux peuvent perdre des messages (défaillance par omission) mais qu'un nombre fini de retransmissions du message suffit à le faire parvenir à destination. Une défaillance par omission peut être causée aussi bien par une surcharge du réseau de communication que par une rupture physique d'une ligne du réseau (on suppose dans ce cas, que la ligne est réparée au bout d'un temps fini).

En pratique, un **détecteur de défaillances** (*failure detector*) est un service réparti sur tous les nœuds du système. Sur chaque nœud, un **module du détecteur de défaillances** (*failure detector module*) informe les processus corrects s'exécutant sur ce nœud, de la liste des processus du système qu'il *suspecte* être défaillants. En effet, l'impossibilité décrite précédemment conduit chaque module du détecteur de défaillances à faire des détections (ou suspicions) erronées. Un module peut suspecter un processus p d'être défaillant alors que p est correct. Les suspicions ne sont pas définitives: un processus p suspecté à un instant donné, peut être ultérieurement retiré de la liste de processus suspectés par le module. Chaque module gère indépendamment sa liste de processus suspectés. Par conséquent, à un instant donné, un processus p peut être suspecté par le module d'un nœud n_1 et considéré comme correct par celui d'un nœud n_2.

Les erreurs du détecteur de défaillances ne doivent en aucun cas perturber le bon fonctionnement des processus corrects du système. Dans ce but, le comportement d'un détecteur de défaillances doit être précisément connu par les algorithmes exécutés par les processus du système. Les auteurs de [Chandra 94] ont défini des propriétés abstraites permettant de caractériser le comportement d'un détecteur de défaillances, indépendemment de toute mise en œuvre. Ces propriétés sont des propriétés de complétude et des propriétés de justesse. La complétude décrit l'aptitude du détecteur à finalement suspecter tous les processus défaillants, tandis que la justesse restreint le nombre de suspicions erronées. Les auteurs de [Chandra 94] ont défini deux propriétés de complétude et quatre propriétés de justesse:

- **complétude** (*completeness*)
  - complétude **forte** (*strong completeness*): il existe un instant à partir duquel tout processus défaillant est définitivement suspecté par *tout* processus correct;
  - complétude **faible** (*weak completeness*): il existe un instant à partir duquel tout processus défaillant est définitivement suspecté par *au moins* un processus correct;
- **justesse** (*accuracy*)

- justesse **forte** (*strong accuracy*): aucun processus correct n'est jamais suspecté;
- justesse **faible** (*weak accuracy*): il existe *au moins* un processus correct qui n'est jamais suspecté;
- justesse **finalement forte** (*eventual strong accuracy*): il existe un instant à partir duquel tout processus correct n'est suspecté par aucun processus correct;
- justesse **finalement faible** (*eventual weak accuracy*): il existe un instant à partir duquel *au moins* un processus correct n'est suspecté par aucun processus correct;

Ces propriétés permettent de définir huit classes de détecteurs de défaillances qui sont résumées par le tableau 2.1. Les noms[3] des classes sont en italique. Par exemple, un détecteur satisfaisant

complétude	**justesse**			
	forte	faible	finalement forte	finalement faible
forte	*parfait* P	*fort* S	*finalement parfait* $\Diamond P$	*finalement fort* $\Diamond S$
faible	*quasiment parfait* Q	*faible* W	*finalement quasiment parfait* $\Diamond Q$	*finalement faible* $\Diamond W$

Tableau 2.1: Huits classes de détecteurs de défaillances

la complétude forte et la justesse forte appartient à la classe des détecteurs parfaits. Cette classe, notée P, représente le détecteur idéal qui détecte toutes les défaillances (complétude forte) et qui ne fait aucune suspicion erronée (justesse forte).

Le mécanisme de détection par délai de garde dans un système synchrone a une sémantique identique aux détecteurs de la classe P. Par conséquent, la notion de détecteur de défaillances peut être utilisée aussi bien dans un système asynchrone que dans un système synchrone. L'avantage de cette approche est d'exprimer les algorithmes le plus généralement possible, sans faire référence à des hypothèses temporelles. Ces dernières sont alors confinées dans le détecteur de défaillances.

Les auteurs de [Chandra 94] ont démontré qu'il est possible de construire un détecteur D' satisfaisant la complétude forte à partir de tout détecteur D satisfaisant la complétude faible. Par conséquent, tout algorithme utilisant D' peut être utilisé avec D. Ce résultat est important car il montre que pour un problème donné, il suffit d'exhiber un algorithme utilisant un détecteur satisfaisant la complétude forte, pour que cet algorithme puisse fonctionner avec un détecteur satisfaisant la complétude faible.

Groupe de processus

Un **groupe de processus** (*process group*) est un ensemble de processus coopérant pour atteindre un objectif commun. Un groupe est désigné par un nom unique et indépendant de la composition du groupe. Le nom du groupe est fixé au moment de la création du groupe. La composition du groupe est variable. À tout moment, un processus peut quitter ou rejoindre le

[3] Les noms des classes Q et $\Diamond Q$ ont été choisis arbitrairement. Les auteurs de [Chandra 94] n'ont pas donné de noms à ces classes.

groupe. En particulier, lorsqu'un processus appartenant à un groupe se termine (normalement ou prématurément), il quitte le groupe.

A tout moment, les membres d'un groupe g ont tous la même vue de la composition du groupe. La **vue** (*group view*) du groupe g à l'instant logique i, notée $v_i(g)$, est la liste des processus appartenant à g à l'instant i. De plus, tous les membres d'un groupe g reçoivent la *même* séquence de vues. Cette séquence constitue une horloge logique dont les tops sont les changements de vue.

Cette sémantique est assurée par le **service de gestion de groupe** (*group membership service*), un service réparti qui s'appuie sur les informations fournies par le détecteur de défaillances. Comme ce dernier, le service de gestion de groupe est constitué par des modules, localisé chacun sur un nœud. Lorsqu'un module détecte un événement suceptible de provoquer un changement de vue (par ex. un processus désire rejoindre le groupe, un membre du groupe est suspecté), il communique avec ses pairs de façon à établir un consensus sur la vue du groupe. Si une nouvelle vue est construite, elle est délivrée à tous les membres du groupe.

Envoi d'un message à un groupe de processus

La notion de groupe de processus s'avère particulièrement utile pour envoyer un message à plusieurs processus. Le nom de groupe permet d'adresser un message à un groupe sans connaître l'identité et la localisation de chaque destinataire. Une primitive réalisant l'envoi d'un message à un groupe est appelée **multicast** (*multicast*). On distingue généralement plusieurs types de multicasts en fonction de la sémantique offerte par chaque primitive.

Un multicast **fiable** (*reliable multicast*) garantit qu'un message m envoyé à un groupe g est reçu par tous les membres corrects de g ou par aucun. L'inconvénient de cette définition est qu'elle ne précise pas l'instant logique (i.e. la vue) auquel m est délivré aux membres de g.

Un multicast **vue-synchrone** (*view-synchronous multicast*) garantit la propriété suivante: soit un processus p appartenant à $v_i(g)$; si p a délivré m dans $v_i(g)$ avant d'installer $v_{i+1}(g)$, alors tous les processus appartenant à $v_i(g)$ ayant installé $v_{i+1}(g)$ ont délivré m avant d'installer $v_{i+1}(g)$. L'installation d'une vue v_k par un processus q correspond au moment où q considère v_k comme la vue courante du groupe. Il est à noter que q n'installe pas v_k immédiatement après l'avoir reçu. Comme le montre cette définition, q doit parfois attendre de délivrer un ou plusieurs messages avant de pouvoir installer v_k. Le multicast vue-synchrone ordonne un message par rapport aux changements de vue. Cependant, au sein d'une même vue, les messages ne sont pas ordonnés les uns par rapport aux autres.

La sémantique vue-synchrone peut être augmentée avec un ordre de délivrance des messages. Cet ordre est le plus souvent FIFO, causal ou total:

- ordre **FIFO** (*FIFO order*): si un processus p envoie un message m au groupe g avant d'envoyer un message m' au même groupe g, alors tous les processus de g ne délivrent m' qu'après avoir délivré m;
- ordre **causal** (*causal order*): si l'envoi d'un message m à un groupe g précède causalement l'envoi d'un message m' à g, alors alors tous les processus de g ne délivrent m' qu'après avoir délivré m. On dit qu'un évènement e précède causalement un évènement e' (noté $e \rightarrow e'$) si et seulement si:
  - un processus exécute e puis e', ou

  - e correspond à l'envoi d'un message m et e' à la délivrance de m, ou
  - il existe un évènement e'' tel que $e \rightarrow e''$ et $e'' \rightarrow e'$;

- ordre **total** (*total order*): Si un processus p et un processus q appartenant à un groupe g délivrent les messages m et m', alors p et q délivrent m et m' dans le même ordre.

2.3.4 Tolérance aux fautes par duplication dans un système réparti

La **duplication**[4] (*replication*) consiste à introduire *intentionellement* de la redondance dans un système dans le but de tolérer les fautes en utilisant la technique de compensation d'erreur décrite précédemment (voir section 2.2.3).

Dans un système réparti, la duplication concerne les composants logiciels. Le terme **composant logiciel** (*software component*) englobe aussi bien la notion de processus, que la notion d'objet définie au chapitre suivant. L'objectif de cette terminologie est d'aborder la duplication en établissant d'emblée, un lien entre l'approche classique des systèmes répartis et l'approche orientée-objets.

La redondance matérielle intrinsèque d'un système réparti offre un support adéquat à la duplication de composants logiciels. En effet, les copies des composants logiciels du système sont disséminées sur les ordinateurs constituant le système réparti.

Pour mettre en œuvre la duplication, deux étapes préalables sont nécessaires:

- identifier les composants logiciels critiques: ceux qui *doivent* être dupliqués;
- fixer pour chaque composant critique, les trois paramètres contrôlant la duplication du composant: le taux de duplication, le placement des copies du composant et la stratégie de duplication.

Un composant est dit **critique** (*critical*) si la défaillance de ce composant risque d'entraîner une défaillance du système. Identifier les composants critiques répond à un double objectif. Assurer une couverture satisfaisante des risques de défaillances d'une part, et minimiser le coût des mécanismes de tolérance aux fautes, d'autre part. En effet, la duplication induit un certain coût aussi bien en termes de performances (temps de réponse) qu'en termes de dimensionnement (nombre de composants) du système. Pour minimiser ce coût, la duplication ne doit être appliquée qu'aux composants critiques.

Paramétrer la duplication de chaque composant critique permet de contrôler finement la tolérance aux fautes désirée pour le système. Le **taux de duplication** (*replication rate*) d'un composant désigne le nombre de copies du composant qui existent dans le système avant l'occurence de la première faute. Le taux de duplication est déterminé par deux paramètres:

- le nombre de fautes que le composant doit tolérer, c.-à-d. le nombre maximal de copies défaillantes que le composant peut tolérer tout en continuant à délivrer un service correct;
- la classe des défaillances affectant les copies du composant.

[4] D'autres auteurs comme [OFTA 94, Sens 94, Anceaume 93] préfèrent utiliser le terme "réplication" pour désigner la même notion. L'inconvénient de ce substantif est qu'il n'existe pas de verbe construit sur la même racine: le verbe "répliquer" a un tout autre sens en français. Par conséquent, le substantif "duplication" et le verbe "dupliquer" sont utilisés systématiquement dans toute la thèse.

Le **placement des copies** (*replica location*) d'un composant consiste à choisir l'ensemble des nœuds du système sur chacun desquels sera située une copie du composant. Ce choix est déterminé par des facteurs multiples comme la probabilité de défaillance des nœuds, les besoins en ressources de traitement des composants par rapport aux performances relatives des machines, ainsi que les dépendances client-serveur entre les composants logiciels du système[5]. Le troisième paramètre de la duplication d'un composant est la **stratégie de duplication** (*replication policy*) qui décrit l'algorithme chargé de la gestion des copies. Ce dernier paramètre fait l'objet de la section 2.5 et n'est donc pas développé ici.

2.4 Modèle client-serveur et duplication

Cette section décrit l'intégration de la duplication au modèle client-serveur. La section 2.4.1 rappelle quelques définitions concernant ce modèle. La section 2.4.2 présente successivement la duplication des serveurs puis celle des clients.

2.4.1 Définitions

Le modèle **client-serveur** (*client-server model*) permet d'exprimer de manière adéquate les interactions entre les composants logiciels d'un système réparti. Il caractérise ces interactions en termes de dépendance entre des fournisseurs de services (les serveurs) et les usagers de ces services (les clients).

Un service a été défini (cf. section 2.2.1) comme le comportement d'un système, tel que les utilisateurs du système le perçoivent. Plus précisément, un **service** (*service*) constitue la spécification d'un ensemble d'opérations liées par une sémantique commune. Cette spécification décrit la sémantique de chaque opération indépendamment de toute mise en œuvre. Par exemple, un service de désignation décrit toutes les opérations manipulant les noms dans un système, telles que la création d'un nom, la recherche de l'existence d'un nom, etc.

Un **serveur** (*server*) fournissant le service Σ est un composant logiciel réalisant les opérations de Σ. La spécification de chaque opération est décrite dans l'interface du composant de manière à la rendre accessible aux autres composants du système. Par contre, les détails de mise en œuvre des opérations sont cachées dans le corps du composant.

Un **client** (*client*) du service Σ est un usager de Σ. Un client est généralement un composant logiciel mais il peut être aussi un utilisateur humain, par exemple dans le cas où Σ est un service d'interface utilisateur.

Les notions de client et de serveur sont toujours relatives à un service donné. Cependant, il est utile de pouvoir caractériser un composant selon qu'il est client et/ou serveur indépendamment du service utilisé et/ou fourni. Cette remarque amène les définitions suivantes.

On appelle **s-composant** (*s-component*) un composant qui est uniquement serveur: un s-composant n'est client d'aucun service. On appelle **c-composant** (*c-component*) un composant qui est uniquement client: un c-composant n'est serveur d'aucun service. Un **cs-composant** (*cs-component*) est un composant qui est à la fois client d'un service et serveur d'un autre service.

Un client d'un service Σ demande l'exécution d'une opération de Σ à l'aide d'une **requête** (*request*) de service. Une requête précise le nom du serveur et le nom de l'opération con-

[5] Ces dépendances sont décrites à la section 2.4

formément à la spécification décrite par l'interface du serveur. Une requête provoque l'exécution de l'opération correspondante par le serveur. Au terme de l'exécution, la totalité ou une partie du résultat de l'exécution est communiquée au client: cette information constitue la **réponse** (*reply*) à la requête formulée par le client. Toute cette interaction requête-réponse entre un client et un serveur est appelée **invocation** (*invocation*). On peut distinguer trois phases dans une invocation:

1. **transmission de la requête** (*request transmission*): le client transmet la requête au serveur;
2. **traitement de la requête** (*request processing*): le serveur traite la requête et produit une réponse;
3. **transmission de la réponse** (*reply transmission*): le serveur transmet la réponse au client.

Les dépendances client-serveur entre les composants peuvent être exprimées par la relation *utilise*, définie comme suit:

Soient deux composants C et S, $C\ utilise\ S$ si et seulement si il existe un service Σ dont S est serveur et dont C est client. Par abus de langage, on dira que S est serveur de C et que C est client de S.

Le graphe de la relation *utilise* permet de représenter les dépendances client-serveur entre les composants du système. La figure 2.3 donne le graphe de la relation *utilise* pour un système

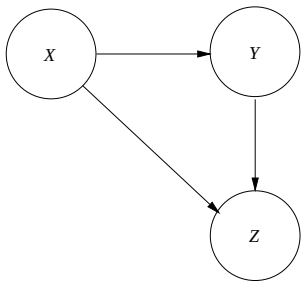

Figure 2.3: Dépendances client-serveur entre les composants du système

très simple comportant trois composants notés X, Y, Z. Les nœuds du graphe représentent les composants et les arcs orientés expriment les dépendances entre les composants. Le nœud X est une racine: il n'a aucun prédécesseur. Cela signifie que X est un c-composant, client de Z. Le nœud Z est une feuille: il n'a aucun successeur. Cela signifie que Z est un s-composant, serveur de X et de Y. Le nœud Y a un prédécesseur (X) et un successeur (Z): le composant Y est un cs-composant, client de Z et serveur de X.

2.4.2 Duplication des clients et des serveurs

Cette section présente, de manière intuitive, les problèmes liés à l'intégration de la duplication au modèle client-serveur. Son objectif est de donner au lecteur une première idée de ces problèmes,

afin qu'il puisse mieux situer les travaux, tirés de la littérature, décrits à la section 2.6 et au chapitre suivant.

Afin de rendre l'exposé plus clair, la présentation des problèmes est progressive. La première étape consiste à ne considèrer que la duplication d'une catégorie particulière de composants serveurs: les s-composants.

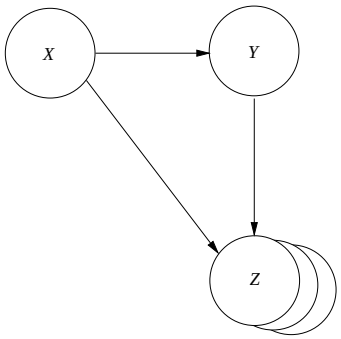

Figure 2.4: Duplication des s-composants

Duplication des s-composants

Cette étape est la plus intuitive et la mieux comprise. La majeure partie des travaux concernant la duplication décrits dans la littérature (voir section 2.6) lui sont consacrés. La figure 2.4 reprend l'exemple de la section précédente en considérant la duplication du s-composant Z en trois exemplaires. Tant qu'il existe une copie de Z non défaillante, le s-composant Z doit continuer à délivrer le ou les services qu'il fournit à ses clients. Par conséquent, la stratégie de duplication de Z assure que l'état de chaque copie du s-composant sont identiques.

La duplication de Z est une caractéristique propre à ce s-composant. À ce titre, elle ne doit pas influer sur la manière dont Z interagit avec ses clients. En d'autres termes, un client doit invoquer Z *comme si* Z n'était pas dupliqué. Pour atteindre cet objectif, il faut cacher au client tout ce qui pourrait révéler la duplication de Z au cours du déroulement de l'invocation. La duplication de Z doit être cachée au client lors des deux phases de communication que comporte une invocation: la transmission de la requête et la transmission de la réponse.

Transmission de la requête Lors de la transmission de la requête, la duplication de Z pose le *problème de la désignation du serveur*. Puisque Z est dupliqué, il existe plusieurs copies de ce s-composant dans le système. Selon la stratégie de duplication de Z, la requête doit être envoyée à une ou plusieurs copies. Le client n'est pas censé connaître la stratégie de duplication de Z, il ne peut donc pas savoir à qui il doit adresser sa requête. De plus, la composition de l'ensemble des copies de Z peut changer à tout moment, suite aux défaillances et aux redémarrages de certaines copies. Par conséquent, le client doit pouvoir désigner dans sa requête l'*ensemble* des copies de Z, et non pas chaque copie individuellement.

Transmission de la réponse Lors de la transmission de la réponse, la duplication de Z pose le *problème du nombre variable de réponses* que Z envoie au client. Ce nombre varie en fonction de la stratégie de duplication de Z. Cependant, conformément à la définition de l'invocation, le client n'attend qu'une seule réponse à sa requête. Par conséquent, il ne sait pas traiter les réponses multiples.

Duplication des s-composants et des cs-composants

La section précédente était consacrée à la duplication des s-composants. Cette section considère également la duplication des cs-composants. Par définition, un cs-composant est à la fois client et

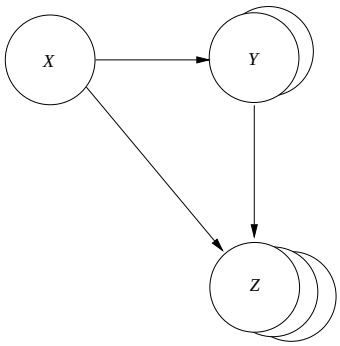

Figure 2.5: Duplication des serveurs

serveur. Les composants dupliqués sont non seulement des serveurs mais aussi des clients. Cette section se focalise donc sur l'impact de la duplication sur le rôle de client d'un cs-composant. Il est important de noter que, contrairement la duplication des s-composants, la duplication des cs-composants a peu été étudiée dans la littérature. Un des objectifs de cette thèse est justement de traiter aussi bien la duplication des clients que celle des serveurs. La figure 2.5 est similaire à la figure 2.4 à la différence près que le cs-composant Y est dupliqué deux fois. Lorsque Y invoque Z, la duplication de Y ne doit pas influer sur le déroulement de l'invocation. Du point de vue de Z, l'invocation doit se dérouler *comme si* Y n'était pas dupliqué. Il est clair que tout ce qui a été dit précédemment concernant la duplication de Z, reste valable. L'objectif visé est de cacher la duplication du client Y au serveur Z. Comme précédemment, la duplication de Y peut être révélée lors de la transmission de la requête et lors de la transmission de la réponse.

Transmission de la requête Lors de la transmission de la requête, la duplication de Y pose le *problème du nombre variable de requêtes* que Y envoie au serveur Z. Ce nombre varie en fonction de la stratégie de duplication de Y. On remarque d'emblée une analogie avec le problème du nombre variable de réponses. Dans les deux cas, une seule requête (respectivement réponse) doit être transmise au serveur (respectivement client). A en juger par le petit nombre de systèmes qui le prennent en compte, ce problème paraît moins évident à résoudre. Pour le comprendre, il faut le situer dans le contexte d'une chaîne d'invocations comme celle de la figure 2.5. Si à la suite d'une invocation de Y par X, plusieurs copies de Y invoquent Z, Z

reçoit plusieurs fois la même requête. Si le traitement correspondant n'est pas idempotent[6] (par exemple, l'incrémentation d'un compteur), l'état du serveur Z risque de devenir incohérent. Comme Z n'est pas conscient de la duplication de Y, il ne sait pas a priori, si les requêtes qu'il reçoit sont des copies de la même requête, ou au contraire, des requêtes distinctes.

Transmission de la réponse Lors de la transmission de la réponse, la duplication de Y pose le *problème de la désignation du client*. On remarque cette fois une analogie avec le problème de la désignation du serveur. Selon la stratégie de duplication de Y, une ou plusieurs copies de Y ont envoyé la requête à Z, et attendent une réponse. Comme Z ignore la stratégie de duplication de Y, il ne peut déterminer l'ensemble des copies de Y auxquelles il doit envoyer une réponse.

Duplication des s-composants, des cs-composants et des c-composants

La duplication des c-composants ne pose pas de nouveaux problèmes. On retrouve les problèmes

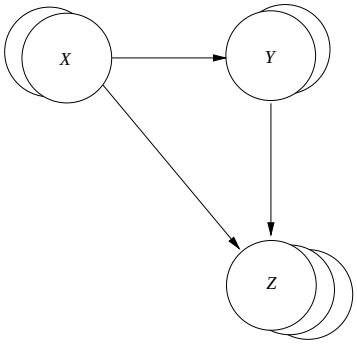

Figure 2.6: Duplication des clients et des serveurs

relatifs à la duplication du client, cités à la section précédente. La figure 2.6 reprend l'exemple des trois composants X, Y, Z, mais cette fois le c-composant X est dupliqué deux fois.

Synthèse

En résumé, l'intégration de la duplication dans le modèle client-serveur pose donc les problèmes suivants:

- lors de la transmission de la requête:
  1. problème de désignation du serveur;
  2. problème du nombre variable de requêtes;
- lors de la transmission de la réponse:

[6] Un traitement est dit *idempotent* si plusieurs exécutions de ce traitement produisent un résultat équivalent à celui d'une seule exécution.

1. problème de désignation du client;
2. problème du nombre variable de réponses.

2.5 Stratégies de duplication

Cette section définit la notion de stratégie de duplication, et présente quatre stratégies représentatives des travaux décrits dans la littérature. Ces stratégies visent à garantir une **cohérence forte** (*strong consistency*) entre les copies d'un composant dupliqué. Informellement, ceci revient à assurer que l'état de chaque copie soit identique. La duplication active (section 2.5.2) et la duplication passive (section 2.5.3) sont véritablement deux stratégies de référence dont s'inspirent la majorité des autres stratégies de duplication. La duplication semi-active (section 2.5.4) et la duplication coordinateur-cohorte (section 2.5.5) sont deux exemples de stratégies hybrides s'inspirant de la duplication active et de la duplication passive.

2.5.1 Notion de stratégie de duplication

Une **stratégie de duplication** (*replication policy*) associé à un composant dupliqué O, définit l'algorithme utilisé pour gérer les copies de O. Cette description comporte plusieurs volets:

- la description des *interactions internes* à O, c.-à-d. les interactions entre les copies de O;
- la description des *interactions externes* à O, c.-à-d. les interactions entre les copies de O et les autres composants du système. Parmi les interactions externes, on peut distinguer:
  - les *interactions client* qui ont lieu lorsque O est client;
  - les *interactions serveur* qui ont lieu lorsque O est serveur.

Dans la littérature, les stratégies de duplication sont décrites le plus souvent pour des s-composants. Les interactions client ne sont quasiment jamais décrites. Pour cette raison, les stratégies présentées dans les sections suivantes sont décrites pour des composants serveurs. Le chapitre 4 revient sur ces stratégies lors de l'analyse des problèmes posés par l'invocation entre objets dupliqués.

2.5.2 Duplication active

La duplication **active** (*active replication* ou *state machine approach*) [Schneider 90, Powell 91b] se caractérise par la symétrie des comportements des copies d'un composant dupliqué. Chaque copie joue un rôle identique à celui des autres.

Principe

La duplication active est définie ainsi:

- réception des requêtes: *toutes les copies reçoivent la même séquence*[7] *de requêtes*;

[7] Une séquence est un ensemble totalement ordonné. Les copies reçoivent les *mêmes* requêtes dans le *même* ordre.

- traitement des requêtes: *toutes les copies traitent les requêtes de manière déterministe*[8];
- émission des réponses: *toutes les copies émettent la même séquence de réponses.*

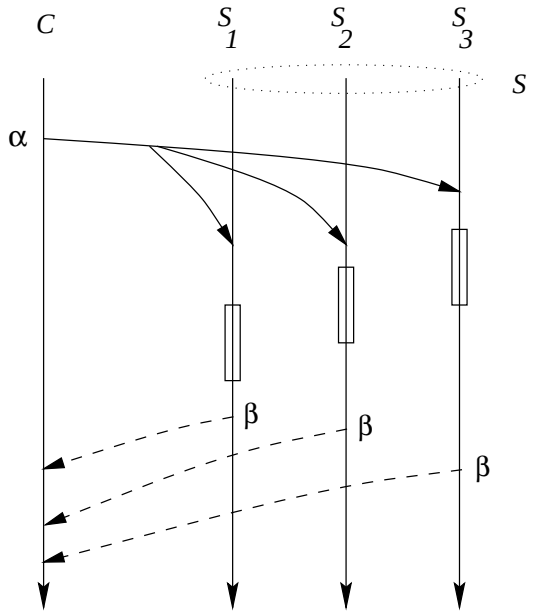

Figure 2.7: Principe de la duplication active

La figure 2.7 illustre ce principe à l'aide d'un diagramme temporel. Les flèches verticales représentent l'exécution de quatre composants: C, S_1, S_2, S_3. Les S_i sont des copies du composant dupliqué S. Elles appartiennent à un groupe (représenté par une ellipse en pointillé). Lorsque C invoque S, il envoie une requête α (flèches en trait continu) à tous les S_i à l'aide d'un multicast fiable assurant l'ordre total. Chaque S_i traite la requête (petit rectangle vertical) et envoie une réponse β (flèche en trait discontinu) à C.

Tolérance aux fautes

La tolérance aux fautes est réalisée par masquage d'erreur. La défaillance d'une copie est masquée par le comportement des copies non défaillantes. Comme chaque copie joue un rôle identique, la défaillance de l'une d'entre elles ne perturbe pas le service fourni par le composant.

2.5.3 Duplication passive

Contrairement à la duplication active, la duplication **passive** (*passive replication* ou *primary-backups replication*) [Budhijara 93, Powell 91b] est asymétrique. Elle distingue deux comportements pour les copies d'un composant dupliqué: la copie **primaire** (*primary copy*) et les copies **secondaires** (*backups*). La copie primaire est la seule à effectuer tous les traitements. Les copies

[8] Un traitement déterministe produit toujours le même résultat à partir des mêmes données. Par conséquent, pour une requête donnée, toutes les copies produisent la même réponse

secondaires, oisives, surveillent la copie primaire. En cas de défaillance de la copie primaire, une copie secondaire devient la nouvelle copie primaire.

Principe

La duplication passive est définie ainsi:

- réception des requêtes: *la copie primaire est la seule à recevoir les requêtes*;
- traitement des requêtes: *la copie primaire est la seule à traiter les requêtes*;
- émission des réponses: *la copie primaire est la seule à émettre les réponses*.

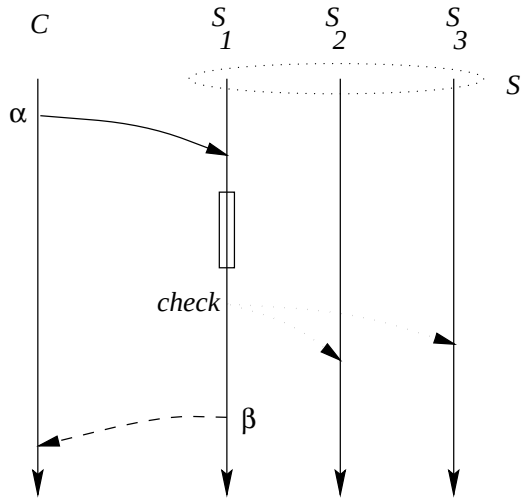

Figure 2.8: Principe de la duplication passive

La figure 2.8 illustre ce principe. Le client C envoie la requête α uniquement à la copie primaire S_1. Celle-ci traite la requête, construit un point de reprise *check* et l'envoie, à l'aide d'un multicast fiable assurant l'ordre FIFO, aux copies secondaires S_2 et S_3 (flèches en pointillé). Le point de reprise contient à la fois la réponse β et le nouvel état de la copie primaire. Puis, la copie primaire S_1 envoie la réponse β à C. Le point de reprise permet de synchroniser l'état des copies secondaires avec celui de la copie primaire puisque celle-ci est la seule qui communique avec le reste du système.

Tolérance aux fautes

La tolérance aux fautes est réalisée par détection et compensation d'erreur. La défaillance d'une copie secondaire ne nécessite aucun traitement particulier: son seul effet est de diminuer le taux de duplication du composant. Par contre, la défaillance de la copie primaire S_1 implique que les copies secondaires désignent l'une d'entre elles (par exemple S_2) comme la nouvelle copie primaire. La défaillance de la copie primaire pendant une invocation a pour conséquence que le client C n'obtient aucune réponse à la requête α. Le client doit alors réémettre sa requête en

l'adressant à la nouvelle copie primaire S_2. Cette dernière doit être en mesure de construire la réponse β que C attend. Deux situations peuvent se présenter selon que la défaillance de S_1 ait été détectée avant ou après la réception de *check* par les copies secondaires.

Si la défaillance de la copie primaire S_1 est détectée *avant* la réception du point de reprise *check*, tout le traitement de α effectué par S_1 est perdu. Lorsque le client réémet α, la nouvelle copie primaire la traite normalement. Par contre, si la défaillance de la copie primaire S_1 est détectée après la réception du point de reprise, la nouvelle copie primaire construit la réponse à partir du point de reprise, et envoie la réponse au client.

La construction d'un point de reprise est faite de façon *systématique* après le traitement d'une requête. Ainsi, lorsque la copie primaire répond au client, l'état de la copie primaire est déjà sauvegardé dans les copies secondaires. Cette approche peut être optimisée en tenant compte de la sémantique des requêtes: si la requête ne modifie pas l'état du composant, il n'est pas nécessaire de construire un point de reprise. Il existe d'autres approches, comme la construction *périodique* de points de reprise qui consiste à construire un point de reprise toutes les n requêtes. Si cette l'approche périodique est moins coûteuse que l'approche systématique, elle est aussi moins sûre, car l'état des copies primaires est toujours "en retard" de plusieurs requêtes (au maximum n) sur celui de la copie primaire. Par conséquent, la défaillance de la copie primaire provoque la perte définitive de cet état.

2.5.4 Duplication semi-active

La duplication **semi-active** (*semi-active replication* ou *leader-followers replication*) [Powell 91b] se situe à mi-chemin entre la duplication active et la duplication passive. Comme cette dernière, la duplication semi-active est une stratégie asymétrique. Contrairement à la duplication passive, les copies secondaires ne sont pas oisives. La copie primaire est appelée *leader* et les copies secondaires sont appelées *suiveurs*.

Principe

Le duplication semi-active est définie ainsi:

- réception des requêtes: *toutes les copies reçoivent le même ensemble*[9] *de requêtes*;
- traitement des requêtes: *toutes les copies traitent toutes les requêtes. La copie primaire traite une requête dès qu'elle la reçoit. Par contre, une copie secondaire doit attendre une notification de la copie primaire pour pouvoir traiter une requête*;
- émission des réponses: *la copie primaire est la seule à émettre les réponses*.

La figure 2.9 illustre ce principe. Le client C envoie la requête α à tous les S_i. La copie primaire S_1 envoie une notification $notify$ aux copies secondaires et commence le traitement de α. Les copies secondaires S_2 et S_3 ne commencent à traiter α qu'après avoir reçu la notification de la copie primaire. Sitôt le traitement terminé, S_1 envoie la réponse β au client C.

[9] Les requêtes ne sont pas totalement ordonnées.

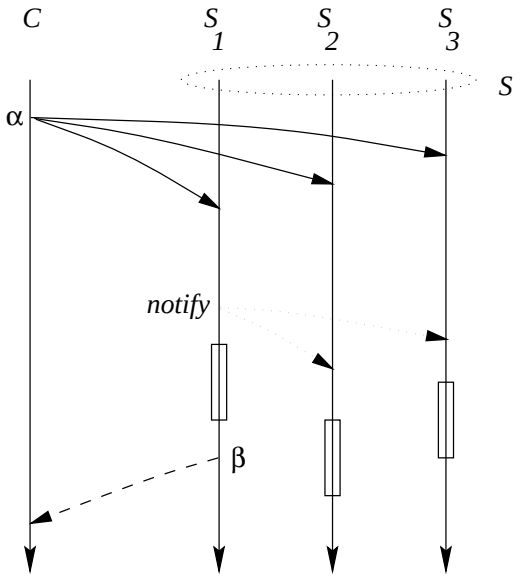

Figure 2.9: Principe de la duplication semi-active

Tolérance aux fautes

La tolérance aux fautes est réalisée par détection et compensation d'erreur, comme dans le cas de la duplication passive. Cependant, comme toutes les copies reçoivent la requête, le client C n'a pas besoin de réémettre sa requête lorsque la copie primaire S_1 défaille. La nouvelle copie primaire S_2 envoie automatiquement la réponse au client, qui risque dans certains cas de recevoir plusieurs fois la même réponse. Comme pour la duplication passive, deux situations peuvent se présenter suivant que la défaillance de la copie primaire est détectée par les copies secondaires avant ou après la réception de la notification.

Si la défaillance de la copie primaire est détectée *avant* la réception de la notification, la nouvelle copie primaire S_2 envoie une notification concernant la première requête présente dans sa queue d'entrée, et la traite normalement. Cette requête peut être aussi bien α qu'une autre requête reçue précédemment pour laquelle S_2 n'a pas reçu de notification. Si la défaillance de S_1 est détectée *après* la réception de la notification, S_2 traite la requête correspondante sans envoyer de notification et envoie la réponse β au client C. Celui-ci a pu déjà recevoir cette réponse de la copie primaire S_1, si celle-ci a défailli après avoir envoyé β.

2.5.5 Duplication coordinateur-cohorte

Comme la duplication semi-active, la duplication **coordinateur-cohorte** (*coordinator-cohort replication*) [Birman 85] est une stratégie hybride entre la duplication active et la duplication passive. Elle distingue une copie primaire appelée *coordinateur*, des copies secondaires appelées *cohorte*.

Principe

La duplication coordinateur-cohorte est définie ainsi:

- réception des requêtes: *toutes les copies reçoivent le même ensemble de requêtes*;
- traitement des requêtes: *la copie primaire est la seule à traiter les requêtes*;
- émission des réponses: *la copie primaire est la seule à émettre les réponses*.

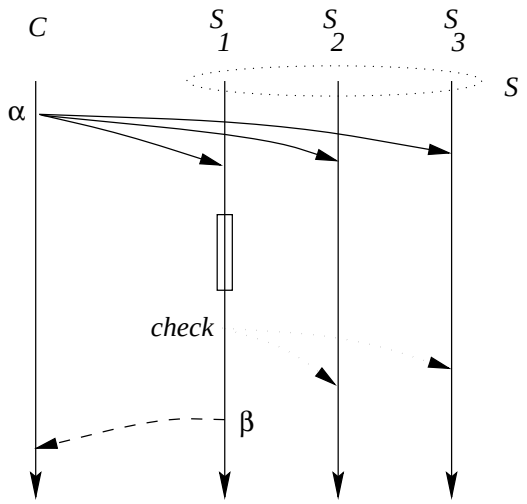

Figure 2.10: Principe de la duplication coordinateur-cohorte

La figure 2.10 illustre ce principe. Le client C envoie la requête α à tous les S_i. La copie primaire S_1 traite la requête et procède exactement comme dans le cas de la duplication passive.

Tolérance aux fautes

La tolérance aux fautes est réalisée par détection et compensation d'erreur. Comme pour la duplication semi-active, le client C n'a pas besoin de réémettre sa requête lorsque la copie primaire S_1 défaille. La nouvelle copie primaire S_2 envoie automatiquement la réponse au client. La duplication coordinateur-cohorte utilise des points de reprise systématiques, comme dans le cas de la duplication passive.

2.6 Exemples de mises en œuvre de la duplication

Cette section donne trois exemples représentatifs de systèmes mettant en œuvre la duplication dans le modèle client-serveur.

2.6.1 ISIS

ISIS [Birman 93] est une boîte à outils logicielle permettant de programmer des applications réparties et tolérantes aux fautes. Ces outils se présentent sous la forme de bibliothèques de "primitives" (i.e. fonctions ou procédures) écrites en langage C. Ces primitives permettent principalement de gérer des groupes de processus UNIX et de réaliser des multicasts vers un groupe de processus avec certaines garanties d'atomicité et d'ordre.

La définition d'un groupe de processus donnée à la section 2.3.3 est largement inspirée des concepts mis en œuvre dans ISIS. Les concepteurs de ISIS ont défini la sémantique vue-synchrone qui consiste à ordonner totalement les vues (séquence de vues) et à ordonner les multicasts par rapport aux vues (multicast vue-synchrones).

La gestion de groupe dans ISIS se base sur *pg_join()*, une primitive sophistiquée qui permet au processus appelant de rejoindre un groupe. Cette primitive permet d'indiquer à ISIS, la procédure à appeler chaque fois que la vue du groupe change. Le programmeur décide ainsi du traitement à effectuer lorsqu'un membre du groupe défaille et lorsqu'un nouveau membre rejoint le groupe. En outre, *pg_join()* permet d'indiquer à ISIS les procédures réalisant le transfert d'état: un nouveau membre doit initialiser son état à partir de celui des membres existants avant de pouvoir être intégré à la vue du groupe.

ISIS met en œuvre trois multicasts vue-synchrones: *fbcast()* garantit l'ordre FIFO, *cbcast()* garantit l'ordre causal et *abcast()* garantit l'ordre total. Pour chaque primitive, le programmeur peut notamment préciser le nombre de réponses attendues. L'appelant est alors bloqué jusqu'à la réception des réponses. Cette sémantique, inspirée de celle des **appels de procédure à distance** (*remote procedure call*) (RPC) [Birrell 84], s'avère très pratique à utiliser.

Les outils fournis par ISIS permettent de mettre en œuvre, avec plus ou moins de difficultés, les quatres stratégies de duplication décrites précédemment. Ils sont surtout adaptés à la duplication des s-composants. L'utilisation de ISIS pour dupliquer des cs-composants est plus complexe mais pas impossible. Cette approche a été utilisée dans la mise en œuvre décrite au chapitre 6.

2.6.2 CIRCUS

CIRCUS [Cooper 85] met en œuvre des mécanismes permettant d'utiliser les notions classiques de module et d'appel de procédure, pour programmer des applications réparties et tolérantes aux fautes.

Les mécanismes de CIRCUS se basent sur un modèle de programmation centralisée, modulaire et concurrente: une application est constituée par un ensemble de modules comportant chacun des sous-programmes et des structures de données. L'appel d'un sous-programme d'un module M_2 par un sous-programme d'un module M_1, correspond à l'invocation de M_2 par M_1. Un module peut être invoqué de manière concurrente par plusieurs modules: chaque invocation possède son propre flot d'exécution. Un flot d'exécution est mise en œuvre dans CIRCUS par un **processus léger** (*lightweight process*).

La notion d'appel de procédure à distance permet de facilement répartir une application modulaire. Chaque invocation d'un module distant est réalisée à l'aide d'un RPC. De façon analogue, CIRCUS définit la notion d'**appel de procédure dupliqué** (*replicated procedure call*) pour invoquer un module dupliqué. L'idée consiste à réaliser la tolérance aux fautes d'une application répartie, en dupliquant les modules de l'application sur des nœuds distincts.

Un module dupliqué est appelé une **troupe** (*troupe*). La stratégie de duplication utilisée est la duplication active. Elle s'applique indistinctement aux clients et aux serveurs. La notion d'appel de procédure dupliqué rend la duplication complètement invisible au programmeur d'application. Cette abstraction est mise en œuvre par un pré-processeur qui génère des appels à des primitives utilisant un protocole de communication inter-troupe spécifique à CIRCUS.

Ce protocole garantit l'ordre total des requêtes en s'appuyant sur des transactions et règle le problème du nombre variable de requêtes. Un des intérêts principaux du travail des auteurs de CIRCUS, est d'avoir identifié et résolu les problèmes posés par la duplication active de clients et de serveurs.

2.6.3 DELTA-4

Le projet DELTA-4 [Powell 91a] a défini une architecture générique pour des systèmes distribués devant offrir des garanties relatives au temps réel et à la sûreté de fonctionnement. Cette architecture comporte deux couches principales:

- Deltase (Delta-4 Application Support Environment), une couche supérieure réalisant les abstractions utilisées par les applications réparties et tolérantes aux fautes;
- OSA (Open System Architecture), une couche inférieure chargée de la gestion de la communication.

La couche Deltase réalise l'abstraction d'une machine virtuelle distribuée et tolérante aux fautes. Les applications utilisant Deltase ne sont conscientes ni de la répartition, ni de la duplication. En outre, elles ignorent complètement les particularités du logiciel (i.e. système d'exploitation) et du matériel constituant chaque nœud.

Deltase définit un modèle de programmation basé sur des composants logiciels pouvant communiquer selon deux modèles d'interactions: le modèle client-serveur et le modèle producteur-consommateur. Le modèle **producteur-consommateur** (*producer-consumer*) est une sorte de modèle client-serveur dégénéré: lorsqu'un producteur P communique avec un consommateur C, le producteur envoie une information à C qui "consomme' (i.e. utilise) cette information sans répondre au producteur. Le modèle producteur-consommateur définit une communication uni-directionnelle contrairement au modèle client-serveur qui définit une communication bi-directionnelle.

Deltase fournit au programmeur trois stratégies de duplication prêtes à l'emploi, sous la forme de composants logiciels spécialisés, appelés **entités de duplication** (*replication entities*). Les trois stratégies proposées sont la duplication active, la duplication passive et la duplication semi-active. Cette dernière a été définie par les auteurs de DELTA-4 pour pallier aux inconvénients des deux autres stratégies.

Ces trois stratégies sont applicables aussi bien à des serveurs qu'à des clients. Un composant dupliqué est mis en œuvre à l'aide d'un groupe d'entités de duplication. Chaque copie du composant est contrôlée par une entité de duplication qui joue le rôle d'un véritable filtre à messages. Le groupe d'entités de duplication est coordonné par un protocole de duplication IRP (Inter-Replica Protocol) spécifique à chaque stratégie.

Les protocoles IRP s'appuient sur MCS (Multipoint Communication System), une couche intégrée à la couche de communication OSA. La couche MCS met en œuvre des primitives de

communication capables d'acheminer un message d'un groupe de clients vers un groupe de serveurs avec des garanties d'ordre et d'atomicité. L'intérêt principal de DELTA-4 est qu'il a traité exhaustivement les problèmes posés par la communication entre des clients et des serveurs dupliqués.

2.7 Conclusion

Résumé

La duplication est une technique permettant de mettre en œuvre la tolérance aux fautes dans un système réparti.

La tolérance aux fautes est un moyen d'assurer la sûreté de fonctionnement d'un système informatique. Ce moyen consiste à faire en sorte que le système délivre un service correct malgré l'occurence de fautes. Une faute est une anomalie, affectant le matériel ou le logiciel, suceptible de provoquer la défaillance du système. Un système est défaillant lorsqu'il ne delivre plus un service conforme à sa spécification.

Un système réparti est un système informatique dont les composants logiciels s'exécutent sur des ordinateurs (ou nœuds) interconnectés par un réseau. La communication entre les composants logiciels d'un système réparti est décrite par la notion d'invocation, définie dans le contexte du modèle client-serveur. Pour obtenir un service, un composant client envoie une requête à un composant serveur, celui-ci traite la requête, puis envoie une réponse au client.

La tolérance aux fautes d'un système réparti est assurée en dupliquant les composants critiques du système. Dupliquer un composant O consiste à faire en sorte qu'il existe plusieurs copies O_i de ce composant dans le système. Chaque copie O_i est localisée sur un nœud distinct. Par ce procédé, on cherche à augmenter la disponibilité du composant O. Si une copie O_i défaille, il existe une probabilité non nulle qu'une autre copie O_j soit toujours opérationnelle.

Dans le cas général, un composant est un cs-composant: il joue à la fois les rôles de client et de serveur. Pourtant, la plupart des travaux décrits dans la littérature ne considèrent que la duplication des s-composants: des composants jouant uniquement le rôle de serveurs. C'est probablement pour cette raison que les stratégies de duplication n'ont été définies que pour des s-composants. La duplication des cs-composants a été peu explorée à l'exception de quelques auteurs (par ex. CIRCUS et DELTA-4) qui ont extrapolé les définitions des stratégies de duplication pour traiter des cs-composants dupliqués.

Commentaires

Ce chapitre a présenté le premier volet du contexte dans lequel s'inscrit ce travail de thèse, à savoir la duplication dans les systèmes répartis. L'objectif de cette présentation était double:

- faire le point sur la notion de duplication et sur son utilisation pour mettre en œuvre la tolérance aux fautes;
- donner une vue intuitive des problèmes que posent l'invocation d'un serveur dupliqué par un client dupliqué.

Le modèle client-serveur est bien adapté à la modélisation des interactions entre les composants d'un système réparti. Ce modèle est naturellement mis en œuvre par la notion d'objet. Pour cette raison, les objets sont particulièrement adaptés pour être utilisés comme composants d'un système réparti. Le chapitre suivant va dans cette direction en explorant la notion d'objets dupliqués dans les systèmes répartis.

Chapitre 3

Les objets dupliqués dans les systèmes répartis

3.1 Introduction

Ce chapitre montre que le concept d'*objet* est particulièrement adapté pour modéliser l'unité de duplication dans un système réparti. Son objectif est de rappeler les principaux résultats publiés dans la littérature concernant les *objets dupliqués*.

Présentation générale

L'approche orientée-objets a vu le jour il y a plus de vingt ans [Birtwistle 73], dans le contexte des langages de programmation. Mais ce n'est que ces dix dernières années, que la notion d'objet a connu un essor sans précédent. Très vite, l'approche orientée-objets a alors dépassé le contexte des langages de programmation pour gagner, avec plus ou moins de succès, d'autres domaines de l'informatique comme les bases de données ou les systèmes répartis.

Informellement, l'approche orientée-objets consiste à résoudre un problème en construisant une solution basée sur des entités, appelées *objets*, regroupant chacune un état et un comportement. L'état d'un objet est un ensemble d'informations *encapsulées* à l'intérieur de l'objet: ces informations ne sont pas accessibles directement aux autres objets. Ceux-ci ne peuvent agir sur l'état d'un objet qu'au travers des opérations définissant le comportement de l'objet.

Le comportement d'un objet est décrit par un ensemble d'opérations agissant sur l'état de l'objet. Chaque opération est décrite par une interface (nom de l'opération et paramètres de l'opération) et par un corps contenant les détails de mise en œuvre de l'opération. L'interface de chaque opération est accessible aux autres objets. Au contraire, le corps de chaque opération est encapsulé dans l'objet.

En d'autres termes, un objet est constitué par une partie publique, décrivant les interfaces des opérations et par une partie privée, contenant l'état de l'objet ainsi que les corps des opérations. Les objets interagissent en utilisant la notion d'invocation, définie au chapitre précédent. En fait, la notion d'objet définit plus rigoureusement le concept de composant logiciel utilisé jusqu'ici.

Cette adéquation entre le modèle objet et le modèle client-serveur font des objets de bons candidats pour servir de briques de base pour la construction de systèmes répartis. Dès les

années quatre-vingts, certains auteurs issus de la "communauté objets" (par ex. les concepteurs de Eden [Black 85] ou de SOS [Shapiro 89]), ont adopté une approche orientée-objet pour la conception de systèmes répartis. Ces auteurs ont cherché principalement à étendre le modèle objet au contexte des systèmes répartis, en exploitant la propriété d'encapsulation des objets.

L'idée consiste à considérer la localisation d'un objet (c.-à-d. local ou distant) comme faisant partie de la mise en œuvre de l'objet. Par conséquent, un objet est invoqué toujours de la même façon, qu'il soit local ou distant. Cette approche présente l'avantage de rendre transparente la répartition: un objet client, conçu dans un contexte centralisé peut invoquer un objet serveur distant, sans que le client soit conscient qu'il effectue une invocation à distance.

Par ailleurs, le concept de *classe* d'objets favorise la réutilisabilité du logiciel et permet par exemple, de réutiliser, d'une application à l'autre, des objets réalisant l'invocation à distance. Informellement, une classe est une description générique (i.e. de manière plus imagée, un moule à objets) qui permet de créer des objets dont les comportements et les états (en structure mais pas en valeur) sont identiques. Deux objets d'une même classe ne diffèrent que par leurs noms et par les valeurs de leurs états. En résumé, la "communauté objets" a surtout abordé les systèmes répartis sous l'angle du génie logiciel, les considérant comme un nouveau domaine d'application du paradigme puissant que sont les objets.

Bien que les efforts de normalisation (ODP [Taylor 92], CORBA [OMG 91, OMG 95]) dans le domaine des systèmes répartis s'orientent clairement vers une approche orientée-objets, la "communauté systèmes répartis" n'a pas encore réellement adopté le concept d'objet. Cette communauté a des préoccupations davantage orientées vers le système que vers le génie logiciel. Elle cherche surtout à identifier et à résoudre les problèmes de fond que posent les systèmes répartis.

Cette divergence de points de vue a eu deux conséquences importantes. D'une part, on assiste à l'émergence de systèmes à objets répartis qui ont souvent une approche naïve de la répartition, occultant certains problèmes de fond. La tolérance aux fautes par duplication est un bon exemple de problème occulté par ces systèmes. D'autre part, les algorithmes et les protocoles résultant des recherches de la "communauté systèmes répartis" sont généralement mis en œuvre sous la forme de prototypes "jetables", conçus sans souci réel de réutilisabilité.

Il paraît donc important d'aborder les systèmes répartis avec une approche mixte, tirant parti des deux points de vue. Cette approche a été adoptée dans cette thèse pour étudier et résoudre les problèmes posés par la communication entre objets dupliqués. Dans ce contexte, le chapitre précédent a présenté la duplication avec un point de vue "système". De façon complémentaire, ce chapitre se consacre aux aspects "génie logiciel" en décrivant précisément la notion d'objet dupliqué.

Organisation

La section 3.2 rappelle quelques définitions fondamentales concernant l'approche orientée-objets. La section 3.3 définit la notion d'objet dupliqué et présente les deux problèmes relatifs à l'encapsulation de la duplication. La section 3.4 présente des travaux concernant le premier problème: l'encapsulation de la pluralité. La section 3.5 présente des travaux concernant à la fois le premier et le second problème: l'encapsulation des stratégies de duplication. La section 3.6 conclut le chapitre.

3.2 Définitions

L'approche orientée-objets [Meyer 88, Booch 91], fondée sur la notion de type abstrait [Liskov 74, Guttag 77], peut être caractérisée par quatre concepts essentiels:

- le concept de classe d'objets;
- le concept d'instanciation d'une classe;
- le concept d'encapsulation;
- le concept d'héritage.

Un **type abstrait** (*abstract data type*) $T = (oper(T), prop(T))$ est une spécification formelle d'une catégorie de structures de données dites de type T. Cette spécification est indépendante de toute représentation des données. Elle est entièrement définie par $oper(T)$ qui désigne l'ensemble des opérations applicables à ces structures de données, et par $prop(T)$ qui décrit l'ensemble des propriétés de ces opérations.

Une **classe d'objets** (*object class*) $C^T = (met(C^T), var(C^T))$ est une mise en œuvre particulière d'un type abstrait T telle que: $var(C^T)$ est l'ensemble des variables d'état de chaque objet de la classe C^T, tandis que $met(C^T)$ est l'ensemble des **méthodes** (*methods*), modélisant le comportement de chaque objet de la classe. Chaque méthode est un sous-programme réalisant une opération de $oper(T)$, conformément à la sémantique définie par $prop(T)$.

Un **objet** (*object*) O appartenant à une classe C est le résultat de l'opération d'instanciation appliquée à la classe C. L'opération d'**instanciation** (*instantiation*) est le mécanisme de création d'objets. Il consiste à utiliser une classe C comme un "moule" à objets. Le résultat de l'opération d'instanciation d'une classe C est aussi appelé **instance** (*instance*) de la classe C. Les instances d'une classe C ont toutes le même comportement, défini par $met(C)$. Chaque instance se distingue des autres objets de la classe par son identificateur et par les valeurs de ses variables d'état.

Chaque objet satisfait la propriété d'**encapsulation** (*encapsulation*). Cette propriété garantit que les informations concernant la mise en œuvre d'un objet (i.e. méthodes et variables d'état) sont *confinées* dans l'objet. Seuls l'identificateur d'un objet O et l'interface de chaque méthode (nom de la méthode et liste des paramètres) définie par la classe de O, sont connus par les objets clients de O. Ces informations sont suffisantes pour qu'un objet client C *invoque* un objet serveur S. En effet, les interactions entre les objets se font à l'aide de la notion d'invocation, définie dans le contexte du modèle client-serveur (voir section 2.4.1).

L'**héritage** (*inheritance*) est une relation définie entre les classes d'objets dont l'objectif est de favoriser la réutilisation du logiciel. La relation d'héritage, notée is_a, est définie de la manière suivante. Soient deux classes C_1 et C_2, C_1 hérite de C_2 s'écrit:

$$C_1\ is\_a\ C_2 \iff (met(C_2) \subset met(C_1)) \wedge (var(C_2) \subset var(C_1))$$

C_2 est appelée généralisation de C_1 tandis que C_1 est appelée spécialisation de C_2. Dans la terminologie définie par le langage Smalltalk [Goldberg 83], C_2 est appelée **super-classe** (*superclass*) de C_1, et C_1 est appelée **sous-classe** (*subclass*) de C_2. Les méthodes et les variables d'état de la super-classe C_2 sont incluses dans celles de la sous-classe C_1. En outre, la sous-classe C_1 définit des variables et des méthodes qui lui sont propres. Ainsi, une instance de C_2

aura un comportement identique à une instance de C_1 augmenté par les particularités liées à C_1. En pratique, les programmeurs ne se contentent pas d'ajouter de nouvelles méthodes à une sous-classe. Ils redéfinissent des méthodes héritées de la super-classe: l'interface d'une méthode héritée ne change pas, mais son code est adapté aux spécificités de la sous-classe.

3.3 Notion d'objet dupliqué

Les objets ont des caractéristiques (par ex. encapsulation et invocation) qui favorisent leur utilisation pour la conception de systèmes répartis. Cependant, la notion de duplication entre en conflit avec le concept d'encapsulation. Il n'est pas évident d'encapsuler la duplication d'un objet, c.-à-d. de faire en sorte que cet objet puisse communiquer avec d'autres objets, exactement comme si il n'était pas dupliqué.

Cette section décrit brièvement les difficultés que posent la modélisation d'un objet dupliqué, tandis que les deux sections suivantes présentent les solutions décrites dans la littérature. Ces difficultés seront détaillées au chapitre 4.

Un objet dupliqué O est un *ensemble* d'objets O_i répartis sur plusieurs nœuds. Chaque O_i est en fait une *copie* de l'objet O. La modélisation d'un objet dupliqué pose essentiellement deux problèmes.

Encapsulation de la pluralité Le premier problème concerne *l'encapsulation de la pluralité*. Informellement, ce problème peut être défini par la question suivante: comment faire en sorte qu'un ensemble d'objets répartis soit considéré comme un objet unique, qui puisse aussi bien invoquer d'autres objets qu'être invoqué par d'autres objets? La difficulté principale provient du caractère *dynamique* de la composition de l'ensemble. Toute copie défaillante doit être retirée de l'ensemble. Toute nouvelle copie doit être ajoutée à l'ensemble. Le problème de l'encapsulation de la pluralité intègre le *problème de la désignation du serveur* et le *problème de la désignation du client*, présentés à la section 2.4.2.

Encapsulation des stratégies de duplication Le second problème concerne *l'encapsulation des stratégies de duplication*. Une stratégie de duplication définit le schéma de communication entre un objet dupliqué et les autres objets du système. Elle détermine notamment le sous-ensemble des O_i qui reçoivent ou envoient une requête, et le sous-ensemble des O_i qui reçoivent ou envoient une réponse. La notion de stratégie de duplication entre en conflit avec la notion d'invocation puisque la correspondance "une à une" entre requête et réponse n'est plus respectée. Le problème de l'encapsulation des stratégies de duplication intègre le *problème du nombre variable de requêtes* et le *problème du nombre variable de réponses*, présentés à la section 2.4.2.

De nombreux auteurs se sont intéressés à ces deux problèmes, mais les solutions sont souvent partielles. Il est à noter que les problèmes ne sont pas indépendants: il faut régler le problème de l'encapsulation de la pluralité avant d'aborder celui de l'encapsulation des stratégies de duplication.

3.4 Vers une encapsulation de la pluralité

Cette section présente cinq exemples de travaux consacrés à l'encapsulation de la pluralité.

3.4.1 Synchroniseur d'objets

Un **synchroniseur** (*synchronizer*) [Frolund 93] a pour objectif de coordonner plusieurs objets en imposant des contraintes sur les invocations destinées à chacun de ces objets. Tout synchroniseur est une instance d'une classe de synchroniseurs.

Une classe de synchroniseurs est une liste de couples (sélecteur, action), décrite à l'aide d'une syntaxe spécifique. Un sélecteur est un prédicat permettant de sélectionner des invocations en fonction du nom des objets et du nom des méthodes que ces invocations concernent. À chaque fois que le sélecteur est vérifié, l'action associée est exécutée. Une action peut être une opération arithmétique portant sur les variables d'état du synchroniseur, ou une opération spéciale du synchroniseur portant sur des sélecteurs.

```
Prendre(fourch1, fourch2, philo)
{
  atomic ( (fourch1.prendre(appelant) where appelant = philo),
           (fourch2.prendre(appelant) where appelant = philo) )
}
```

Figure 3.1: Un exemple de synchroniseur d'objets

La figure 3.1 illustre l'utilisation de la notion de synchroniseur pour assurer l'exclusion mutuelle dans l'exemple classique des philosophes. Chaque philosophe a une méthode `Manger(fourch1, fourch2)` qui est invoquée lorsqu'un philosophe a faim. Les deux paramètres de la méthode `Manger` désigne les deux fourchettes dont un philosophe a besoin pour manger.

L'invocation de la méthode `Manger` provoque l'instanciation du synchroniseur `Prendre` décrit par la figure 3.1. Ce synchroniseur a trois paramètres: l'identité du philosophe et celles des deux fourchettes. L'opération `atomic` exprime que les invocations de la méthode `Prendre` sur `fourch1` et `fourch2` doivent se faire de façon atomique (sémantique "tout ou rien").

La notion de synchroniseur est intégré au langage d'acteurs défini dans [Agha 86]. Le compilateur traduit les contraintes exprimées par les synchroniseurs en termes d'invocations entre les objets concernés. Selon les auteurs de [Frolund 93], les synchroniseurs peuvent être mise en œuvre comme une extension de tout langage orienté-objet supportant la concurrence.

L'intérêt du concept de synchroniseur est de fournir une abstraction de haut niveau permettant de coordonner plusieurs objets. Cependant, les objets coordonnés sont invoqués individuellement: il n'y a aucun moyen d'adresser une invocation à *l'ensemble* de ces objets. De plus, l'ensemble des objets coordonnés est défini une fois pour toutes au moment de l'instanciation du synchroniseur. Par conséquent, la notion de synchroniseur semble difficilement utilisable pour modéliser des objets dupliqués.

3.4.2 Mandataire

Un **mandataire** (*proxy*) [Shapiro 86] est un représentant *local*, d'un service réparti mis en œuvre par un ensemble de serveurs localisés sur plusieurs nœuds distants. Le rôle d'un mandataire est

de cacher aux clients d'un service, le caractère réparti de ce service. Pour invoquer un service réparti, un client invoque le mandataire local de ce service. Du point de vue d'un client, le mandataire et le service réparti sont confondus.

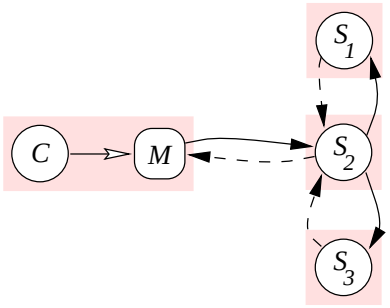

Figure 3.2: Un exemple de mandataire

Sur la figure 3.2, un mandataire M représente un service réparti mis en œuvre par trois serveurs: S_1, S_2, S_3. Lorsque le client C désire invoquer le service, il invoque le mandataire M. Ce dernier transmet la requête aux serveurs selon un protocole convenu avec les serveurs. Sur cet exemple, M transmet la requête à S_2 (flèches en trait continu), qui la transmet à S_1, S_3. Ces derniers envoient une réponse à S_2 (flèches en trait discontinu) qui la transmet à M. Enfin, C reçoit la réponse lorsque l'invocation de M se termine. Le client C ne connait que le mandataire M: il ne perçoit ni les trois serveurs, ni le protocole utilisé par M pour communiquer avec eux.

La notion de mandataire a été initialement mise en œuvre dans le système SOS [Shapiro 89], un système d'exploitation orienté-objets, réalisé en C++ [Stroustrup 94] au-dessus de UNIX. Le système SOS fournit tous les services de base nécessaires à la gestion d'objets répartis: création/destruction d'objets, désignation. migration, stockage, etc.. Depuis, de nombreux auteurs s'intéressant à l'approche objet dans les systèmes répartis ont repris cette idée ou s'en sont inspirés. Les exemples cités dans les sections suivantes l'attestent. Par ailleurs, la notion de représentant symétrique, proposée au chapitre 5, est une extension de la notion de mandataire.

Le principal intérêt de la notion de mandataire est qu'elle a étendu le concept d'encapsulation aux objets répartis. Un objet réparti utilisant cette notion est invoqué comme un objet local: la répartition et la pluralité sont encapsulées par le mandataire. En d'autres termes, les clients d'un objet réparti ne perçoivent ni le nombre de serveurs, ni leurs localisations. Comme un mandataire est un objet, i.e. une entité programmable, on peut imaginer n'importe quel protocole de communication entre lui et ses serveurs. En particulier, ce protocole pourrait être celui d'une stratégie de duplication.

Pourtant, la notion de mandataire ne suffit pas pour mettre en œuvre un objet dupliqué: rien n'est dit sur la structure de l'ensemble de serveurs, ni sur son évolution face aux défaillances/redémarrages des copies. C'est probablement pour cette raison que les mêmes auteurs ont intégré la notion de mandataire au concept d'objet fragmenté (cf. section 3.5.1).

3.4.3 Objet dispersé

Un **objet dispersé** (*dispersed object*) [Finke 93] est un objet dont l'état est composé par plusieurs "sous-objets" *de même classe* géographiquement répartis sur des nœuds distincts. Tout objet modélisant une collection répartie d'objets (par ex. une liste, un ensemble ou un vecteur d'objets) est un bon exemple d'objet dispersé. Dans ce cas, les éléments de la collection sont répartis dans des sous-collections, localisée chacune sur un nœud distinct. Chaque sous-collection est une partie de l'objet dispersé représentant la collection.

La figure 3.3 donne un exemple d'objet dispersé composé de deux sous-collections réparties sur deux nœuds. Chaque sous-collection est représenté par un cercle en pointillé. Les objets appartenant à une sous-collection sont représentés par des cercles.

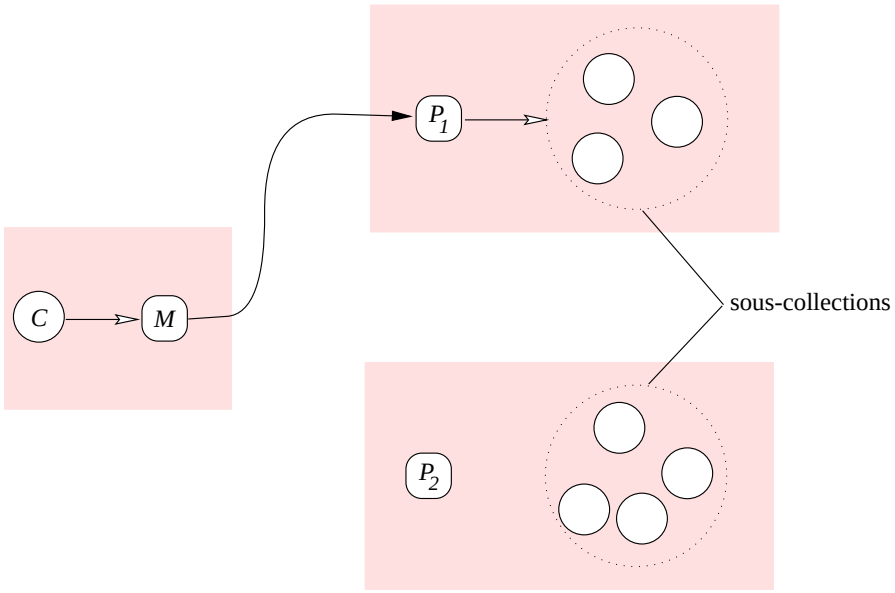

Figure 3.3: Un exemple d'objet dispersé

Un objet dispersé est invoqué comme un objet local. La répartition d'un objet dispersé est rendue transparente par la notion de mandataire. Le mandataire est chargé d'analyser la requête afin de l'acheminer vers le sous-objet concerné. À chaque sous-objet est associé un **pilote** (*driver*) qui réceptionne la requête provenant du mandataire. Après avoir traité la requête, le sous-objet transmet la réponse à son pilote qui l'achemine vers le mandataire. Ce dernier délivre alors la réponse au client. Sur la figure 3.3, un client C invoque le mandataire M qui transmet cette invocation au pilote P_1 qui contrôle l'une des sous-collections.

Ces mécanismes sont mis en œuvre dans HERON, un environnement de programmation d'applications distribuées basé sur le langage Eiffel [Meyer 87]. La répartition des objets est décrite à l'aide d'un langage de configuration. À partir de cette description, HERON génère automatiquement les mandataires et les pilotes. Par conséquent, la tâche du programmeur se limite à écrire son application en Eiffel, et à décrire la répartition désirée à l'aide du langage de configuration.

La notion d'objet dispersé permet d'invoquer plusieurs objets répartis comme un seul objet. Elle présente toutefois deux contraintes importantes. D'une part, la composition de l'objet dispersé

est statique. D'autre part, tous les sous-objets doivent être de la même classe que l'objet dispersé. Ceci est dû à la manière dont le mandataire traite une invocation: il recherche le sous-objet concerné par l'invocation. Cette organisation interne rend le concept d'objet dispersé plus adapté à la coopération entre plusieurs objets qu'à la duplication. Dans le cas d'une coopération, chaque objet a un rôle spécifique connu par le mandataire. Celui-ci peut ainsi aiguiller chaque invocation vers l'objet compétent.

3.4.4 Troupeau d'objets

Un **troupeau** (*gaggle*) d'objets [Black 93] est un ensemble d'objets, répartis sur plusieurs nœuds, appartenant à la même classe Cl et pouvant être invoqués comme un *seul* objet de la classe Cl. La création d'un troupeau se fait en deux temps. Tout d'abord, un **gestionnaire de troupeau** (*gaggle manager*) est construit par instanciation de la classe *Gaggle*. Cette construction est paramétrée par le nom de la classe des objets qui vont constituer le troupeau.

Dans un second temps, une méthode particulière du gestionnaire de troupeau est invoquée à chaque fois que l'on veut ajouter un objet dans le troupeau. En outre, le gestionnaire de troupeau définit une méthode qui retourne une référence sur une **interface de troupeau** (*gaggle interface*). Cette dernière est une instance de la classe à laquelle appartiennent les objets du troupeau.

Pour invoquer un troupeau, un objet client doit avoir une référence sur une interface de troupeau. Dès lors, l'invocation du troupeau apparaît au client comme l'invocation d'un objet local. En réalité, l'interface de troupeau choisit un *seul* objet du troupeau et l'invoque à distance. Le choix de l'objet invoqué est fait de manière interne par la classe *Gaggle*.

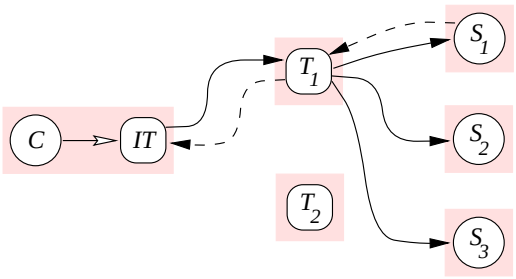

Figure 3.4: Un exemple de troupeau d'objets

La figure 3.4 illustre l'utilisation d'un troupeau T pour invoquer un objet dupliqué S. Le troupeau T est composé de deux objets T_1, T_2. L'objet S est dupliqué en trois exemplaires: S_1, S_2, S_3. Le troupeau T permet à un client C d'invoquer le serveur S sans devoir connaître l'emplacement des copies de S.

Pour invoquer S, le client C invoque localement l'interface de troupeau IT. Cette dernière invoque à distance le membre T_1 qui procède à l'invocation du serveur dupliqué S. Si T_1 défaille, l'interface IT choisit T_2 pour transmettre les invocations à S.

La notion de troupeau a été mise en œuvre dans EMERALD [Black 87, Jul 93], un langage orienté objet supportant la distribution. Deux primitives ont été ajoutées au langage initial pour mettre en œuvre le concept de troupeau. Ces primitives correspondent à l'ajout d'un objet

à un troupeau et à la création d'un interface de troupeau.

L'objectif visé par les auteurs du concept de troupeau d'objets est de fournir une abstraction destinée à encapsuler la pluralité. Cet objectif est atteint puisqu'un troupeau d'objets peut être invoqué comme un seul objet. Cependant, trois points importants restent en suspens. Premièrement, l'interface du troupeau ne transmet une invocation qu'à un seul objet de son choix. Deuxièmement, s'il est possible d'ajouter de nouveaux objets à un troupeau, il n'est pas possible d'en enlever. Enfin, aucun mécanisme ne permet de déterminer la composition d'un troupeau. Ces trois points résultent du souci affiché par les auteurs de faire en sorte que le concept de troupeau soit un mécanisme minimal, à partir duquel le programmeur peut construire des sémantiques plus ambitieuses. En particulier, ils assurent que le concept de troupeau peut être enrichi afin de mettre en œuvre des objets dupliqués.

3.4.5 Communauté d'objets

Une **communauté d'objets** (*object community*) [Chiba 93] est un ensemble d'objets, répartis sur des nœuds différents et capables de réagir collectivement à une invocation. La notion de communauté est basée sur un modèle d'objets à deux niveaux. Le premier niveau est constitué par les objets utilisés habituellement pour la programmation d'applications. Le second niveau est constitué par les **méta-objets** (*meta-objects*) dont le rôle est de contrôler certaines invocations faites aux objets du premier niveau.

Lorsque le programmeur désire contrôler les invocations d'un objet O, il lui associe un méta-objet. L'objet O est alors appelé **objet réflexif** (*reflective object*). Le contrôle des invocations de O est fait sélectivement en fonction de la méthode appelée. Les méthodes dont les invocations sont contrôlées par le méta-objet sont appelées **méthodes réflexives** (*reflective methods*). Le contrôle exercé par le méta-objet sur une méthode réflexive consiste à intercepter toute invocation concernant cette méthode. Cette interception consiste à exécuter une méthode du méta-objet *avant* d'appeler la méthode réflexive. La méthode du méta-objet met en œuvre des mécanismes devant rester cachés au programmeur d'applications.

Précisément, une communauté d'objets est réalisée à partir d'une méta-communauté, i.e. une communauté de méta-objets. La notion de méta-communauté est mise en œuvre par une classe spécifique de méta-objets appelée $OcCoreMetaObj$. Cette classe définit les méthodes permettant de gérer une méta-communauté: créer et détruire une méta-communauté, rejoindre et quitter une méta-communauté. Par conséquent, une méta-communauté est une entité dynamique: des méta-objets peuvent rejoindre ou quitter une méta-communauté à tout instant.

La classe $OcCoreMetaObj$ définit en outre des méthodes permettant aux membres d'une méta-communauté de communiquer à travers le réseau: invoquer un méta-objet distant, diffuser un message de notification à tous les membres de la méta-communauté, attendre une notification diffusée par un membre de la méta-communauté. La primitive de diffusion de messages de notification garantit que tous les membres de la méta-communauté reçoivent la même *séquence* de messages (ordre total). Ces messages permettent aux membres d'une méta-communauté de réaliser un traitement de manière concertée (par ex. chaque méta-objet invoque une méthode particulière de son objet réflexif).

Comme, une communauté d'objets est construite à partir d'une méta-communauté, chaque objet doit être associé à un méta-objet appartenant à la même méta-communauté. Alors que les méta-objets sont conscients d'appartenir à une méta-communauté, les objets réflexifs associés

ne perçoivent pas la notion de communauté.

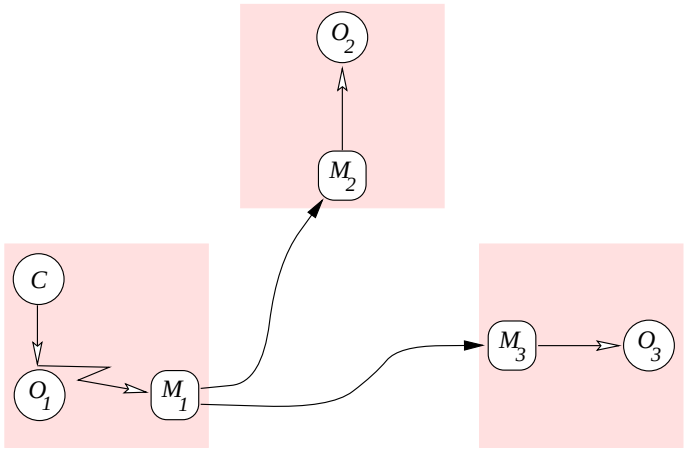

Figure 3.5: Un exemple de communauté d'objets

La figure 3.5 illustre l'invocation d'une communauté d'objets par un client C. La communauté comporte trois objets: O_1, O_2, O_3. À chaque membre O_i est associé un méta-objet M_i appartenant à une méta-communauté. La méta-communauté est donc composée de trois membres: M_1, M_2, M_3.

Lorsqu'un objet client C invoque O_1, le méta-objet M_1 intercepte cette invocation et diffuse une notification aux autres membres de la méta-communauté. À la réception de cette notification, M_2 et M_3 transmettent l'invocation à leurs objets réflexifs.

Ces concepts ont été mis en œuvre dans le cadre de Open C++, une extension du langage C++. Des constructions syntaxiques ont été ajoutées au langage initial pour déclarer une classe d'objets réflexifs, une méthode réflexive ou une classe de méta-objets. De plus, différentes sous-classes de $OcCoreMetaObj$ ont été définies afin de mettre en œuvre d'autres abstractions utiles pour la programmation répartie (par ex. les transactions).

Parmi les exemples présentés dans cette section, Open C++ propose les mécanismes les plus aboutis pour encapsuler la pluralité. L'intérêt majeur de cette approche réside dans la distinction des deux niveaux d'objets. Cette distinction permet de confiner les aspects liés à la distribution et à la pluralité au sein des méta-objets. La programmation des objets du premier niveau n'est donc pas affectée par ces aspects.

Cependant, certains points restent obscurs. Bien que les auteurs décrivent le caractère dynamique d'une méta-communauté, le cas de la défaillance d'un membre de la méta-communauté n'est pas évoqué. D'autre part, il n'est pas possible d'invoquer *directement* une communauté d'objets. En effet, la notion de mandataire n'existe pas. Pour invoquer une communauté d'objets, un objet client doit invoquer explicitement un membre de la communauté. À partir de là, le méta-objet associé diffuse une notification aux autres membres de la méta-communauté.

3.5 Vers une encapsulation des stratégies de duplication

Cette section présente quatre exemples de travaux consacrés à l'encapsulation des stratégies de duplication.

3.5.1 Objets fragmentés dans SOS

Un **objet fragmenté** (*fragmented object*) [Makpangou 92] est présenté par ses auteurs comme une extension du concept d'objet aux systèmes répartis. Un objet fragmenté peut être abordé selon deux points de vue. D'un point de vue externe, un objet fragmenté apparaît comme un seul objet monolithique. D'un point de vue interne, un objet fragmenté est constitué par un ensemble de fragments répartis sur plusieurs nœuds et interconnectés par des objets liens. Les **fragments** (*fragments*) d'un objet fragmenté sont des objets ordinaires appartenant à une même classe. Un **objet lien** (*connective object*) est lui-même un objet fragmenté. Il met en œuvre la communication entre les fragments de l'objet fragmenté dont il fait partie.

Pour invoquer un objet fragmenté, un objet client s'adresse à un fragment interface résidant sur le même nœud. Ce fragment interface correspond exactement à la définition d'un mandataire. Le fragment interface transmet *éventuellement*[1] l'invocation aux autres fragments via les objets liens, puis il retourne une réponse au client. En effet, le protocole adopté par les fragments pour le traitement d'une invocation est conditionné par la classe des fragments. Le modèle est suffisamment flexible pour imaginer différentes stratégies de coordination entre les fragments d'un objet fragmenté.

Le modèle à objets fragmentés a été utilisé [Gourhant 92] pour mettre en œuvre des objets dupliqués. Un objet dupliqué est un objet fragmenté constitué par plusieurs fragments appelés copies et par trois objets liens: un canal de duplication, un canal de journalisation et une mémoire stable. Une **copie** (*replica*) est composée par deux objets: un objet donnée et un objet interface. L'**objet donnée** (*data object*) correspond à l'état de la copie. L'**objet interface** (*interface object*) gère l'interaction avec les clients locaux.

Lorsqu'un objet client invoque un objet dupliqué, deux scénarios sont possibles, selon qu'il existe ou non une copie sur le nœud du client. S'il existe une copie locale, l'objet interface joue le rôle de mandataire de l'objet fragmenté. Sinon, un mandataire est explicitement créé sur le nœud du client. Lorsqu'il reçoit une invocation, un objet interface peut la transmettre à l'objet donnée, au canal de journalisation ou aux autres fragments via le canal de duplication. Ces choix sont déterminés par la stratégie de duplication de l'objet fragmenté.

Le **canal de duplication** (*replicating channel*) est lui-même constitué par deux objets liens: un canal de diffusion et un protocole de synchronisation. Le **canal de diffusion** (*multicast channel*) permet d'assurer la communication entre les copies. Selon sa classe, il offre différentes garanties en termes d'ordonnancement de messages et d'atomicité. Le **protocole de synchronisation** (*synchronization protocol*) est chargé de garantir la cohérence des copies. Selon sa classe, il met en œuvre des mécanismes comme un verrou réparti ou un jeton réparti.

Le **canal de journalisation** (*logging channel*) est lui-même un objet dupliqué. Il permet de conserver la trace des modifications effectuées sur les copies de l'objet dupliqué. Enfin, la **mémoire stable** (*reliable storage object*) stocke l'état des copies. Elle permet de mettre en œuvre des points de reprise utiles pour certaines stratégies de duplication (par ex. la duplication

[1] Une alternative consiste à traiter l'invocation localement.

passive).

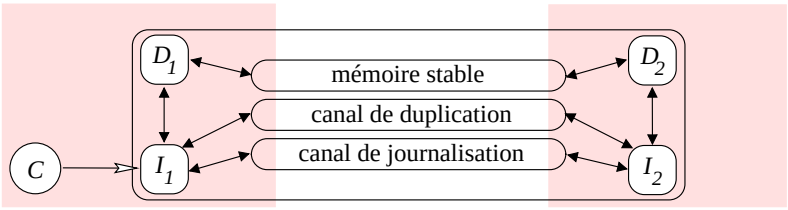

Figure 3.6: Mise en œuvre d'un objet dupliqué à l'aide d'un objet fragmenté

La figure 3.6 donne un exemple d'objet dupliqué réalisé à l'aide du concept d'objet fragmenté. Cet objet dupliqué possède deux copies. Chaque copie est constitué par un objet interface, étiqueté I_i, et par un objet donnée, étiqueté D_i. Pour invoquer cet objet dupliqué, un client C invoque I_1, l'objet interface qui réside sur le même nœud que lui. Selon la stratégie de duplication, l'objet interface I_1 invoque son objet donnée D_1 et/ou transmet cette invocation aux autres copies par l'intermédiaire du canal de duplication.

Les objets fragmentés ont été mis en œuvre dans FOG (Fragmented Object Generator) une extension au langage C++. Cette extension définit un langage de description d'un objet fragmenté et une hiérarchie de classes d'objets fragmentés de base. La description d'un objet fragmenté en langage FOG est traduite en C++ par un compilateur spécifique à FOG.

Les auteurs de [Gourhant 92] affirment que les objets fragmentés sont adaptés à la mise en œuvre d'objets dupliqués. Ils insistent surtout sur la flexibilité du modèle en décrivant les différentes possibilités offertes par les objets fragmentés. Toutefois, plusieurs aspects importants ne sont pas détaillés. En particulier, on ne sait pas si le modèle supporte la duplication de cs-composants (i.e. des objets à la fois clients et serveurs). Le seul exemple mentionné dans [Gourhant 92] concerne un s-composant. D'autre part, il n'est rien dit sur la mise en œuvre d'une stratégie de duplication particulière. Enfin, les problèmes liés à l'occurence de défaillances ne sont pas abordés.

3.5.2 Groupes d'interfaces dans ANSA

ANSA (Advanced Networked Systems Architecture) est une proposition d'architecture orientée-objets pour systèmes répartis hétérogènes, développée dans le contexte du projet ESPRIT ISA [ANSA 91]. Cette proposition définit la notion de groupe d'interfaces pour encapsuler la duplication dans le modèle objet. Un **groupe d'interfaces** (*interface group*) [Olsen 91] est un regroupement de plusieurs interfaces fournissant l'abstraction d'une interface unique. L'objectif visé est de permettre à un ensemble d'objets d'être invoqués (rôle de serveur) *et* d'invoquer (rôle de client) comme si cet ensemble n'était qu'un seul et même objet.

Une **interface** (*interface*) est un point d'accès aux services fournis par un objet. Pour invoquer un objet O, un client doit invoquer l'interface[2] de O. Celle-ci peut être vue comme un objet spécialisé chargé de contrôler les invocations destinées à O.

[2]En réalité, un objet O peut avoir plusieurs interfaces distinctes. Cette possibilité n'est pas envisagée ici afin de simplifier l'explication.

Un groupe d'interfaces a une structure complexe. Outre ses membres, il comporte plusieurs entités nécessaires au fonctionnement du groupe. On peut distinguer les entités associées à chaque membre du groupe, et les entités associées à chaque mandataire du groupe. À chaque membre d'un groupe d'interfaces, est associé un **agent** (*member agent*) chargé d'intercepter les requêtes destinées à ce membre. Le rôle d'un agent consiste à filtrer les requêtes multiples (cf. le problème du nombre variable de requêtes évoqué à la section 2.4.2), à communiquer avec les autres agents pour garantir que chaque membre du groupe reçoit la même séquence de requêtes (ordre total), et à détecter les défaillances des autres agents.

Un mandataire d'un groupe d'interfaces est constitué par une interface de groupe, une interface de gestion, un distributeur et un collecteur. L'**interface de groupe** (*group interface*) est chargée de réceptionner les requêtes concernant les services fournis par le groupe d'interfaces. Tout client d'un groupe d'interfaces invoque l'interface de groupe pour obtenir un service. L'**interface de gestion** (*management interface*) est chargée de réceptionner les requêtes concernant la gestion de la composition du groupe (par ex. ajouter un membre, retirer un membre, etc.). Cette interface n'est accessible qu'aux clients pouvant agir sur la composition du groupe.

Le **distributeur** (*distributor*) est chargé de diffuser toute requête provenant de l'interface de groupe à tous les membres du groupe. Le **collecteur** (*collator*) est chargé de récupérer toutes les réponses provenant des membres du groupe, afin de construire une réponse unique (problème du nombre variable de réponses) et de la transmettre à l'interface de groupe. Ce dernier transmet alors cette réponse au client qui a émis la requête correspondante.

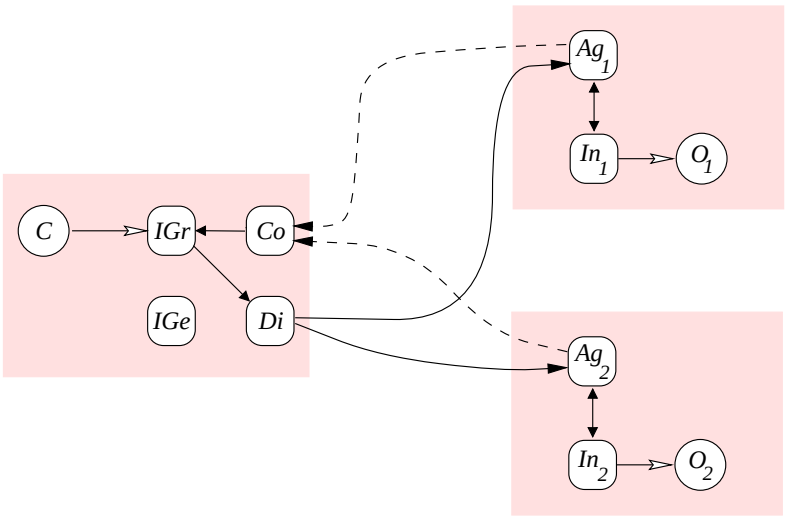

Figure 3.7: Un exemple de groupe d'interfaces

La notion de groupe d'interfaces a été utilisée pour mettre en œuvre deux stratégies de duplication: la duplication active et la duplication passive. La duplication active nécessite essentiellement que toutes les copies d'un objet dupliqué perçoivent la même séquence de requêtes. Cette contrainte est assurée à la fois par le distributeur (diffusion) et par les agents (ordre total) associés à chacun des membres d'un groupe d'interfaces. Les multiples réponses à une requête générées par les copies d'un objet dupliqué activement sont filtrées par le collecteur associé à tout mandataire. La duplication passive nécessite essentiellement un mécanisme permettant à

la copie primaire de diffuser son état aux copies secondaires dans un but de synchronisation. Le distributeur peut être utilisé à cet effet. La détection de la défaillance de la copie primaire et l'élection d'une nouvelle copie primaire sont pris en charge par les agents.

La figure 3.7 donne un exemple de groupe d'interfaces mettant en œuvre la duplication active pour un objet dupliqué O. Cet objet a deux copies O_1, O_2. Un membre du groupe d'interfaces est associé à chaque copie: le groupe comporte deux membres In_1, In_2. Chaque membre In_i contrôle les accès à la copie O_i. De plus, un agent Ag_i est associé à chaque membre In_i.

Pour invoquer le groupe d'interfaces, un client C s'adresse au mandataire du groupe. Ce dernier est composé par une interface de groupe IGr, une interface de gestion IGe, un collecteur Co et un distributeur Di.

Lorsque C invoque IGr, celle-ci transmet l'invocation au distributeur Di. Ce dernier transmet l'invocation aux deux agents Ag_1, Ag_2. Chaque Ag_i transmet l'invocation au membre In_i. Ce dernier invoque la copie O_i puis transmet la réponse qu'il obtient à l'agent Ag_i. Le collecteur Co filtre les réponses redondantes que lui transmettent les Ag_i. Puis, il transmet la réponse au client C via l'interface IGr.

La notion de groupe d'interfaces a été partiellement mise en œuvre dans un prototype écrit en langage C [Kernighan 77]. De façon surprenante, la programmation de ce prototype n'est pas orientée-objet. Le prototype est constitué par un pré-processeur, des bibliothèques de sous-programmes et par un outil de configuration. Pour définir un groupe d'interfaces, le programmeur doit le déclarer en utilisant une syntaxe ad hoc. Au moment de la compilation, le pré-processeur traduit cette déclaration en appels à des sous-programmes contenus dans les bibliothèques.

En outre, la composition de chaque groupe d'interfaces est spécifiée de manière interactive pendant l'exécution, à l'aide de l'outil de configuration. En d'autres termes, la composition d'un groupe d'interfaces n'est pas gérée automatiquement par le prototype. La communication est basée sur GEX, une extension du protocole REX [Otway 87], qui intègre un protocole de diffusion totalement ordonnée basé sur la transmission d'un jeton [Chang 84].

L'approche adoptée par ANSA pour encapsuler la duplication est intéressante. Selon ses auteurs elle s'inspire fortement du modèle défini par la norme ODP (Open Distributed Processing) [Taylor 92]. Cette approche considère d'emblée l'ensemble des problèmes en envisageant des clients et des serveurs dupliqués, et plusieurs stratégies de duplication. Toutefois, bien que le concept de groupe d'interfaces soit décrit précisément, la prise en compte des différentes stratégies de duplication n'est pas clairement expliquée. Aucun exemple ne décrit en détail l'invocation d'un objet dupliqué par un autre objet dupliqué. Par ailleurs, le problème des défaillances n'est pas pris en compte de manière satisfaisante: l'évolution de la composition d'un groupe en fonction des défaillances et de l'arrivée de nouveaux membres n'est pratiquement pas décrite.

3.5.3 Objets dupliqués dans ARJUNA

ARJUNA [Shrivastava 94] est un ensemble d'outils logiciels destinés à la programmation orientée-objets d'applications réparties et tolérantes aux fautes. Le modèle de programmation défini par ARJUNA est basé sur des transactions imbriquées manipulant des objets persistants[3]. En d'autres termes, l'invocation d'un objet se fait toujours dans le contexte d'une transaction

[3]Un objet persistant se caractérise par une longue durée de vie. Il est stocké en mémoire non volatile lorsqu'il n'est pas utilisé et chargé en mémoire volatile au moment de son utilisation.

(ou d'une sous-transaction). Les auteurs de ARJUNA ont mis en œuvre la duplication active et la duplication passive pour permettre au programmeur de garantir la tolérance aux fautes des objets critiques de son application.

Ces stratégies de duplication s'appuient sur des mécanismes fournis par le service de désignation de ARJUNA (appelé *binding module*) et par son service d'invocation à distance (appelé *RPC module*). Le service de désignation assure la correspondance entre l'identificateur d'un objet et l'adresse d'un objet. L'identificateur est global (connu par toutes les machines où ARJUNA s'exécute) et indépendant de la localisation de l'objet. Par contre, l'adresse d'un objet permet de localiser précisément la représentation physique d'un objet.

Un objet dupliqué est considéré comme un objet ayant plusieurs représentations, réparties sur plusieurs machines. Chaque représentation correspond à une copie de l'objet. En d'autres termes, les copies d'un objet dupliqué forment un groupe d'objets. Le groupe est désigné dans son ensemble à l'aide de l'identificateur de l'objet tandis que chaque copie est désignée par son adresse. L'association entre l'identificateur d'un objet et la liste des adresses de ses copies est conservée dans la GVD (Group View Database), un objet dupliqué particulier géré par le service de désignation. Le service d'invocation à distance permet de diffuser une invocation à tous les membres non défaillants d'un groupe d'objets. Ce service est basé sur rel/REL_{fifo}, un protocole de diffusion fiable assurant l'ordre FIFO.

Pour invoquer un objet O dupliqué passivement, la transaction associée commence par déterminer l'adresse de la copie primaire à partir de la liste des copies de O, contenue dans la GVD. Si la copie primaire est défaillante, une des copies secondaires est choisie comme nouvelle copie primaire. L'ancienne copie primaire est marquée comme défaillante dans la GVD et l'invocation est effectuée sur la nouvelle copie primaire. Dès que la transaction a trouvé une copie primaire opérationnelle, l'invocation de O est effectuée sur cette copie. Puis, la transaction met à jour les copies secondaires accessibles à partir de la copie primaire. Avant de valider, la transaction marque toutes les copies secondaires inaccessibles comme défaillantes dans la GVD. En dernier lieu, le résultat de l'invocation est transmis au client. Si la copie primaire défaille pendant l'invocation, la transaction est annulée et la copie primaire est marquée comme défaillante dans la GVD.

L'invocation d'un objet O dupliqué activement se fait de façon similaire. La différence réside essentiellement dans le fait qu'il faut réaliser l'invocation sur *toutes* les copies non défaillantes de O. Chaque copie est invoquée dans une sous-transaction. Si une copie défaille, la sous-transaction associée est annulée. Lors de la validation de la transaction englobante, la GVD est mise à jour afin de tenir compte des copies défaillantes.

La définition de la duplication active exige que chaque copie de O perçoive la même *séquence* d'invocations. Comme le protocole de diffusion du service d'invocation à distance n'assure pas l'ordre total, les copies de O ne reçoivent pas la même séquence d'invocations. La contrainte est assurée par le contrôle des invocations concurrentes que mettent en œuvre les méthodes définies par la classe de O. Chaque méthode invoquée se charge d'acquérir le verrou nécessaire au traitement qu'elle va effectuer. Dans le cas d'un objet dupliqué activement, une invocation n'est effectuée que si toutes les copies ont pu être verrouillées. Ainsi, lorsque des invocations concurrentes parviennent dans des ordres différents aux copies de O, chaque invocation ne verrouille que les copies qu'elle a atteint avant les autres invocations. Par conséquent, toutes les invocations échouent et les transactions associées sont annulées.

ARJUNA est programmé en C++ et s'exécute au-dessus de UNIX et de TCP/IP. Les outils qu'il

fournit au programmeur sont essentiellement des classes écrites en C++ et un pré-processeur qui génère du C++. Ces classes mettent en œuvre les différents services de ARJUNA (gestion de transactions, désignation, persistance et invocation à distance). La répartition et la duplication sont cachées au programmeur: un objet dupliqué est invoqué avec la syntaxe C++ habituelle et le pré-processeur se charge de générer le code C++ nécessaire (par ex. accès à la GVD, diffusion d'un message, etc.).

Les auteurs de ARJUNA ne décrivent que la duplication des serveurs. Ils envisagent [Little 91, Shrivastava 94] pourtant qu'un objet dupliqué puisse en invoquer un autre. À ce propos, ils évoquent l'existence d'un mécanisme de filtrage, intégré au protocole rel/REL_{fifo}, dont l'objectif est de régler le problème du nombre variable de requêtes, dans le cas d'un client dupliqué activement. Plus généralement, l'interaction entre des objets dupliqués avec des stratégies différentes n'est jamais décrite. Ainsi, ARJUNA ne réalise que partiellement l'encapsulation de la pluralité et l'encapsulation des stratégies de duplication.

3.5.4 Systèmes à objets basés sur ISIS

Plusieurs auteurs ont choisi de développer un modèle objet au-dessus du logiciel ISIS (voir section 2.6.1). Cette approche consiste à définir une interface objet au concept de groupe de processus défini par ISIS.

Groupes d'objets et RDOM

En fondant la société IDS (Isis Distributed Systems) en 1988, les concepteurs d'ISIS ont vite compris l'intérêt de définir une interface orientée-objets à leur logiciel. Outre l'adéquation technique des objets aux systèmes répartis, le "boom" commercial que connaît actuellement l'approche orientée-objets était de bonne augure pour cette jeune société. Les activités d'IDS dans ce contexte ont été menées sur deux fronts. D'une part, le logiciel RDOM (Reliable Distributed Object Manager) a été développé. D'autre part, une proposition d'intégrer les groupes d'objets à la norme CORBA 2.0 a été faite.

La norme CORBA (Common Request Broker Architecture) est une proposition d'architecture orientée-objet pour les systèmes répartis hétérogènes. Son objectif est donc le même que celui de ANSA. Alors que ANSA s'appuie sur la norme ODP définie par l'ISO, CORBA est une norme définie de façon indépendante par OMG (Object Management Group) un groupe de constructeurs informatiques (essentiellement DEC, HP, Sun, NCR). Une première proposition (CORBA 1.0) [OMG 91] a été faite en 1991 dans laquelle il n'y avait aucune référence au problème de la duplication et de la tolérance aux fautes en général. Une seconde proposition (CORBA 2.0) [OMG 95] a vu le jour en mai 1995 dans laquelle un concept de transactions a été intégré.

Cependant, la duplication est toujours absente de CORBA. Pourtant, IDS a publié en 1993 une proposition d'intégration du concept de groupe à CORBA [IDS 93]. Cette intégration consiste à étendre l'ORB (Object Request Broker), la couche de communication de CORBA, avec des mécanismes mettant en œuvre la notion de **groupe d'objets** (*object groups*). Dans cette proposition, un groupe d'objets doit apparaître comme un seul objet, aussi bien comme client que comme serveur. Le problème de l'encapsulation de la pluralité est donc le souci majeur de cette proposition. Les stratégies de duplication ne sont pas explicitement mentionnées, mais IDS fait référence à tous les outils fournis par ISIS (multicasts, changement de vue, etc.) qui

permettent de mettre en œuvre ces stratégies.

Parallèllement à sa proposition, IDS a développé RDOM, un logiciel réalisant une interface à ISIS, orientée-objets et compatible CORBA. Ce logiciel consiste en un ensemble de classes d'objets permettant au programmeur d'utiliser les fonctionnalités d'ISIS. Pour l'essentiel, ces classes font appel aux fonctions et aux procédures définies dans les bibliothèques fournies par ISIS. Les classes de RDOM existent pour plusieurs langages orientés-objets, notamment pour Smalltalk et C++. Bien que RDOM se veut être une concrétisation de la proposition de IDS, ce logiciel ne met pas en œuvre tous les éléments de la proposition. Notamment, le concept de groupe défini dans ISIS (et dans RDOM) n'est pas conçu pour des objets clients. Dans RDOM, la pluralité n'est donc pas encapsulée complètement.

ELECTRA

ELECTRA [Maffeis 95] est un environnement écrit en C++ destiné à la programmation orientée-objets d'applications réparties et tolérantes aux fautes. Cet environnement respecte la norme CORBA et définit un ensemble de classes d'objets fournissant des abstractions utiles aux programmeurs d'applications. Ces abstractions (groupes d'objets, multicasts, etc.) sont essentiellement les mêmes que celles proposées par ISIS, mais adaptées à des objets. Par conséquent, ELECTRA n'apporte pas de meilleures solutions que RDOM au problème de l'encapsulation de la duplication.

Les classes définies par ELECTRA utilisent les services d'une machine virtuelle qui constitue la couche basse de ELECTRA. Cette machine virtuelle permet aux applications programmées avec ELECTRA, d'être indépendantes des services répartis sur lesquels se base ELECTRA. En effet, pour mettre en œuvre ses abstractions, ELECTRA utilise une plate-forme de services répartis existante, telle que ISIS ou HORUS [Renesse 94]. Selon les concepteurs d'ELECTRA, l'utilisation d'une plate-forme particulière ne demande que l'écriture d'un **adaptateur** (*adaptor*), un module d'interface entre la machine virtuelle et la plate-forme. Cette modularité est certainement le meilleur atout de ELECTRA.

P3200

P3200 [Eychenne 93] est un système de supervision de processus industriels conçu en termes d'objets et développé en Objective-C [Pinson 91]. Pour garantir la tolérance aux fautes du système, les objets critiques sont dupliqués activement. La duplication est basée sur des groupes d'objets réalisés à partir des groupes de processus fournis par ISIS. Les seuls objets dupliqués sont des s-composants. Comme dans les deux exemples précédents, la sémantique des groupes d'objets suit scrupuleusement celle des groupes de processus définie dans ISIS. Par conséquent, la pluralité n'est encapsulée que partiellement. Comme seule la duplication active est considérée, le problème de l'encapsulation des stratégies de duplication n'est pas abordé.

3.6 Conclusion

Résumé

L'approche orientée-objets consiste à concevoir un système informatique comme un ensemble d'objets communiquant à l'aide d'invocations. Elle est fondée sur la notion de type abstrait et consiste en quatre concepts principaux: la classe, l'instanciation, l'encapsulation et l'héritage.

La notion d'objet est particulièrement adaptée à la conception de systèmes répartis. Cette adéquation provient essentiellement de la propriété d'encapsulation des objets et de leur mode d'interaction par invocations, qui correspond exactement au modèle client-serveur. L'intérêt d'adopter une approche orientée-objets pour la conception d'un système réparti, réside surtout dans les possibilités offertes par cette approche pour la réutilisabilité du logiciel. Cet aspect est particulièrement important dans un système réparti car les problèmes les plus complexes (comme la tolérance aux fautes par duplication) et leurs solutions sont souvent indépendants du domaine d'application du système. Par conséquent, les solutions peuvent être avantageusement réutilisées d'un système réparti à l'autre.

La modélisation d'un objet dupliqué consiste à résoudre le problème de l'encapsulation de la duplication. Ce problème n'est pas trivial. Il consiste à encapsuler la pluralité, et à encapsuler les stratégies de duplication. Encapsuler la pluralité consiste à cacher, aux clients et aux serveurs d'un objet dupliqué O, le fait que O est un ensemble de copies O_i, localisées sur des nœuds distincts. Encapsuler les stratégies de duplication consiste à cacher, aux clients et aux serveurs d'un objet dupliqué O, le protocole utilisé pour communiquer avec les copies de O.

Comme le montre ce chapitre, de nombreux auteurs se sont intéressés à la modélisation d'objets dupliqués. Certains se sont consacrés uniquement à l'encapsulation de la pluralité, alors que d'autres ont aussi essayer d'encapsuler les stratégies de duplication. Les solutions proposées sont souvent intéressantes mais incomplètes. L'étude de ces solutions permet cependant de mieux cerner les problèmes posés par la prise en compte de la duplication dans le modèle objet.

Commentaires

Ce chapitre a présenté le second volet du contexte dans lequel s'inscrit ce travail de thèse. Il a permis notamment d'isoler les problèmes qui sont étudiés en détails au chapitre suivant. De plus, l'étude des travaux effectués par d'autres auteurs ont inspiré les solutions décrites au chapitre 5.

Chapitre 4

L'invocation entre objets dupliqués: analyse et modélisation

4.1 Introduction

Ce chapitre analyse les problèmes posés par l'invocation entre objets dupliqués et propose une modélisation originale de l'invocation entre objets dupliqués.

Présentation générale

L'invocation entre objets dupliqués pose le problème de *l'encapsulation de la duplication*. Cette appelation générique regroupe les difficultés liées à l'intégration de la notion de duplication avec celle d'invocation, et les difficultés liées à la prise en compte des défaillances.

Un objet dupliqué O peut être défini comme un *groupe* de copies O_i dont la cohérence est gérée par la stratégie de duplication de O. La notion d'invocation définit un schéma de communication "requête-réponse" entre *un* client et *un* serveur. Elle ne peut donc pas être utilisée directement pour exprimer la communication entre un *groupe* de copies clientes et un *groupe* de copies serveurs. Pourtant, si l'on considère que la duplication d'un objet O est une caractéristique de cet objet (au même titre que la persistance, par exemple), il est légitime de vouloir utiliser la notion d'invocation pour exprimer la communication entre objets dupliqués.

Pour atteindre cet objectif, on cherche à encapsuler la duplication c.-à-d. à rendre la duplication de tout objet O complètement transparente aux objets communiquant avec O. Encapsuler la duplication d'un objet O consiste à encapsuler la pluralité de O et la stratégie de duplication de O.

Encapsuler la pluralité de O consiste à rendre transparent, aux interlocuteurs de O, le fait que O n'est pas un seul objet mais un groupe d'objets O_i, répartis sur plusieurs nœuds. L'encapsulation de la pluralité permet de garantir que le schéma de communication entre O et ses interlocuteurs, est indépendant de la composition du groupe des O_i.

Encapsuler la stratégie de duplication de O consiste à rendre transparent, aux interlocuteurs de O, le protocole utilisé pour maintenir la cohérence des O_i. L'encapsulation de la stratégie de duplication de O permet de garantir que le schéma de communication entre O et ses interlocuteurs, est indépendant du protocole spécifique à chaque stratégie de duplication. De cette

façon, le choix de la stratégie de duplication pour un objet O n'a pas d'influence sur la façon de communiquer avec O.

La duplication du client et du serveur vise à garantir le bon déroulement d'une invocation malgré l'occurence de défaillances affectant les copies du serveur et/ou du client[1]. Par conséquent, il est crucial de prendre en compte les défaillances dans la modélisation du problème de l'encapsulation de la duplication.

Dans un système réparti asynchrone, l'absence de bornes sur les délais de communication ne permet pas de détecter avec certitude les défaillances des objets. Il est alors nécessaire d'utiliser des abstractions (par ex. les groupes d'objets et les détecteurs de défaillances) qui expriment des informations *approximatives* sur les défaillances.

Organisation

Le reste du chapitre est organisé comme suit. La section 4.2 présente les hypothèses sur lesquelles se base cette étude. La section 4.3 analyse le problème de l'encapsulation de la duplication et propose une spécification de l'invocation entre objets dupliqués, basée sur une modélisation symétrique. La section 4.4 étend cette spécification afin de prendre en compte les défaillances. La section 4.5 applique la spécification aux quatre stratégies de duplication, présentées au chapitre 2. La section 4.6 conclut le chapitre.

4.2 Hypothèses de base

La section 4.2.1 présente les hypothèses relatives au système réparti. La section 4.2.2 présente les hypothèses relatives aux objets.

4.2.1 Hypothèses relatives au système réparti

Processus, canaux et objets

Cette étude s'applique à des objets dupliqués dans le contexte d'un système réparti asynchrone (cf. section 2.3.1). Le système réparti est modélisé par un ensemble de processus s'exécutant sur des nœuds distincts. Chaque paire de processus est connectée par un canal fiable.

L'espace mémoire de chaque processus du système contient des objets. On appelle **processus hôte** *(host process)* de l'objet O, le processus qui contient O dans son espace mémoire.

Défaillance d'un processus

Les processus sont uniquement affectés par des défaillances par arrêt. Un processus défaillant est un processus qui s'arrête de fonctionner de façon impromptue. La défaillance d'un processus est définitive: un processus qui redémarre est considéré comme un nouveau processus.

[1] Jusqu'à un certain point déterminé par la stratégie de duplication, le taux de duplication, le type de défaillances et la qualité des informations sur les défaillances.

Les fautes pouvant causer la défaillance d'un processus sont de nature logicielle (par ex. un "bug" dans le programme du processus) ou de nature matérielle (par ex. une panne d'un composant électronique constituant le nœud).

Défaillance d'un canal

Les canaux sont affectés uniquement par des défaillances par omission. Un canal défaillant perd les messages qu'il est chargé de transporter. Contrairement aux défaillances affectant les processus, les défaillances affectant les canaux sont temporaires: le fonctionnement d'un canal alterne entre un comportement correct et un comportement défaillant.

Les fautes pouvant causer la défaillance par omission d'un canal sont de nature matérielle (par ex. une panne physique du réseau: on suppose dans ce cas que la panne est réparée dans un délai fini) ou de nature logicielle (par ex. saturation des tampons stockant les messages émis ou reçus).

Défaillance d'un objet

Soient un objet O, et son processus hôte p. La défaillance de O *équivaut* à la défaillance de p. Si p est défaillant, alors tout objet O résidant dans l'espace mémoire de p est défaillant. Réciproquement, si O est défaillant, alors le processus p est défaillant.

4.2.2 Hypothèses relatives aux objets

Désignation des objets

Soient O un objet dupliqué et $copies(O) = \{O_1, O_2, ..., O_k\}$ l'ensemble des copies de l'objet O. Le processus hôte de chaque copie de O est distinct de celui des autres copies. On assure ainsi que les copies de O défaillent indépendemment les unes des autres.

L'identificateur O désigne un objet logique (ou conceptuel) dont l'état est représenté par les objets physiques (ou réels) appartenant à $copies(O)$. L'identificateur O_i désigne un objet physique modélisant une copie de l'objet logique O.

Les identificateurs d'objets (logiques et physiques) sont uniques et globaux à tout le système réparti. À partir de l'identificateur d'un objet logique, il est possible d'obtenir la liste des identificateurs des objets physiques correspondants. Inversement, à partir de l'identificateur d'un objet physique, il est possible de retrouver l'identificateur de l'objet logique correspondant.

L'invocation d'un objet dupliqué O consiste en l'invocation d'une ou plusieurs copies de O. Pour invoquer O, il suffit de connaître l'identificateur O, à partir duquel on obtient les identificateurs des copies de O que l'on souhaite invoquer.

Invocations, requêtes, réponses et messages

Une invocation a été définie à la section 2.4.1 comme une interaction requête-réponse entre un client et un serveur. Une invocation comporte trois phases:

1. transmission de la requête: le client transmet la requête au serveur;

2. traitement de la requête: le serveur traite la requête et produit une réponse;

3. transmission de la réponse: le serveur transmet la réponse au client.

Une requête d'invocation est constituée par:

- un identificateur unique d'invocation;
- l'identificateur de l'objet client émettant la requête;
- l'identificateur de l'objet serveur auquel la requête est destinée;
- le nom et les paramètres de l'opération que le serveur doit exécuter.

Une réponse à une invocation est constituée par:

- l'identificateur unique de l'invocation;
- la valeur de la réponse.

Lorsque des objets dupliqués interagissent, pour une même invocation, il peut exister plusieurs exemplaires de la requête et/ou de la réponse. Par conséquent, on adoptera les notations suivantes. Soient C et S deux objets dupliqués tels que C invoque S. On désigne par α la requête d'invocation et par β la réponse.

Si *plusieurs* copies de C invoquent S, chaque copie de S reçoit *plusieurs* exemplaires de la requête α. Chaque exemplaire de α est identique. Lorsque c'est nécessaire, on distinguera l'exemplaire de la requête α envoyé par la copie C_i, en le notant $\alpha(C_i)$. De façon analogue, on notera $\beta(S_j)$ l'exemplaire de la réponse β envoyé par la copie S_j.

Un exemplaire d'une requête (ou d'une réponse) est un message transmis par un objet émetteur vers un objet destinataire. La transmission d'un message comporte les étapes suivantes:

1. émission du message: l'objet émetteur confie le message au canal en utilisant une primitive de communication;

2. transport du message: le canal achemine le message vers le nœud de l'objet destinataire;

3. réception du message: le message est reçu sur le nœud du destinataire mais celui-ci n'a pas encore connaissance du message;

4. délivrance du message: le message est remis à l'objet destinataire.

La distinction entre la réception et la délivrance d'un message permet d'intercaler un traitement éventuel (par ex. ignorer certains messages, traiter les messages dans un ordre différent de l'ordre de réception, etc.) entre le moment où un message est reçu sur un nœud, et le moment où le message est traité par son destinataire.

4.3 Encapsulation de la duplication

Cette section analyse le problème de l'encapsulation de la duplication *sans* considérer les défaillances[2]. La section 4.3.1 étudie quelques exemples d'invocation afin d'identifier précisément les problèmes. Ces exemples sont basés sur la modélisation asymétrique usuelle de l'invocation. La section 4.3.2 propose une classification de ces problèmes et montre que cette modélisation ne permet pas d'exprimer complètement le problème de l'encapsulation de la duplication. Par conséquent, la section 4.3.3 propose une modélisation symétrique comme alternative. La section 4.3.4 présente une spécification, basée sur cette modélisation symétrique.

4.3.1 Étude de quelques exemples

L'étude de quelques exemples permet de mieux cerner le problème de l'encapsulation de la duplication. Ces exemples illustrent l'invocation d'un serveur S par un client C. On considère trois cas:

1. seul le serveur S est dupliqué;
2. seul le client C est dupliqué;
3. le client C et le serveur S sont tous deux dupliqués.

Dans chacun des cas, la stratégie de duplication de C et/ou de S est soit la duplication active, soit la duplication passive. Ces deux stratégies définissent chacune une approche caractéristique de la duplication, et suffisent pour illustrer les problèmes. En effet, la duplication semi-active et la duplication coordinateur-cohorte sont des stratégies hybrides, dérivées des deux autres stratégies.

Duplication du serveur

La figure 4.1 présente deux exemples correspondant au premier cas: seul le serveur S est dupliqué et il comporte trois copies S_1, S_2, S_3.

Duplication active La figure 4.1-a correspond au cas où S est dupliqué activement. Elle illustre le principe de la duplication active, énoncé à la section 2.5.2:

- délivrance des requêtes: *toutes les copies délivrent la même séquence*[3] *de requêtes*;
- traitement des requêtes: *toutes les copies traitent de manière déterministe*[4];
- émission des réponses: *toutes les copies émettent la même séquence de réponses.*

[2] Les problèmes liés aux défaillances sont traités à la section 4.4.

[3] Une séquence est un ensemble totalement ordonné. Les copies délivrent les *mêmes* requêtes dans le *même* ordre.

[4] Un traitement déterministe produit toujours le même résultat à partir des mêmes données. Par conséquent, pour une requête donnée, toutes les copies produisent la même réponse

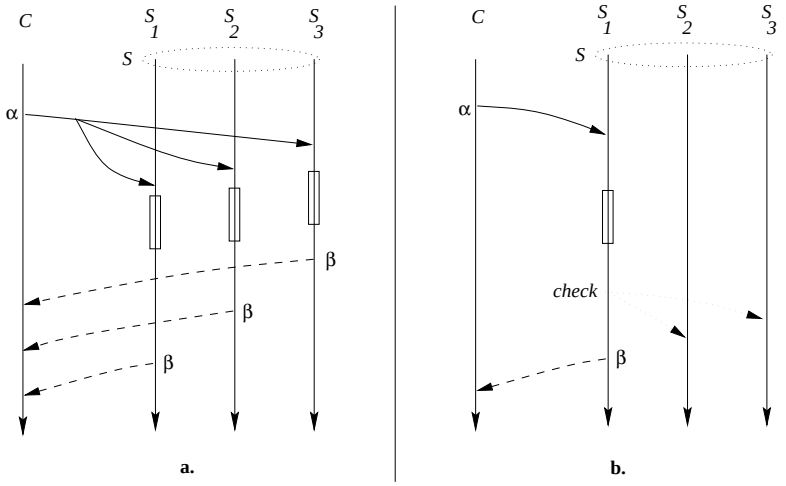

Figure 4.1: Duplication du serveur: a. active, b. passive

Lorsqu'il invoque le serveur S, le client C transmet la requête α (flèche en trait continu) à *toutes* les copies de S. Le principe de la duplication active précise que toutes les copies de S doivent délivrer la même séquence de requêtes. Par conséquent, le client C transmet la requête α à l'aide d'un multicast garantissant l'ordre total.

Chaque S_i effectue le traitement (rectangle transparent) de la requête α, construit un exemplaire de la réponse β, puis transmet cet exemplaire (flèche en trait discontinu) au client C. Par conséquent, le client C reçoit trois exemplaires de la réponse β.

Duplication passive La figure 4.1-b correspond au cas où S est passivement dupliqué. Elle illustre le principe de la duplication passive, énoncé à la section 2.5.3:

- délivrance des requêtes: *la copie primaire est la seule à délivrer les requêtes*;
- traitement des requêtes: *la copie primaire est la seule à traiter les requêtes*;
- émission des réponses: *la copie primaire est la seule à émettre les réponses*.

Sur cet exemple, la copie primaire est S_1. Lorsqu'il invoque le serveur S, le client C transmet la requête α à une *seule* copie, la copie primaire S_1. La copie primaire traite α, construit un exemplaire de la réponse β, et le transmet au client C. Par conséquent, le client C reçoit un seul exemplaire de la réponse β.

Analyse de la duplication du serveur La duplication du serveur pose les problèmes suivants:

- la désignation du serveur;
- l'ordre de délivrance des requêtes;
- le nombre variable de réponses.

Le problème de la désignation du serveur peut être résumé par la question suivante: à quelles copies de S, le client C doit-il transmettre la requête α? La réponse à cette question est conditionnée par la stratégie de duplication de S, car c'est elle qui détermine l'ensemble des copies de S qui doivent délivrer α. Si S est dupliqué activement, toutes les copies de S doivent délivrer α. Si S est dupliqué passivement, seule la copie primaire doit délivrer α. Lorsqu'il transmet la requête au serveur S, le client C doit interpréter l'identificateur de l'objet logique S, en fonction de la stratégie de duplication de S. Si S est dupliqué activement, C doit considérer toutes les copies de S. Si S est dupliqué passivement, C doit considérer uniquement la copie primaire de S.

Le problème de l'ordre de délivrance des requêtes peut être résumé par la question suivante: avec quelle garantie d'ordre, les copies de S doivent délivrer les requêtes? Si S est dupliqué activement, les copies de S doivent délivrer toutes les requêtes dans le même ordre. Si S est dupliqué passivement, le problème ne se pose pas car la copie primaire est la seule à recevoir les requêtes. Lorsqu'il transmet la requête au serveur S, le client C doit choisir la primitive de communication en fonction de la stratégie de duplication de S.

Le problème du nombre variable de réponses peut être résumé par la question suivante: combien d'exemplaires de la réponse β le client C reçoit-il? La réponse à cette question est conditionnée par la stratégie de duplication de S, car c'est elle qui détermine l'ensemble des copies de S qui transmettent un exemplaire de β à C. Si S est dupliqué activement, chaque S_i transmet un exemplaire de la réponse au client C. Si S est dupliqué passivement, seule la copie primaire transmet la réponse β au client C. Quelque soit le nombre d'exemplaires de β qu'il reçoit, le client C ne doit en délivrer qu'un seul. Ceci en vertu du principe de l'invocation: à *une* requête correspond *une* réponse. Lorsqu'il reçoit un exemplaire de la réponse β, le client C doit interpréter cette réception en fonction de la stratégie de duplication de S. Si S est dupliqué activement, C doit vérifier qu'il n'a pas déjà délivré un exemplaire de β. Si S est dupliqué passivement, le client C peut délivrer β immédiatement.

Duplication du client

La figure 4.2 présente deux exemples correspondant au second cas: seul le client C est dupliqué et il comporte trois copies C_1, C_2, C_3. À la section 2.5, le principe de chaque stratégie de duplication n'a été défini que pour des objets serveurs. Pour appliquer ces stratégies à des objets clients, il suffit d'extrapoler chaque définition en considérant le cas d'un cs-composant (i.e. un objet à la fois client et serveur)

Sur la figure 4.2, il faut voir le client C comme un cs-composant. Le client C invoque S au cours du traitement correspondant à l'invocation de C par O, un objet tiers, client de C, et non représenté sur la figure.

Duplication active La figure 4.2-a correspond au cas où C est dupliqué activement. Selon le principe de la duplication active, chaque C_i traite la requête envoyée par O. Par conséquent, chaque C_i transmet un exemplaire de la requête α au serveur S. Ce dernier reçoit trois exemplaires de la requête α! Si S délivre et traite chaque exemplaire de α, son état risque de devenir incohérent dans le cas où le traitement de α n'est pas idempotent. Si S ne délivre que le premier exemplaire qu'il reçoit (c.-à-d. $\alpha(C_1)$), les copies C_2 et C_3 risquent d'attendre indéfiniment une réponse. En effet, les réponses ne sont transmises qu'aux émetteurs des requêtes: ce principe caractérise la modélisation asymétrique de l'invocation.

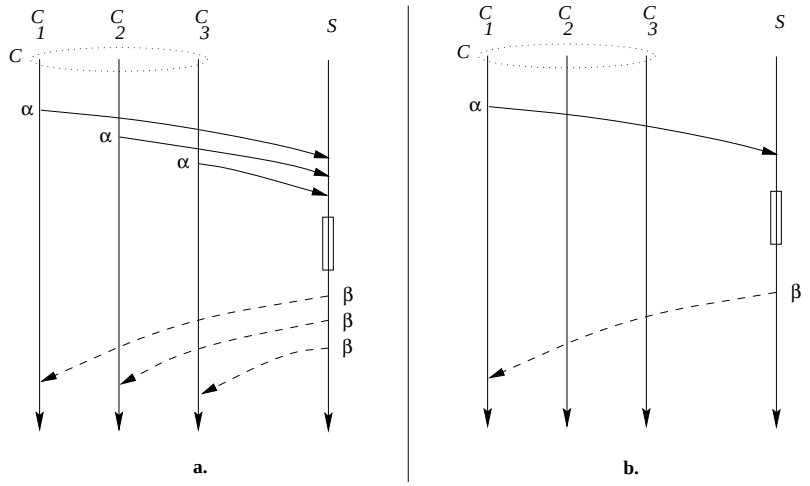

Figure 4.2: Duplication du client: a. active, b. passive

Le problème ci-dessus peut être évité si S_1 délivre successivement chaque exemplaire de α, mais ne traite que le premier exemplaire. Sur la figure 4.2-a, S délivre et traite $\alpha(C_1)$. Au terme de ce traitement, S transmet la réponse β à C_1. Lorsqu'il délivre $\alpha(C_2)$ (resp. $\alpha(C_3)$), S ne refait pas le traitement. Il transmet directement un exemplaire de β à C_2 (resp. C_3).

Duplication passive La figure 4.2-b correspond au cas où C est dupliqué passivement. La copie primaire est C_1. Selon le principe de la duplication passive, seule la copie primaire traite la requête envoyé par O. Par conséquent, la copie C_1 transmet la requête α au serveur S. Ce dernier traite α, construit la réponse β, et transmet β à C_1.

Analyse de la duplication du client La duplication du client pose le problème du nombre variable de requêtes.

Ce problème peut être résumé par la question: combien d'exemplaires de α le serveur S reçoit-il? La réponse à cette question est conditionnée par la stratégie de duplication de C, car c'est elle qui détermine l'ensemble des copies de C qui transmettent un exemplaire de α à S. Si C est dupliqué activement, chaque C_i transmet la requête α au serveur S. Si C est dupliqué passivement, seule la copie primaire transmet la requête α au serveur S. Dans tous les cas, S ne doit traiter α qu'une seule fois afin d'éviter les incohérences si le traitement de α n'est pas idempotent. Lorsqu'il reçoit un exemplaire de α, le serveur S doit interprèter cette réception en fonction de la stratégie de duplication de C. Si C est dupliqué activement, S doit vérifier qu'il n'a pas déjà traité un exemplaire de α. Si S est dupliqué passivement, le serveur S peut traiter α immédiatement.

Duplication du client et du serveur

La figure 4.3 présente quatre exemples correspondant au troisième cas (le cas général): le client C et le serveur S sont tous deux dupliqués. Cette figure permet de récapituler les problèmes évoqués précédemment en illustrant les quatre combinaisons possibles lorsque l'on considère

la duplication active et la duplication passive. Le tableau 4.1 permet de retrouver la figure correspondant à chaque combinaison.

client C	serveur S	
	duplication active	duplication passive
duplication active	figure 4.3-a	figure 4.3-b
duplication passive	figure 4.3-c	figure 4.3-d

Tableau 4.1: Duplication active, duplication passive: les quatre combinaisons possibles

Désignation du serveur Le problème de la désignation du serveur dépend de la stratégie de duplication du serveur S. Selon que le serveur S est dupliqué activement (cf. figure 4.3-a et figure 4.3-c) ou passivement (cf. figure 4.3-b et figure 4.3-d), le client C doit transmettre la requête à toutes les copies de S, ou bien uniquement à la copie primaire S_1.

Ordre de délivrance des requêtes Le problème de l'ordre de délivrance des requêtes dépend de la stratégie de duplication du serveur S. Si le serveur S est dupliqué activement (cf. figure 4.3-a et figure 4.3-c), le client C utilise une primitive de communication garantissant l'ordre total pour transmettre les requêtes à toutes les copies de S. Si le serveur S est dupliqué passivement (cf. figure 4.3-b et figure 4.3-d), le client C transmet les requêtes uniquement à la copie primaire, et le problème ne se pose pas.

Nombre variable de réponses Le problème du nombre variable de réponses dépend de la stratégie de duplication du serveur S. Selon que le serveur S est dupliqué activement (cf. figure 4.3-a et figure 4.3-c) ou passivement (cf. figure 4.3-b et figure 4.3-d), le client C reçoit plusieurs exemplaires de la réponse ou bien un seul.

Nombre variable de requêtes Le problème du nombre variable de requêtes dépend de la stratégie de duplication du client C. Selon que le client C est dupliqué activement (cf. figure 4.3-a et figure 4.3-b) ou passivement (cf. figure 4.3-c et figure 4.3-d), le serveur S reçoit plusieurs exemplaires de la requête ou bien un seul.

4.3.2 Classification des problèmes rencontrés

Le tableau 4.2 propose une classification de chaque problème rencontré selon deux critères:

- la phase de l'invocation durant laquelle le problème se pose;
- le contexte plus général auquel le problème est rattaché.

Cette classification met en évidence l'asymétrie entre la transmission de la requête et la transmission de la réponse. En d'autres termes, la transmission de la réponse n'est pas modélisée comme une phase symétrique de la transmission de la requête. La grande majorité des travaux

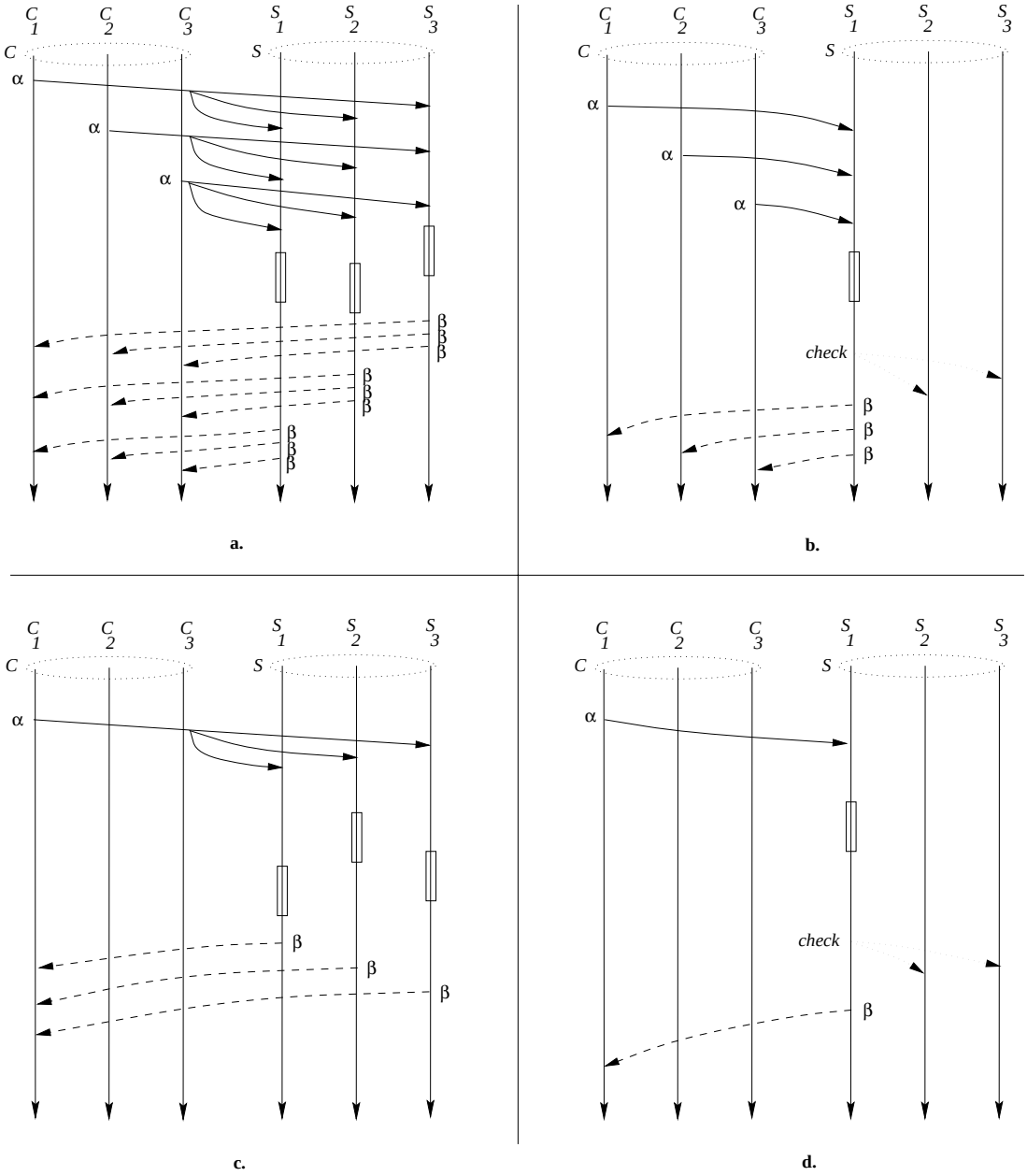

Figure 4.3: Duplication du client et du serveur: a. active-active, b. active-passive, c. passive-active, d. passive-passive

	encapsulation de la pluralité	encapsulation des stratégies de duplication
transmission de la requête	désignation du serveur	nombre variable de requêtes ordre de réception des requêtes
transmission de la réponse		nombre variable de réponses

Tableau 4.2: Modélisation asymétrique: classification des problèmes liés à l'encapsulation de la duplication

décrits dans la littérature sont basés sur cette modélisation asymétrique de l'invocation entre objets dupliqués.

Le principal inconvénient de cette approche est qu'elle ne permet pas de poser tous les problèmes relatifs à l'encapsulation de la duplication. Pour s'en convaincre, il suffit de mettre en regard les problèmes mis en évidence sur les exemples, avec les deux volets de l'encapsulation de la duplication, à savoir l'encapsulation de la pluralité et l'encapsulation des stratégies de duplication.

Encapsulation de la pluralité

Encapsuler la pluralité d'un objet dupliqué O consiste à cacher aux objets communiquant avec O, le fait que O est en réalité un *groupe* d'objets répartis sur des nœuds distincts.

Soit I, un objet (dupliqué ou non) participant (en tant que client ou serveur) à une invocation avec l'objet dupliqué O. L'objet I doit percevoir le groupe des copies de O comme un tout. La perception que l'objet I a de l'objet dupliqué O est déterminée par le moyen que I utilise pour désigner O. Si ce moyen est indépendant de la composition du groupe des copies de O, et de la stratégie de duplication de O, la pluralité de O est encapsulée. Par conséquent, le problème de l'encapsulation de la pluralité peut être ramené au problème de la désignation d'un objet dupliqué.

Si l'objet dupliqué O est serveur, pour transmettre une requête à O, le client I doit pouvoir désigner l'ensemble des copies de O qui délivrent les requêtes. Cependant, cet ensemble varie selon la composition du groupe de copies de O, et selon la stratégie de duplication de O. Il s'agit là du problème de la désignation du serveur. Par conséquent, encapsuler la pluralité de O correspond à résoudre le problème de la désignation du serveur.

Si l'objet dupliqué O est client, le serveur I transmet *individuellement* un exemplaire de la réponse à chaque copie de O. Il n'est donc pas possible, dans ce cas, d'envisager l'encapsulation de la pluralité de O.

Encapsulation des stratégies de duplication

Encapsuler la stratégie de duplication d'un objet dupliqué O consiste à cacher, aux objets communiquant avec O, le protocole particulier qu'il faut utiliser pour communiquer avec les copies de O. Ce protocole de communication est défini par la stratégie de duplication dans le but de maintenir la cohérence de l'état des copies.

Soit I un objet (dupliqué ou non) participant (en tant que client ou serveur) à une invocation

avec l'objet O. L'objet I doit percevoir la communication avec O comme *l'échange de deux messages*: l'un transportant la requête d'invocation et l'autre transportant la réponse à cette invocation. La perception que l'objet I a de la communication avec O est déterminée par:

- la primitive de communication que I utilise pour transmettre des messages à O;
- l'ensemble des messages (i.e. requêtes et réponses), en provenance de O, que I reçoit.

Si le choix de la primitive de communication est transparent pour I, et si les messages en provenance de O sont filtrés à l'insu de I, la stratégie de duplication de O est encapsulée.

Si l'objet dupliqué O est serveur, le choix de la primitive de communication, que I utilise pour transmettre une requête à O, est déterminé par l'ordre de délivrance des requêtes aux copies de O. Cet ordre dépend de la stratégie de duplication de O. Il s'agit là du problème de l'ordre de délivrance des requêtes. Le nombre de réponses que I reçoit varie en fonction de la stratégie de duplication de O. Il s'agit là du problème du nombre variable de réponses.

Si l'objet dupliqué O est client, le serveur I transmet *individuellement* un exemplaire de la réponse à chaque copie de O. Le problème du choix de la primitive de communication ne se pose donc pas. Par contre, le nombre de requêtes que le client O transmet au serveur I varie en fonction de la stratégie de duplication de O. Il s'agit là du problème du nombre variable de requêtes.

4.3.3 Modélisation symétrique de l'invocation entre objets dupliqués

Cette section présente la modélisation symétrique de l'invocation entre objets dupliqués, une alternative originale à la modélisation asymétrique.

Présentation de la modélisation symétrique

La figure 4.4 reprend le cas d'un client C et d'un serveur S, tous deux dupliqués activement. Elle permet de présenter la modélisation symétrique (a.) par opposition à la modélisation asymétrique (b.). La différence entre les deux modélisations réside dans le mode de transmission de la réponse.

La modélisation symétrique considère la transmission de la réponse et la transmission de la requête comme deux instances du même problème: la transmission d'un message à un objet dupliqué activement. En effet, la figure 4.4-a montre que chaque S_i diffuse la réponse β à toutes les copies du client C.

Par contre, la modélisation asymétrique considère la transmission de la réponse comme un cas particulier de communication. En effet, la figure 4.4-b montre que chaque S_i transmet successivement un exemplaire de la réponse β à chaque copie C_j.

La modélisation asymétrique suppose que la réponse à une invocation est toujours transmise en suivant exactement le même chemin inverse de la requête. Ceci revient à faire l'hypothèse que la communication entre le client et le serveur est assurée par des mécanismes réalisant une sémantique d'appel de procédure à distance.

La modélisation symétrique ne fait pas cette hypothèse: la requête et la réponse sont vues comme des messages indépendants. L'intérêt de cette approche est qu'elle permet de prendre en compte tous les problèmes relatifs à l'encapsulation de la duplication.

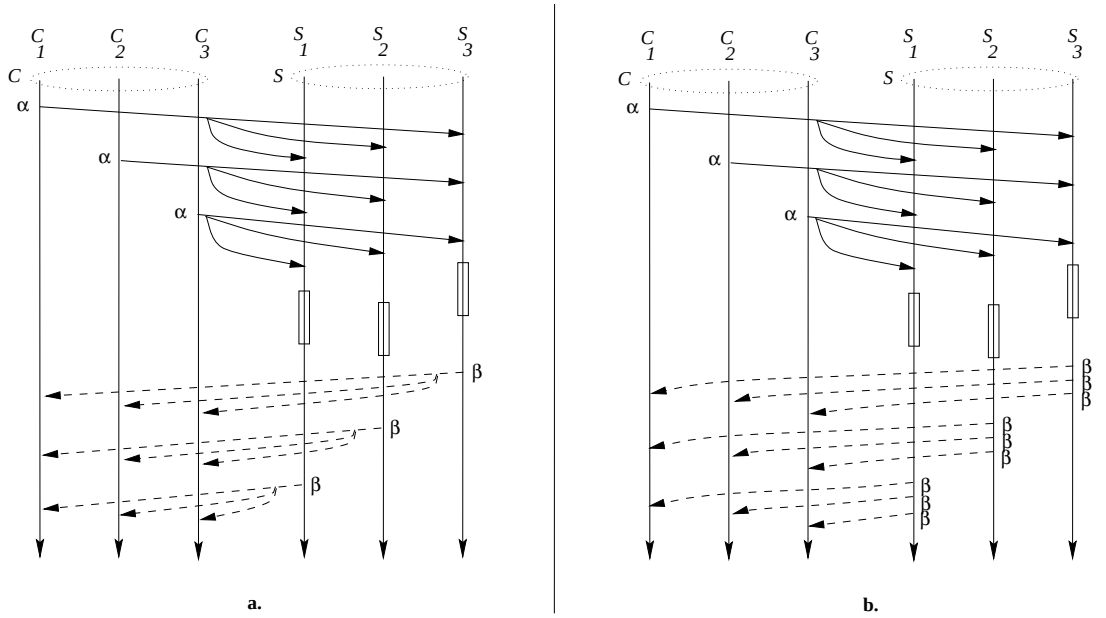

Figure 4.4: Duplication active du client et du serveur: a. modélisation symétrique, b. modélisation asymétrique

Encapsulation de la pluralité

La modélisation symétrique considère la transmission de la réponse comme une phase symétrique de la transmission de la requête. Par conséquent, l'ensemble des copies du client qui délivrent les réponses varie, en fonction de la composition du groupe de copies du client, et en fonction de la stratégie de duplication du client. Il s'agit là du problème de la désignation du client. Résoudre ce problème correspond à encapsuler la pluralité du client.

Sur la figure 4.4-a, comme C est dupliqué activement, toutes les copies de C doivent délivrer la réponse β. Par conséquent, chaque copie S_j transmet un exemplaire de β à toutes les copies de C.

Contrairement à la modélisation symétrique, la modélisation asymétrique ne permet pas de poser le problème de l'encapsulation d'un client dupliqué. Sur la figure 4.4-b, chaque copie S_i doit pouvoir désigner *individuellement* chaque copie C_j, afin que S_i puisse transmettre la réponse β à C_j. Ce schéma de communication ne permet pas aux copies de S de percevoir le client dupliqué C comme un seul objet.

Encapsulation des stratégies de duplication

La modélisation symétrique considère la transmission de la réponse comme une phase symétrique de la transmission de la requête. Par conséquent, l'ordre de délivrance des réponses aux copies du client varie en fonction de la stratégie de duplication du client. Cet ordre détermine le choix de la primitive de communication que le serveur utilise pour transmettre les réponses aux copies du client. Il s'agit là du problème de l'ordre de délivrance des réponses.

Sur la figure 4.4-a, comme C est dupliqué activement, chaque S_j utilise une primitive de communication garantissant l'ordre total pour transmettre la réponse β à toutes les copies de C. L'ordre total des réponses sert à assurer que toutes les copies de C délivrent la même séquence de réponses, dans le cas où elles attendent les réponses correspondant à plusieurs requêtes.

Synthèse

Le tableau 4.3 fait la synthèse des problèmes liés à l'encapsulation de la duplication lorsque la modélisation symétrique est utilisée. Les différences avec le tableau 4.2 sont en italique.

	encapsulation de la pluralité	encapsulation des stratégies de duplication
transmission de la requête	désignation du serveur	nombre variable de requêtes ordre de délivrance des requêtes
transmission de la réponse	*désignation du client*	nombre variable de réponses *ordre de délivrance des réponses*

Tableau 4.3: Modélisation symétrique: classification des problèmes liés à l'encapsulation de la duplication

Aux problèmes qui se posaient déjà avec la modélisation asymétrique, s'ajoutent le problème de la désignation du client et le problème de l'ordre de délivrance des réponses. Le premier problème correspond à l'encapsulation de la pluralité du client qui ne pouvait pas être prise en compte avec la modélisation asymétrique. Le second problème découle du premier: si les réponses sont délivrées par plusieurs copies du client, il est nécessaire de se préoccuper de l'ordre de délivrance des réponses à ces copies.

Contrairement à modélisation asymétrique, la modélisation symétrique permet de prendre en compte complètement le problème de l'encapsulation de la pluralité. Par conséquent, cette approche est adoptée dans toute la suite: de la spécification des problèmes à la mise en œuvre des solutions (cf. chapitre 6), en passant par la conception des solutions (cf. chapitre 5).

Il est important de noter que si l'on considère uniquement la duplication de s-composants, le problème de l'encapsulation de la pluralité du client ne se pose pas. Dans ce cas, la modélisation asymétrique suffit. C'est probablement pour cette raison que de nombreux auteurs ont choisi d'utiliser la modélisation asymétrique.

4.3.4 Spécification de l'invocation entre objets dupliqués

Cette section présente une spécification de l'invocation entre objets dupliqués, basée sur une modélisation symétrique. Cette spécification est générique, dans le sens où elle est applicable aux quatre stratégies de duplication considérées dans cette thèse.

Chaque phase de l'invocation est spécifiée en termes de propriétés paramétrables, dont les paramètres permettent de prendre en compte les différentes stratégies de duplication.

Transmission de la requête

Lorsqu'un client C transmet une requête α à un serveur S, les questions suivantes se posent:

- *désignation du serveur*: quelles copies de S doivent délivrer la requête α?
- *ordre de délivrance de la requête*: quelle garantie d'ordre doit être assurée sur la délivrance de α?
- *nombre variable de requêtes*: quelles copies de C émettent un exemplaire de la requête α?

Soient $copies(S)$ l'ensemble des copies de S et $copies(C)$ l'ensemble des copies de C. Ces questions permettent de définir trois paramètres:

- $dst(S)$: le sous-ensemble de $copies(S)$ qui doivent délivrer α;
- $ord(S)$: la garantie d'ordre avec laquelle les éléments de $dst(S)$ doivent délivrer α;
- $src(C)$: le sous-ensemble de $copies(C)$ qui émettent α.

Les valeurs de $dst(S)$ et $ord(S)$ sont déterminées par $str(S)$, la stratégie de duplication de S. La valeur de $src(C)$ est déterminée par $str(C)$, la stratégie de duplication de C.

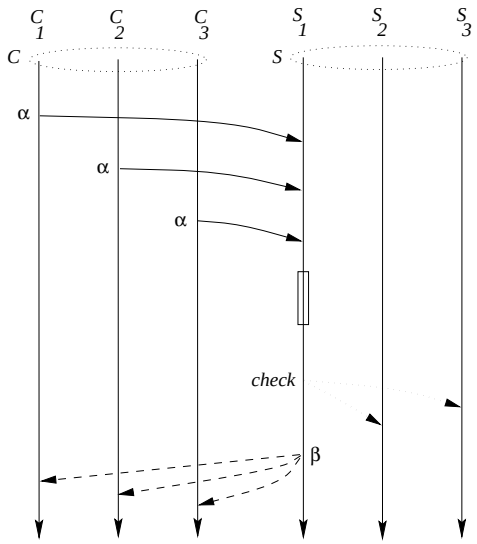

Figure 4.5: Un exemple d'invocation d'un serveur dupliqué passivement par un client dupliqué activement

Sur l'exemple de la figure 4.5, le serveur S est dupliqué passivement tandis que le client C est dupliqué activement. Les valeurs des paramètres pour cet exemple sont:

- $dst(S) = prim(S) = S_1$;
- $ord(S) = FIFO$;

- $src(C) = copies(C) = \{C_1, C_2, C_3\}$

La valeur de $dst(S)$ exprime que seule $prim(S)$, la copie primaire du serveur S, délivre la requête. La valeur de $ord(S)$ exprime que les requêtes sont transmises en suivant l'ordre FIFO. La valeur de $src(C)$ exprime que chaque copie de C émet un exemplaire de la requête.

Ces paramètres permettent de spécifier la transmission d'une requête α à l'aide des propriétés suivantes:

- **Validité:** *Si un élément de $dst(S)$ délivre une requête α, alors un élément de $src(C)$ a émis α.*
- **Intégrité:** *Tout élément de $dst(S)$ délivre α au plus une fois.*
- **Ordre:** *Tout élément de $dst(S)$ délivre α dans l'ordre spécifié par $ord(S)$.*

La propriété d'intégrité garantit que chaque requête n'est pas délivrée plusieurs fois aux copies du serveur. Cette propriété est nécessaire à cause du nombre variable de requêtes que le client C transmet au serveur S. Ce nombre est égal à $\|src(C)\|$, le cardinal de l'ensemble $src(C)$. Si $\|src(C)\| > 1$, chaque élément de $dst(S)$ reçoit *plusieurs* exemplaires de la même requête α. Si le traitement correspondant à cette requête n'est pas idempotent, l'état des éléments de $dst(S)$ devient incohérent.

Traitement de la requête

Lorsqu'un serveur S s'apprête à traiter une requête α, deux questions se posent:

1. quelles copies de S effectuent le traitement de α?
2. si plusieurs copies de S effectuent le traitement, quelle est la copie dont on va considérer la réponse?

La première question permet de définir un nouveau paramètre:

- $cpu(S)$: le sous-ensemble de $dst(S)$ dont chaque élément effectue le traitement de α et produit une réponse β.

La valeur de $cpu(S)$ est déterminée par $str(S)$. La seconde question est réglée par l'hypothèse que tous les éléments de $cpu(S)$ satisfont la propriété suivante:

Déterminisme: *Tout élément de $cpu(S)$ produit la même réponse β comme résultat du traitement de α.*

Dans le cas de la figure 4.5, S est dupliqué passivement et $cpu(S) = prim(S)$. Comme l'ensemble $cpu(S)$ est un singleton, la condition de déterminisme n'est pas nécessaire.

Transmission de la réponse

Lorsqu'un serveur S transmet une réponse β à un client C, les questions suivantes se posent:

- *désignation du client*: quelles copies de C doivent délivrer la réponse β?

- *ordre de délivrance de la réponse*: quelle garantie d'ordre doit être assurée sur la délivrance de β?
- *nombre variable de réponses*: quelles copies de S émettent un exemplaire de la réponse β?

Ces questions sont *exactement* les mêmes que celles concernant la transmission de la requête. Comme la modélisation est symétrique, la spécification de la transmission de la réponse est identique à celle de la transmission de la requête: les deux spécifications utilisent les mêmes paramètres et les mêmes propriétés.

Dans le cas de la figure 4.5, les paramètres ont les valeurs suivantes:

- $dst(C) = copies(C)$;
- $ord(C) = TOTAL$
- $src(S) = prim(S)$.

La valeur de $dst(C)$ exprime que toutes les copies de C délivrent la réponse. La valeur de $ord(C)$ exprime que les réponses doivent être délivrées en respectant l'ordre total. La valeur de $src(C)$ exprime que seule la copie primaire $prim(S)$ transmet la réponse à C.

Synthèse

La stratégie de duplication d'un objet dupliqué O, notée $str(O)$, définit quatre paramètres qui spécifient le comportement de O au cours d'une invocation. Ces paramètres sont: $src(O)$, $dst(O)$, $ord(O)$ et $cpu(O)$.

Lorsqu'un objet dupliqué S est invoqué par un objet dupliqué C, les phases de cette invocation sont paramétrées de la façon suivante:

1. la transmission de la requête est paramétrée par $dst(S), ord(S), src(C)$;
2. le traitement de la requête est paramétré par $cpu(S)$;
3. la transmission de la réponse est paramétrée par $dst(C), ord(C), src(S)$;

4.4 Prise en compte des défaillances

Pour réaliser l'encapsulation de la duplication en dépit de l'occurence de fautes, il est essentiel de prendre en compte les défaillances dans la spécification de l'invocation entre objets dupliqués. La section 4.4.1 décrit les abstractions choisies pour exprimer les informations sur les défaillances. En s'appuyant sur quelques exemples, la section 4.4.2 met en évidence les problèmes que posent cette prise en compte. La section 4.4.3 étend la spécification proposée à la section 4.3.4.

4.4.1 Expression des informations sur les défaillances

L'absence de bornes sur les délais de communication rend impossible la détection précise de la défaillance d'un objet. Les informations sur les défaillances sont nécessairement approximatives. Afin de prendre en compte les défaillances dans la spécification de l'invocation, il est important de pouvoir exprimer précisément les informations sur les défaillances.

Introduction

Soient deux objets C et S évoluant dans un système réparti satisfaisant aux hypothèses énoncées à la section 4.2. On suppose que C invoque S. Tant que C ne reçoit pas de réponse en provenance du serveur S, le client C est dans l'impossibilité de distinguer les cas suivants:

- le serveur S est défaillant (défaillance par arrêt);
- le canal de communication reliant C à S est défaillant (défaillance par omission);
- la réponse (ou même la requête) est encore en transit sur le canal (pas de défaillance).

Pour poursuivre son exécution, le client C doit lever cette ambiguïté en utilisant les informations sur les défaillances dont il dispose. Ces informations sont exprimées à l'aide d'abstractions telles que les détecteurs de défaillances ou les groupes d'objets[5] (cf. section 2.3.3).

Un objet dupliqué = un groupe à partition primaire

L'abstraction choisie dans cette thèse pour exprimer les informations sur les défaillances est le groupe d'objets à partition primaire.

La composition d'un groupe g est généralement appelée vue du groupe g (cf. section 2.3.3). Lorsqu'un membre de g est suspecté, une nouvelle vue du groupe est construite puis transmise à tous les membres corrects de g. Un membre du groupe g n'est informé que des défaillances affectant les membres de g.

La notion de **partition primaire**[6] (*primary partition*) permet d'éviter, dans le cas de la partition du groupe, que deux vues disjointes soient construites. La figure 4.6 illustre le problème.

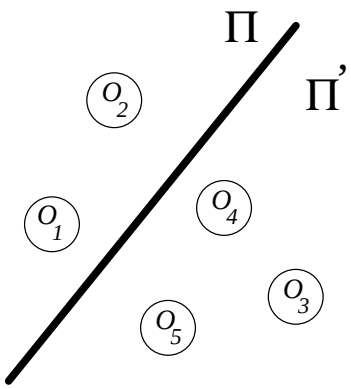

Figure 4.6: Partition d'un groupe de copies

Soit un objet dupliqué O dont l'ensemble des copies est $copies(O) = \{O_1, O_2, O_3, O_4, O_5\}$. Cet

[5] Un groupe d'objets peut être aisément construit à partir du groupe des processus hôtes de ces objets.

[6] On notera au passage que cette désignation n'est pas trés appropriée: le mot "partition" désigne, en anglais comme en français, le fait qu'un ensemble E est subdivisé en sous-ensembles disjoints dont la réunion est égale à E. Il ne désigne pas chacun des sous-ensembles. On préfèrera donc parler de sous-ensemble primaire plutôt que de partition primaire.

ensemble est mis en œuvre par un groupe d'objets dont la vue initiale est $v_0(O) = copies(O)$. L'occurence de défaillances par omission affectant les canaux reliant les processus hôtes des copies de O a provoqué une partition du groupe. Cette partition est constituée par deux sous-ensembles disjoints contenant chacun des copies correctes: $\Pi = \{O_1, O_2\}$ et $\Pi' = \{O_3, O_4, O_5\}$.

Les copies appartenant à Π suspectent les copies appartenant à Π' et réciproquement. Si l'unicité des vues n'est pas garantie, cette situation peut conduire à la construction d'une nouvelle vue v_1 différente dans *chaque* sous-ensemble: dans Π, $v_1(O) = \{O_1, O_2\}$ tandis que dans Π', $v_1(O) = \{O_3, O_4, O_5\}$.

L'unicité des vues est assurée si la nouvelle vue n'est construite que dans un *seul* des sous-ensembles composant la partition. Ce sous-ensemble est appelé sous-ensemble primaire. Le sous-ensemble primaire peut être choisi comme le sous-ensemble contenant une majorité de copies. Ce choix a l'avantage d'être systématique mais l'inconvénient de ne pas être toujours possible (i.e. dans certains cas, aucun sous-ensemble ne contient une majorité de copies).

La majorité est définie en fonction de la vue précédente. Dans le contexte de l'exemple de la figure 4.6, la majorité vaut $(5 + 1)/2 = 3$. La nouvelle vue est alors construite dans Π': $v_1(O) = \{O_3, O_4, O_5\}$.

Les copies minoritaires sont traitées *comme si* elles étaient défaillantes. Lorsqu'une copie défaillante redémarre, elle est considérée comme une nouvelle copie[7]. Son état est initialisé à partir de celui des copies existantes.

Ainsi, les copies appartenant à Π sont bloquées tant que la partition persiste: étant minoritaires, toute participation à une invocation leur est interdite. Lorsque la partition disparaît, chaque copie minoritaire rejoint le groupe des copies majoritaires comme si elle venait de redémarrer.

La notion de groupe d'objets est souvent associée à la communication vue-synchrone (*VSC*) (cf. section 2.3.3). La communication vue-synchrone consiste à synchroniser les messages échangés par les objets avec les vues.

Chaque objet dupliqué est mis en œuvre à l'aide d'un groupe d'objets. Ce choix comporte plusieurs avantages:

- la portée limitée des informations sur les défaillances va dans le sens de l'encapsulation de la pluralité: les copies d'un objet dupliqué appartiennent au même groupe et elles ne sont alors informées que des défaillances de leurs pairs;
- à chaque instant logique (i.e. vue), les copies d'un objet dupliqué connaissent avec précision la liste de leurs pairs;
- la communication vue-synchrone est particulièrement utile pour gérer la cohérence des copies d'un objet dupliqué.

Cependant, lorsque l'on considère *plusieurs* objets dupliqués (i.e. dans le cas d'une invocation), la portée limitée des informations sur les défaillances exprimées par un groupe d'objets devient un handicap. Les exemples décrits à la section 4.4.2 le montrent. Cet inconvénient peut être contourné en construisant un groupe d'objets au-dessus d'un détecteur de défaillances. Lorsque cela est nécessaire (cf. problème 3 décrit à la section 4.4.2), il est alors aisé d'interroger directement le détecteur de défaillances.

[7] La sémantique de défaillances d'un objet suit celle de son processus hôte (cf. section 4.2)

4.4.2 Étude de quelques exemples

Cette section présente deux exemples illustrant des scénarios de défaillances perturbant le déroulement d'une invocation d'un serveur dupliqué S par un client dupliqué C. Le premier exemple considère un client dupliqué passivement et un serveur dupliqué activement. Le second exemple considère un client dupliqué activement et un serveur dupliqué passivement.

L'ensemble des copies de S est noté $copies(S) = \{S_1, S_2, S_3\}$. Cet ensemble est mis en œuvre à l'aide d'un groupe d'objets dont la vue initiale, notée $v_0(O)$, comprend tous les objets appartenant à $copies(S)$. Ainsi, $v_0(S) = \{S_1, S_2, S_3\}$. De même, $v_0(C) = \{C_1, C_2, C_3\}$.

Duplication passive du client et duplication active du serveur

La figure 4.7 illustre deux scénarios de défaillances perturbant la transmission de la requête. Les paramètres spécifiant la transmission de la requête ont les valeurs suivantes:

- $src(C) = prim(C)$;
- $dst(S) = copies(S)$;
- $ord(S) = TOTAL$.

La copie primaire de C (initialement C_1) transmet un exemplaire de la requête α à *toutes* les copies de S, en utilisant un multicast garantissant l'ordre total.

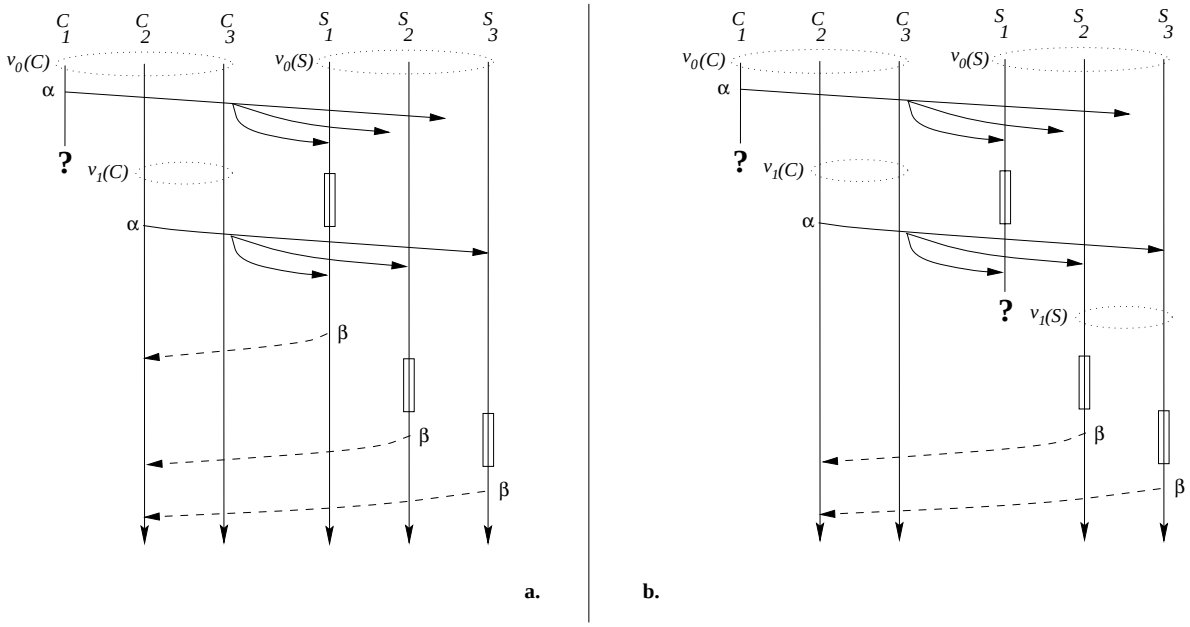

Figure 4.7: Duplication passive du client et duplication active du serveur: deux scénarios de défaillances

Défaillance de la copie primaire du client La figure 4.7-a illustre la défaillance de C_1, pendant la transmission de la requête α. La défaillance d'une copie est représentée par un point d'interrogation en gras.

La défaillance de C_1 pose principalement deux problèmes:

Problème 1. *Certaines copies correctes de S délivrent α alors que d'autres ne délivrent jamais α.*

Soit $\alpha(C_1)$ l'exemplaire de la requête α transmis par la copie C_1. Comme C_1 défaille *pendant* la transmission de $\alpha(C_1)$, la copie S_1 délivre α alors que S_2 et S_3 ne délivrent pas α. La stratégie de duplication active n'est plus respectée et les états des copies de S risquent de diverger.

L'hypothèse de canal fiable (cf. section 2.3.3) ne prévient pas cette situation car la copie C_1 est défaillante. L'acheminement d'un message à travers un canal fiable n'est garanti que si les objets aux deux extrémités du canal sont corrects.

Pour prendre en compte ce cas de défaillance, il faut ajouter une propriété d'agrément à la spécification de la transmission de la requête (cf. section 4.4.3).

Problème 2. *Certaines copies correctes de S délivrent la requête α plusieurs fois.*

Les copies secondaires C_2 et C_3 détectent la défaillance de C_1 lorsqu'elles délivrent la nouvelle vue $v_1(C) = \{C_2, C_3\}$. En conséquence, elles choisissent C_2 comme nouvelle copie primaire.

Soit un objet O tel que O invoque C. Au cours du traitement de cette invocation, C invoque S. Lorsque O apprend la défaillance de la copie primaire C_1, il retransmet sa requête d'invocation en la dirigeant vers la nouvelle copie primaire C_2.

À la réception de la requête en provenance de O, la copie primaire C_2 transmet la requête α aux copies du serveur. Toutes les copies de S délivrent α. Comme la copie S_1 a déjà délivré cette requête, son état risque de devenir incohérent si le traitement de α n'est pas idempotent. On retrouve le problème du nombre variable de requêtes évoqué précédemment (cf. section 4.3.2). La propriété d'intégrité, déjà incluse dans la spécification, permet d'éviter le problème.

Il est clair que si le problème 2 se pose, le problème 1 est résolu. Les copies de S qui n'ont pas délivré $\alpha(C_1)$ délivrent $\alpha(C_2)$. Dès lors, toutes les copies de S délivrent α.

Défaillances de la copie primaire du client et d'une copie du serveur La figure 4.7-b reprend le scénario de la figure précédente en y ajoutant la défaillance de S_1, une copie du serveur. La défaillance de S_1 survient juste après que S_1 ait délivré α.

La défaillance de S_1 complique les problèmes 1 et 2:

Problème 1'. *La copie défaillante S_1 délivre la requête α avant de défaillir et les copies correctes de S ne délivrent jamais α.*

Si S_1 est la *seule* copie de S à délivrer α, aucune copie *correcte* (i.e. appartenant à $v_1(S)$) ne délivre $\alpha(C_1)$. Dans le cas où le traitement de α conduit S_1 à effectuer une action externe (par ex. invocation d'un objet tiers), l'état du système risque de devenir incohérent. En

effet, les copies correctes de S n'ont aucun moyen de prendre connaissance ni de cette action et ni de son effet.

Pour traiter ce cas, il faut ajouter la notion d'uniformité à la propriété d'agrément évoquée précédemment.

Problème 2'. *La copie défaillante S_1 délivre la requête α plusieurs fois.*

Pour que la propriété d'intégrité s'applique aussi à la copie défaillante S_1, il faut y ajouter la notion d'uniformité.

Duplication active du client et duplication passive du serveur

La figure 4.8 illustre deux scénarios de défaillances perturbant le traitement de la requête α. Les paramètres spécifiant la transmission de la requête ont les valeurs suivantes:

- $src(C) = copies(C)$;
- $dst(S) = prim(S)$;
- $ord(S) = FIFO$.

Chaque copie de C transmet un exemplaire de la requête α à S_1, la copie primaire de S, en utilisant une primitive garantissant l'ordre FIFO.

Défaillance d'une copie du client La figure 4.8-a illustre la défaillance de C_1 pendant la transmission de la requête α. Cette défaillance risque de compromettre la transmission de $\alpha(C_1)$, l'exemplaire de α que C_1 transmet à la copie primaire du serveur. Ce risque est lié à la sémantique de canal fiable comme dans le cas du problème 1.

Comme C_1 défaille pendant la transmission de $\alpha(C_1)$, il est possible que la copie primaire S_1 ne délivre jamais $\alpha(C_1)$. Mais comme S_1 est le seul destinataire de $\alpha(C_1)$, le risque d'incohérence évoqué à l'exemple précédent n'existe pas.

Comme le client est dupliqué activement, l'effet de cette défaillance est masqué par les copies correctes du client. Chaque copie correcte de C transmet un exemplaire de α à S_1, la copie primaire du serveur. Par conséquent, tant qu'il existe une copie correcte de C, la copie primaire S_1 a des chances de délivrer la requête α.

Défaillance d'une copie du client et défaillance de la copie primaire du serveur Sur la figure 4.8-b, la défaillance de la copie primaire S_1 s'ajoute à celle de C_1.

La défaillance de S_1 survient pendant le traitement de α, après que S_1 ait délivré $\alpha(C_2)$. La copie C_3 est en retard sur les autres: elle n'a pas encore commencé la transmission de $\alpha(C_3)$.

Par conséquent, S_1 ne délivre jamais $\alpha(C_3)$. De plus, les opérations effectuées au cours du traitement de $\alpha(C_2)$ sont perdues. Comme la copie primaire n'a pas synchronisé son état sur les copies secondaires S_2 et S_3, ces dernières ignorent l'existence de la requête α.

Les copies secondaires sont informées de la défaillance de la copie primaire, lorsqu'elles délivrent la vue $v_1(S)$. En conséquence, les copies secondaires élisent une nouvelle copie primaire parmi les membres de $v_1(S)$. Sur cet exemple, la nouvelle copie primaire est S_2.

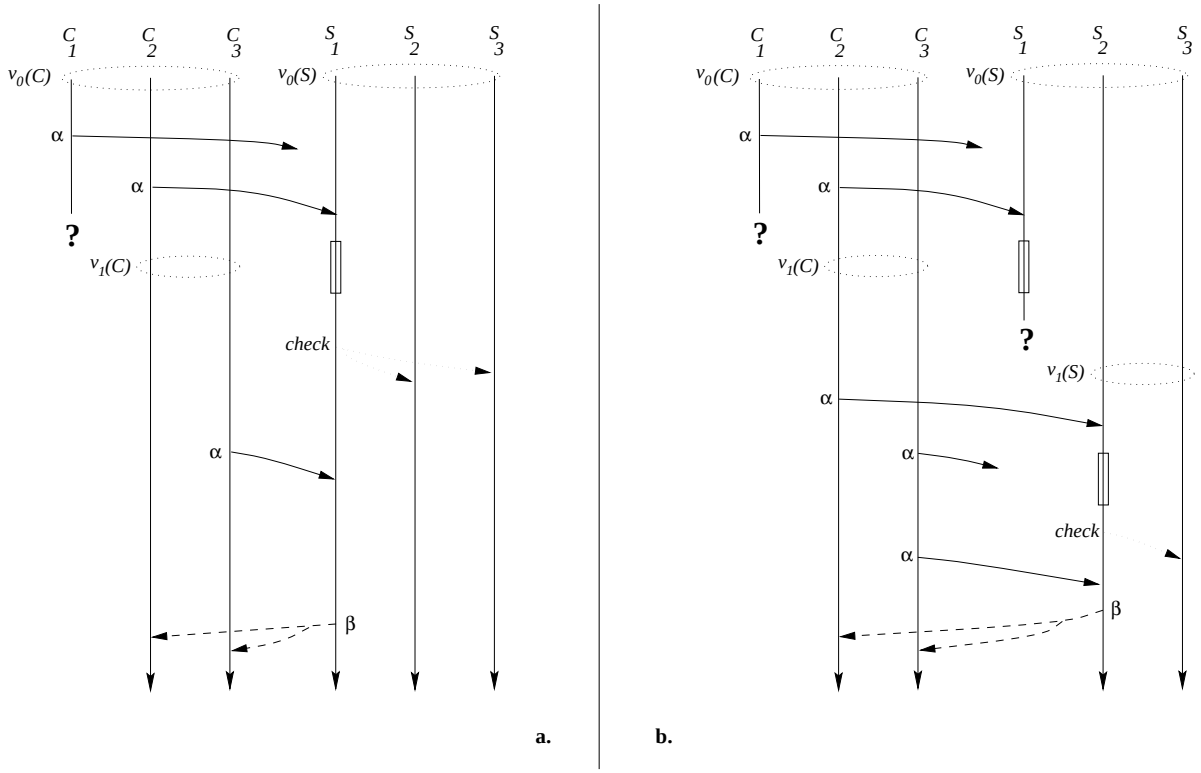

Figure 4.8: Duplication active du client et duplication passive du serveur: deux scénarios de défaillances

Pour que l'invocation puisse continuer malgré la défaillance de S_1, les copies correctes de C doivent retransmettre la requête α en la destinant cette fois à la nouvelle copie primaire S_2. Cette retransmission pose deux problèmes:

Problème 3. *Les copies correctes de C doivent détecter la défaillance de la copie primaire initiale S_1.*

Pour détecter la défaillance de la copie primaire initiale S_1, les copies correctes de C ne peuvent pas s'appuyer sur la délivrance de la vue $v_1(S)$. Cette vue est délivrée uniquement aux copies correctes de S: la portée des informations sur les défaillances exprimées par un groupe d'objets est limitée aux membres de ce groupe.

Pour détecter la défaillance de la copie primaire S_1, chaque copie correcte de C utilise le module du détecteur de défaillances, situé sur le nœud où elle réside.

Le détecteur de défaillances doit satisfaire la complétude forte: on garantit ainsi que la défaillance de S_1 finira par être suspectée par toutes les copies correctes de C. Par contre, la justesse forte n'a pas besoin d'être vérifiée: si une copie correcte C_i suspecte à tort une copie S_j, la copie C_i risque de transmettre plusieurs fois la même requête à S_j. La propriété d'intégrité protège S_j contre les effets indésirables de cette retransmission.

Problème 4. *Les copies correctes de C doivent déterminer l'identité de la nouvelle copie primaire S_2.*

Les copies correctes de C doivent déterminer l'identité de la nouvelle copie primaire pour pouvoir lui transmettre la requête α. Ceci revient en fait à réévaluer le paramètre $dst(S)$ qui identifie l'ensemble des copies de S à qui les copies de C doivent transmettre α.

Cette réévaluation correspond exactement au problème de la désignation du serveur (cf. section 4.3.2). Ce scénario met en évidence le caractère dynamique de la valeur de $dst(S)$: à chaque défaillance d'une copie de S, cette valeur doit être recalculée.

Il faut noter que ce problème se pose quelque soit la stratégie de duplication du serveur. Sur la figure 4.7-b, le problème est moins perceptible car le serveur est dupliqué activement. La copie C_2 peut utiliser la valeur initiale de $dst(S)$, soit $v_0(S)$, pour transmettre la requête aux copies appartenant à $v_1(S)$, car $v_0(S) \subset v_1(S)$. Par contre, lorsque le serveur est dupliqué passivement, il n'existe aucune relation entre les valeurs successives de $dst(S)$.

Commentaires

Tous les exemples sont basés sur la duplication active et sur la duplication passive. Les deux autres stratégies (i.e. duplication semi-active et duplication coordinateur-cohorte) s'inspirent à la fois de la duplication active et de la duplication passive. Elles ne posent pas de nouveaux problèmes, comme le montre la section 4.5.

Aucun scénario ne considère des défaillances survenant pendant la transmission de la réponse. La modélisation symétrique de l'invocation permet d'affirmer que les problèmes posés par les défaillances pendant la transmission de la réponse, sont les mêmes que pendant la transmission de la requête.

Le lecteur attentif aura remarqué que les scénarios sont basés sur des défaillances d'objets, par opposition à des défaillances de canaux. À la section 4.2, la défaillance d'un objet O a été définie comme étant équivalente à la défaillance par arrêt du processus hôte de l'objet O. Il est alors

légitime de se demander si les défaillances par omission affectant les canaux posent d'autres problèmes que ceux évoqués ici.

En réalité, l'abstraction de groupe prend en compte les défaillances par omission affectant les canaux. Lorsqu'une copie est exclue d'une vue, il est possible que cette copie ne soit pas défaillante mais que les canaux la reliant aux autres copies soient défaillants. Se retrouvant minoritaire (cf. section 4.4.1), cette copie est bloquée jusqu'au rétablissement de la communication.

Par conséquent, l'abstraction de groupe fait en sorte que les défaillances par omission de canaux reviennent à des défaillances par arrêt d'objets. La seule différence est le blocage de tout le groupe dans le cas ou il n'est pas possible de déterminer un sous-ensemble majoritaire.

4.4.3 Spécification étendue de l'invocation entre objets dupliqués

Cette section étend la spécification présentée à la section 4.3.4 pour prendre en compte les problèmes causés par les défaillances.

Transmission de la requête

Lorsque les défaillances sont prises en compte, les trois propriétés (validité, intégrité et ordre) ne suffisent plus à spécifier la transmission d'une requête α. Les paramètres $src(C)$, $dst(S)$ et $ord(S)$ sur lesquels se basent ces propriétés sont toujours valables mais l'interprétation des valeurs de $src(C)$ et de $dst(S)$ changent.

Le paramètre $dst(S)$ désigne l'ensemble formé par les copies de S qui doivent délivrer α. La composition de $dst(S)$ dépend de la vue du groupe des copies de S, et de $str(S)$, la stratégie de duplication de S.

Entre le moment où une copie C_i détermine la valeur de $dst(S)$ pour transmettre $\alpha(C_i)$ et le moment où S_j appartenant à $dst(S)$ délivre $\alpha(C_i)$, certaines copies appartenant à $dst(S)$ ont pu défaillir. Par conséquent, l'ensemble $dst(S)$ est composé par des copies correctes et par des copies défaillantes.

Ces considérations conduisent à spécifier la transmission de la requête à l'aide des propriétés suivantes:

- **Validité:** *Si un élément* correct *de* $dst(S)$ *délivre une requête* α*, alors un élément* correct *de* $src(C)$ *a émis* α*.*
- **Agrément uniforme:** *Si un élément* (correct ou défaillant) *de* $dst(S)$ *a délivré* α*, alors tout élément* correct *de* $dst(S)$ *délivre* α*.*
- **Intégrité uniforme:** *Tout élément* (correct ou défaillant) *de* $dst(S)$ *délivre* α *au plus une fois.*
- **Ordre:** *Tout élément* correct *de* $dst(S)$ *délivre* α *dans l'ordre spécifié par* $ord(S)$*.*

Les propriétés de validité et d'ordre sont similaires à celles de la section 4.3.4, mais elles ne s'appliquent qu'aux copies correctes de C et de S.

La propriété d'agrément uniforme combine deux notions: l'agrément et l'uniformité. La notion d'agrément exprime l'atomicité de la délivrance de la requête α aux éléments de $dst(S)$. En

d'autres termes, cette notion exprime que α est délivrée *par tous* les éléments corrects de $dst(S)$ ou bien *par aucun* d'entre eux. Cette propriété permet d'éviter le problème 1 présenté à la section 4.4.2. En l'absence de défaillances, la délivrance de α ne peut être compromise et cette notion n'est donc pas nécessaire.

L'uniformité étend la notion d'agrément aux copies défaillantes: si une copie S_j *correcte ou défaillante* appartenant à $dst(S)$ délivre α, alors toutes les copies correctes de $dst(S)$ délivrent α. L'uniformité permet de prendre en compte le problème 1' présenté à la section 4.4.2.

La propriété d'intégrité est également uniforme: elle concerne *toutes* les copies de S, qu'elles soient correctes ou défaillantes. L'intégrité uniforme garantit qu'*aucune* copie de S ne délivre α plusieurs fois. On évite ainsi le problème 2'.

Traitement de la requête

Contrairement aux autres phases de l'invocation, le traitement de la requête ne concerne que le serveur S. Tant que les défaillances ne sont pas considérées, le traitement de la requête ne nécessite aucune communication. La seule propriété que doivent vérifier les éléments de $cpu(S)$ est le déterminisme: chaque copie de S qui traite une requête α doit produire la même réponse β.

Lorsque les défaillances sont considérées, la propriété de déterminisme ne s'applique qu'aux copies correctes de $cpu(S)$:

- **Déterminisme:** *Tout élément correct de $cpu(S)$ produit la même réponse β comme résultat du traitement de α.*

Cette spécification doit prendre en compte les problèmes causés par les défaillances survenant pendant le traitement de la requête. Dans le cas de la duplication active, la défaillance d'une copie pendant le traitement de la requête est masquée par les autres copies, puisque toutes les copies effectuent le traitement. Par conséquent, la propriété de déterminisme suffit.

Par contre, dans le cas d'une stratégie de duplication à copie primaire (par ex. duplication passive, semi-active et coordinateur-cohorte), cette spécification est incomplète car elle ne modélise pas deux aspects essentiels:

- la synchronisation de l'état des copies;
- l'élection d'une copie primaire.

Synchronisation de l'état des copies La copie primaire de S, $prim(S)$, contrôle les interactions entre les copies secondaires et le reste du système. Ce rôle privilégié rend S particulièrement vulnérable à la défaillance de $prim(S)$. Afin d'assurer qu'une copie secondaire puisse être promue copie primaire, il est essentiel de synchroniser régulièrement l'état des copies secondaires sur celui de $prim(S)$.

Selon la stratégie de duplication considérée, la copie primaire de S effectue la synchronisation *avant* (pré-synchronisation) de traiter une requête α ou bien *après* (post-synchronisation). Ces deux approches diffèrent essentiellement par l'information de synchronisation que la copie primaire transmet aux copies secondaires.

Dans le cas de la *pré-synchronisation*, la copie primaire $prim(S)$ n'a pas encore traité la requête α et elle transmet aux copies secondaires toute l'information nécessaire au traitement de α.

Cette information est constituée par la requête α et par une indication portant sur l'instant logique auquel le traitement de α doit être entrepris. En cas de défaillance de $prim(S)$, la copie secondaire promue nouvelle copie primaire, peut alors effectuer le traitement de α et répondre au client.

La figure 4.9-a présente un serveur S, dupliqué semi-activement, qui utilise la pré-synchronisation. À la réception de la requête α, la copie primaire S_1 transmet un message $notify$ aux copies secondaires pour leur préciser l'instant logique auquel elles devront traiter la requête α.

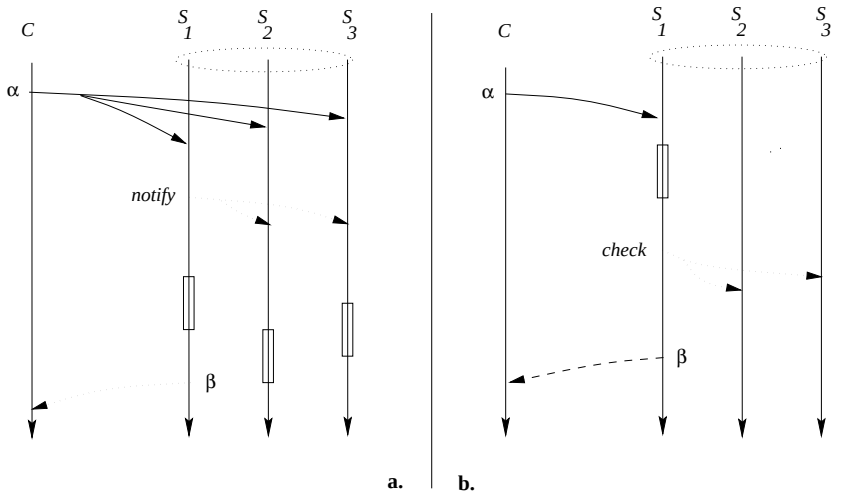

Figure 4.9: Pré-synchronisation (a.) vs post-synchronisation (b.)

Dans le cas de la *post-synchronisation*, la copie primaire $prim(S)$ a déjà traité la requête α et elle transmet aux copies secondaires les résultats de ce traitement. Ces résultats sont constitués par la réponse β que $prim(S)$ va envoyer ultérieurement au client C, et par l'état courant de $prim(S)$. Chaque copie secondaire S_i met à jour son état avec celui de $prim(S)$ et stocke β. Si S_i devient la nouvelle copie primaire suite à la défaillance de $prim(S)$, d'une part son état est à jour, et d'autre part, elle est en mesure de transmettre la réponse β au client C. Ainsi, lors de la terminaison de la phase de traitement de la requête, toutes les copies de S ont l'air d'avoir traité S, alors que seule $prim(S)$ a effectué le traitement.

La figure 4.9-b présente un serveur S, dupliqué passivement, qui utilise la post-synchronisation. Après avoir traité la requête α, la copie primaire S_1 transmet un message *check* contenant la réponse β et son nouvel état.

Indépendemment de l'approche adoptée, la synchronisation de l'état des copies est spécifiée par les propriétés définies ci-dessous. Soient *sync* l'information de synchronisation et $sec(S)$ l'ensemble des copies secondaires de S:

- **Validité:** *Si un élément correct de $sec(S)$ délivre une information de synchronisation sync, alors la copie $prim(S)$ correcte a émis sync.*

- **Agrément uniforme:** *Si un élément* (correct ou défaillant) *de $sec(S)$ délivre sync, alors tout élément* correct *$sec(S)$ délivre sync.*

- **Intégrité uniforme:** *Tout élément* correct ou défaillant *de* $sec(S)$ *délivre sync au plus une fois.*
- **Ordre FIFO:** *Si* $prim(S)$ *envoie sync puis* $sync'$*, alors tout élément de* $sec(S)$ *ne délivre* $sync'$ *qu'après avoir délivré sync.*

Cette spécification s'apparente à celle de la transmission de la requête (cf. section 4.4.3). Dans ce cas, $src(S) = prim(S)$ et $dst(S) = sec(S)$. La différence réside dans le fait que $prim(S)$ et $sec(S)$ sont tous deux des sous-ensembles de $copies(S)$, l'ensemble des copies de S. À cette différence près, on peut voir la synchronisation de l'état des copies comme un cas particulier de la transmission de la requête: $src(S) = prim(S)$ et $ord(S) = FIFO$. Cette spécification est en outre rigoureusement identique à celle du multicast fiable uniforme, garantissant l'ordre FIFO, donnée dans [Hadzilacos 93].

Élection d'une copie primaire Lorsque la copie primaire défaille, une copie secondaire devient la nouvelle copie primaire. Cette promotion est le résultat d'une *élection* à laquelle participent toutes les copies secondaires. Une élection est spécifiée ainsi [Sabel 95]:

- **Sûreté:** *Il existe au plus une copie primaire.*
- **Vivacité:** *S'il n'existe pas de copie primaire, une copie secondaire est finalement élue copie primaire.*

Pour obtenir une spécification de l'élection basée sur les propriétés utilisées jusqu'ici (intégrité, validité, etc.), il suffit de faire le rapprochement entre le problème de l'élection et celui du consensus.

Le problème du consensus est défini comme suit: un ensemble d'objets se concertent dans le but de *décider*, de manière unanime et irrévocable, la valeur d'une information. La valeur décidée est choisie parmi celles que chaque objet *propose* à ses pairs. Les auteurs de [Chandra 94] spécifie le consensus ainsi:

- **Terminaison:** *Tout objet correct finit par décider une valeur.*
- **Intégrité uniforme:** *Tout objet décide au plus une fois.*
- **Agrément:** *Si un objet correct décide une valeur* v*, alors tout objet correct décide* v*.*
- **Validité uniforme:** *Si un objet décide* v*, alors il existe un objet qui a proposé* v*.*

Si la propriété d'agrément est uniforme, on parle alors de consensus uniforme.

Dans le contexte de l'élection, la valeur décidée est l'identité de la copie primaire. Chaque copie *vote* (propose) pour une copie particulière. Toutes les copies correctes *élisent* (décident) la même copie primaire. L'élection est spécifiée ainsi:

- **Sûreté:** *Il existe au plus une copie primaire.*
- **Terminaison:** *Chaque élément correct de* $sec(S)$ *élit finalement une copie primaire.*
- **Intégrité uniforme:** *Chaque élément de* $sec(S)$ *élit au plus une fois.*

- **Agrément uniforme:** *Si un élément de $sec(S)$ élit S_i, alors tout élément correct de $sec(S)$ élit S_i.*
- **Validité uniforme:** *Si un élément de $sec(S)$ élit S_i, alors il existe un élément de $sec(S)$ qui a voté pour Si.*

Exception faite de la propriété de sûreté, cette spécification est identique à celle du consensus uniforme. La propriété de sûreté exprime que, s'il existe une copie primaire pour S, cette copie primaire doit être *unique*.

Transmission de la réponse

La spécification de la transmission de la réponse est identique à celle de la transmission de la requête:

- **Validité:** *Si un élément* correct *de $dst(C)$ délivre une réponse β, alors un élément* correct *de $src(S)$ a émis β.*
- **Agrément uniforme:** *Si un élément* (correct ou défaillant) *de $dst(C)$ délivre β, alors tout élément* correct *de $dst(C)$ délivre β.*
- **Intégrité uniforme:** *Tout élément* correct ou défaillant *de $dst(C)$ délivre β au plus une fois.*
- **Ordre:** *Tout élément* correct *de $dst(C)$ délivre β dans l'ordre spécifié par $ord(C)$.*

4.5 Application aux stratégies de duplication

Cette section applique la spécification de l'invocation aux quatres stratégies de duplication considérées dans cette thèse: la duplication active (cf. section 4.5.1), la duplication passive (cf. section 4.5.2), la duplication semi-active (cf. section 4.5.3) et la duplication coordinateur-cohorte (cf. section 4.5.4). La section 4.5.5 compare les quatre stratégies.

4.5.1 Duplication active

Principe

La duplication active est définie ainsi:

- pour un objet serveur:
  - délivrance des requêtes: *toutes les copies délivrent la même séquence[8] de requêtes;*
  - traitement des requêtes: *toutes les copies traitent les requêtes de manière déterministe[9];*

[8] Une séquence est un ensemble ordonné totalement. Les copies délivrent les *mêmes* requêtes dans le *même* ordre.

[9] Un traitement déterministe produit toujours le même résultat à partir des mêmes données. Par conséquent, pour une requête donnée, toutes les copies produisent la même réponse.

  - émission des réponses: *toutes les copies émettent la même séquence de réponses;*

- pour un objet client:
  - émission des requêtes: *toutes les copies émettent la même séquence de requêtes;*
  - délivrance des réponses: *toutes les copies délivrent la même séquence de réponses.*

Cette définition complète la définition présentée à la section 2.5.2. La première définition ne décrivait le principe de la duplication active que pour un objet serveur.

Spécification

Soit O un objet dupliqué activement. La vue courante du groupe des copies de O est notée $v_k(O)$. Le principe de la duplication active se traduit, en termes des paramètres définis à la section 4.3.4, de la façon suivante:

- pour un objet serveur:
  - délivrance des requêtes: $dst(O) = v_k(O)$ et $ord(O) = TOTAL$;
  - traitement des requêtes: $cpu(O) = v_k(O)$;
  - émission des réponses: $src(O) = v_k(O)$;
- pour un objet client:
  - émission des requêtes: $src(O) = v_k(O)$;
  - délivrance des réponses: $dst(O) = v_k(O)$ et $ord(O) = TOTAL$.

Participation d'un objet dupliqué activement à une invocation

La figure 4.10 illustre la participation de l'objet O à une invocation, en tant que serveur (a.) et en tant que client (b.). L'autre objet participant à l'invocation n'est pas représenté: la stratégie de duplication de cet objet peut être quelconque.

Sur cet exemple, $v_k(O) = \{O_1, O_2, O_3\}$. Les numéros en gras indiquent les différentes phases de l'invocation:

1. transmission de la requête;
2. traitement de la requête;
3. transmission de la réponse.

Les phases sont séparées les unes des autres par un trait horizontal en pointillé.

Lorsque O est serveur (cf. figure 4.10-a), il participe aux trois phases de l'invocation. Chaque copie O_i délivre la requête α (phase 1), produit la réponse β en traitant α (phase 2), puis transmet la réponse β vers le client (phase 3).

Lorsque O est client (cf. figure 4.10-b), il ne participe qu'aux phases 1 et 3. Chaque copie O_i transmet la requête α (phase 1), et délivre la réponse β transmise par le serveur (phase 3).

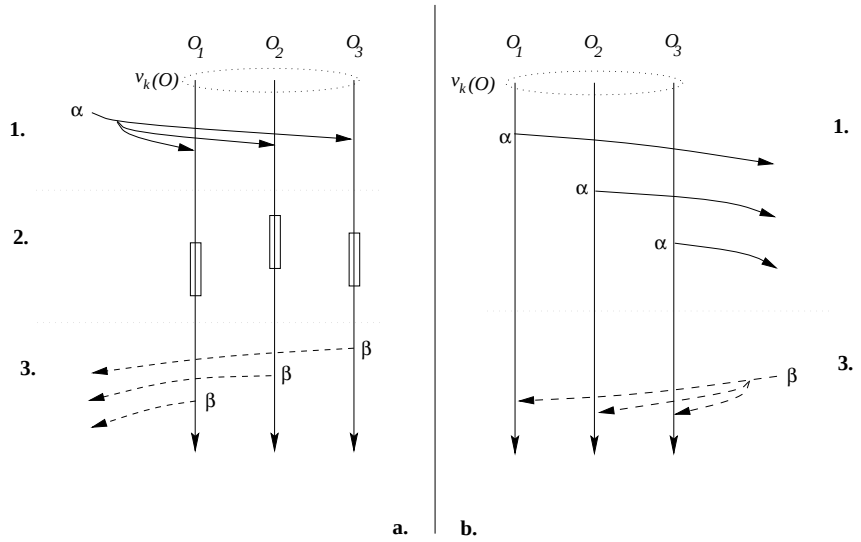

Figure 4.10: a. serveur dupliqué activement, b. client dupliqué activement

Tolérance aux fautes d'un objet dupliqué activement

Comme toutes les copies de O ont un comportement strictement identique, la défaillance d'une copie O_i est généralement masquée par les copies correctes. Quelque soit la phase pendant laquelle elle survient, la défaillance de O_i ne perturbe pas le déroulement de l'invocation.

Il est clair que, si la défaillance de O_i empêche la construction d'une nouvelle vue (i.e. plus de majorité), toutes les copies correctes sont bloquées. Cette situation est inhérente à l'abstraction de groupe: elle peut se produire quelque soit la stratégie de duplication de O.

Défaillance d'une copie d'un serveur dupliqué activement Chaque cas de défaillance correspond à une phase précise de l'invocation:

a.1. Si O_i défaille pendant la transmission de la requête α, la propriété d'agrément uniforme garantit que les copies correctes de O restent cohérentes.

a.2. Si O_i défaille pendant le traitement de α, les copies correctes de O ne sont pas affectées. Si au cours de ce traitement, O_i invoque d'autres objets, on retrouve le cas de la défaillance d'une copie d'un client dupliqué activement (cf. cas b.1 et b.3)

a.3. Si O_i défaille pendant la transmission de la réponse β, le message $\beta(O_i)$ risque de ne pas être délivré au client. Comme celui-ci délivre les réponses des copies correctes de O, cette défaillance est sans conséquence.

Défaillance d'une copie d'un client dupliqué activement Chaque cas de défaillance correspond à une phase précise de l'invocation:

b.1 Si O_i défaille pendant la transmission de la requête α, la délivrance de $\alpha(O_i)$ risque d'être compromise. Comme les copies correctes de O transmettent également α, cette défaillance

est sans conséquence.

b.3. Si O_i défaille pendant la transmission de la réponse β, la propriété d'agrément uniforme garantit que les copies correctes de O restent cohérentes. Ce cas est symétrique au cas a.1.

Particularités

Si O est serveur (resp. client), comme $src(O) = v_k(O)$, le problème du nombre variable de réponses (resp. requêtes) se pose systématiquement, même en l'absence de défaillances.

Si O est serveur (resp. client), comme $dst(O) = v_k(O)$, la propriété d'agrément uniforme est particulièrement importante pour la transmission de la requête (resp. de la réponse).

4.5.2 Duplication passive

Principe

La duplication passive est définie ainsi:

- pour un objet serveur:
  - délivrance des requêtes: *la copie primaire est la seule à délivrer les requêtes;*
  - traitement des requêtes: *la copie primaire est la seule à traiter les requêtes;*
  - émission des réponses: *la copie primaire est la seule à émettre les réponses;*
- pour un objet client:
  - émission des requêtes: *la copie primaire est la seule à émettre les requêtes;*
  - délivrance des réponses: *la copie primaire est la seule à délivrer les réponses;*

Cette définition complète la définition présentée à la section 2.5.3. La première définition ne décrivait le principe de la duplication passive que pour un objet serveur.

Spécification

Soit un objet O dupliqué passivement. La copie primaire de O est notée $prim(O)$. Le principe de la duplication passive se traduit, en termes des paramètres définis à la section 4.3.4, de la façon suivante:

- pour un objet serveur:
  - délivrance des requêtes: $dst(O) = prim(O)$ et $ord(O) = FIFO$;
  - traitement des requêtes: $cpu(O) = prim(O)$;
  - émission des réponses: $src(O) = prim(O)$;
- pour un objet client:
  - émission des requêtes: $src(O) = prim(O)$;
  - délivrance des réponses: $dst(O) = prim(O)$ et $ord(O) = FIFO$;

Participation d'un objet dupliqué passivement à une invocation

La figure 4.11 illustre la participation de l'objet O à une invocation, en tant que serveur (a.) et en tant que client (b.).

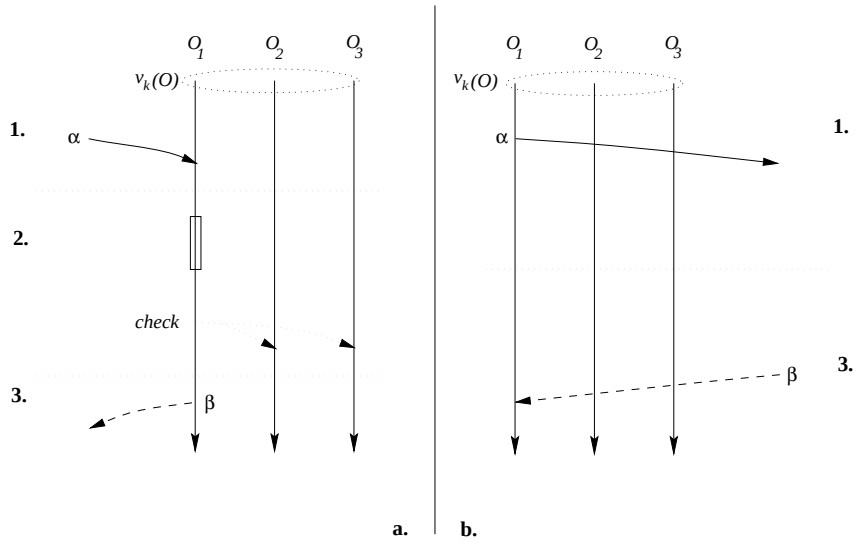

Figure 4.11: a. serveur dupliqué passivement, b. client dupliqué passivement

Lorsque O est serveur (cf. figure 4.11-a), il participe aux trois phases de l'invocation. Seule la copie primaire O_1 délivre la requête α (phase 1), produit la réponse β en traitant α (phase 2), puis transmet la réponse β vers le client (phase 3).

Lorsque O est client (cf. figure 4.11-b), il ne participe qu'aux phases 1 et 3. Seule la copie primaire O_1 transmet la requête α (phase 1), et délivre la réponse β transmise par le serveur (phase 3).

Tolérance aux fautes d'un objet dupliqué passivement

Comme la copie primaire joue un rôle primordial, sa défaillance pose problème. Lorsque la copie primaire défaille:

- les copies secondaires doivent élire une nouvelle copie primaire;
- l'interlocuteur de O doit déterminer l'identité de la nouvelle copie primaire (problème de désignation);
- si O est serveur (resp. client), le client (resp. serveur) doit impérativement retransmettre sa requête (resp. réponse) pour avoir une chance de délivrer une réponse (resp. requête).

Par contre, la défaillance d'une copie secondaire ne pose pas de problème particulier. La duplication passive utilise la post-synchronisation pour synchroniser les copies secondaires sur l'état de la copie primaire.

Défaillance de la copie primaire d'un serveur dupliqué passivement Chaque cas de défaillance correspond à une phase précise de l'invocation:

a.1. Si $prim(O)$ défaille pendant la transmission de la requête α, la requête est perdue. Les copies secondaires n'ont aucun moyen de connaître l'existence de cette requête.

a.2. Si $prim(O)$ défaille pendant le traitement de α, il faut distinguer deux cas:

a.2.1 Si $prim(O)$ défaille *avant* de délivrer le message de synchronisation *check*, la requête α et le traitement effectué par $prim(O)$ sont perdus. Ce cas de défaillance est sensiblement plus grave que le cas a.1 car $prim(O)$ a pu invoquer un objet tiers avant de défaillir. Il faut toutefois noter que les copies secondaires n'ont aucun moyen de faire la différence entre ces deux cas.

Lorsque α est retransmise par le client, la nouvelle copie primaire refait systématiquement l'invocation de l'objet tiers. L'état de ce dernier serait corrompu si la propriété d'intégrité ne le protégeait pas des requêtes redondantes. Il s'agit ici d'une variante du problème du nombre variable de requêtes.

a.2.2 Si $prim(O)$ défaille *après* avoir délivré le message de synchronisation *check*, les copies secondaires ont également délivré *check*. Elles ont donc pris connaissance de la requête α, de la réponse β et du changement d'état occasionné par le traitement de α.

Lorsque α est retransmise par le client, la nouvelle copie primaire transmet directement la réponse β sans effectuer le traitement.

a.3. Si $prim(O)$ défaille pendant la transmission de la réponse β, on retrouve le cas a.2.2. Les copies secondaires n'ont aucun moyen de distinguer ces deux cas. La nouvelle copie primaire envoie systématiquement la réponse β au client. La propriété d'intégrité empêche le client de délivrer la réponse β plusieurs fois.

Défaillance de la copie primaire d'un client dupliqué passivement Chaque cas de défaillance correspond à une phase précise de l'invocation:

b.1. Si $prim(O)$ défaille pendant la transmission de la requête α, cette requête est perdue. Les copies secondaires n'ont aucun moyen de connaître l'existence de cette requête.

Soit α' une requête transmise à O par un objet O', telle que, au cours du traitement de α', l'objet O transmet la requête α. L'objet O joue le rôle de serveur traitant la requête α'. Ce cas revient donc exactement au cas a.2.1.

b.3. Si $prim(O)$ défaille pendant la transmission de la réponse β, cette réponse est perdue. Les copies secondaires n'ont aucun moyen de connaître l'existence de cette réponse.

Le serveur n'a pas d'autre choix que de retransmettre la réponse β. Ce cas est symétrique au cas a.1.

Particularités

Si le problème de l'élection d'une copie primaire est commun à toutes les stratégies de duplication à copie primaire, le problème de la retransmission est propre à la duplication passive. Ce problème est du principalement au fait que les messages (i.e. requêtes et réponses) ne sont délivrés que par la copie primaire.

4.5.3 Duplication semi-active

Principe

La duplication semi-active est définie ainsi:

- pour un objet serveur:
  - délivrance des requêtes: *toutes les copies délivrent le même ensemble*[10] *de requêtes;*
  - traitement des requêtes: *toutes les copies traitent toutes les requêtes. La copie primaire traite une requête dès qu'elle la délivre. Par contre, une copie secondaire doit attendre une notification de la copie primaire pour pouvoir traiter une requête;*
  - émission des réponses: *la copie primaire est la seule à émettre les réponses.*
- pour un objet client:
  - émission des requêtes: *toutes les copies émettent toutes les requêtes;*
  - délivrance des réponses: *toutes les copies délivrent le même ensemble de réponses.*

Cette définition complète la définition présentée à la section 2.5.4. La première définition ne décrivait le principe de la duplication semi-active que pour un objet serveur.

Spécification

Soit un objet O dupliqué semi-activement. Le principe de la duplication semi-active se traduit, en termes des paramètres définis à la section 4.3.4, de la façon suivante:

- pour un objet serveur:
  - délivrance des requêtes: $dst(O) = v_k(O)$ et $ord(O) = FIFO$;
  - traitement des requêtes: $cpu(O) = v_k(O)$;
  - émission des réponses: $src(O) = prim(O)$;
- pour un objet client:
  - émission des requêtes: $src(O) = v_k(O)$;
  - délivrance des réponses: $dst(O) = v_k(O)$ et $ord(O) = FIFO$;

Participation d'un objet dupliqué semi-activement à une invocation

La figure 4.12 illustre la participation de l'objet O à une invocation, en tant que serveur (a.) et en tant que client (b.).

Lorsque O est serveur (cf. figure 4.12-a), il participe aux trois phases de l'invocation. Chaque copie O_i délivre la requête α (phase 1) et produit la réponse β en traitant α (phase 2). Seule la copie primaire transmet la réponse β vers le client (phase 3).

Lorsque O est client (cf. figure 4.12-b), il ne participe qu'aux phases 1 et 3. Chaque copie O_i transmet la requête α (phase 1), et délivre la réponse β transmise par le serveur (phase 3).

[10] Les requêtes ne sont pas totalement ordonnées.

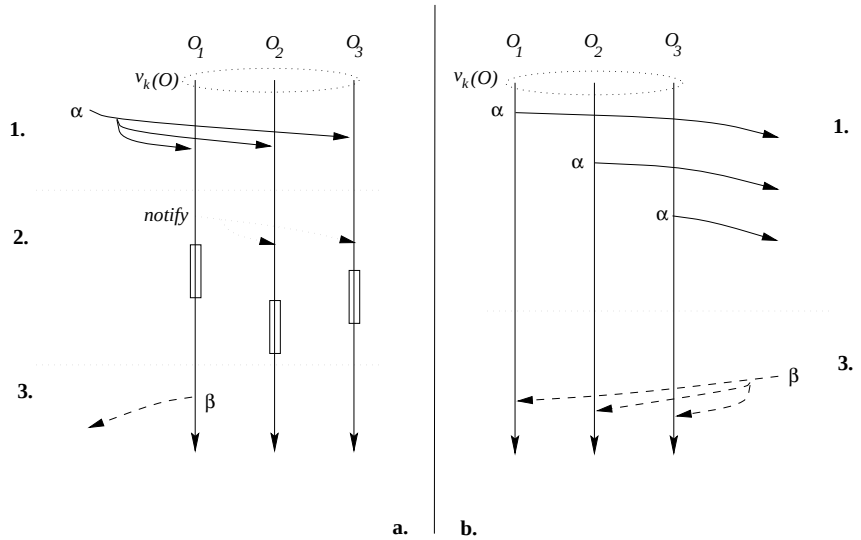

Figure 4.12: a. serveur dupliqué semi-activement, b. client dupliqué semi-activement

Tolérance aux fautes d'un objet dupliqué semi-activement

La duplication semi-active est une stratégie à copie primaire, s'inspirant à la fois de la duplication active et de la duplication passive. La défaillance de la copie primaire pose toutefois moins de problèmes que dans le cas de la duplication passive: la retransmission de la requête (resp. réponse) par le client (resp. serveur) n'est pas nécessaire. La duplication semi-active utilise la pré-synchronisation pour synchroniser les copies secondaires sur l'état de la copie primaire.

Défaillance de la copie primaire d'un serveur dupliqué semi-activement Chaque cas de défaillance correspond à une phase précise de l'invocation:

a.1. Si $prim(O)$ défaille pendant la transmission de la requête α, la propriété d'agrément uniforme garantit que les copies correctes de O restent cohérentes. Cette remarque est aussi valable pour la défaillance d'une copie secondaire.

a.2. Si $prim(O)$ défaille pendant le traitement de α, il faut distinguer deux cas:

a.2.1 Si $prim(O)$ défaille avant de délivrer le message de pré-synchronisation $notify$, les copies secondaires ne peuvent pas traiter la requête α avant d'avoir élu une nouvelle copie primaire et d'avoir délivré le message de pré-synchronisation correspondant à α.

Le message de pré-synchronisation indique l'instant logique à partir duquel les copies secondaires peuvent traiter la requête α. La propriété de vue-synchronisme (cf. section 2.3.3) doit être vérifiée pour que ce cas de défaillances soit traité correctement.

L'exemple de la figure 4.13 le montre. Le serveur O est invoqué simultanément par deux clients (non représentés sur la figure). Ces deux invocations correspondent aux requêtes α et α'.

Les copies O_1 et O_3 délivrent α avant α' mais O_2 délivre α après α'. La transmission de la requête n'assurant pas l'ordre total, ce scénario est possible. Juste avant de défaillir,

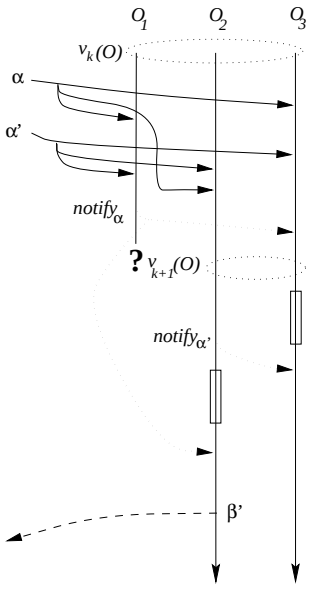

Figure 4.13: Importance de la propriété de vue-synchronisme pour un serveur semi-activement dupliqué

la copie primaire O_1 transmet une notification $notify_\alpha$ aux copies secondaires O_2 et O_3 leur indiquant qu'elle peuvent traiter α.

La copie O_2 délivre la nouvelle vue $v_{k+1}(O)$ avant de délivrer la notification $notify_\alpha$. Par contre, O_3 a déjà délivré $notify_\alpha$ et commencé le traitement de α, lorsqu'elle délivre $v_{k+1}(O)$.

La copie O_2 est élue nouvelle copie primaire, elle effectue alors la pré-synchronisation pour les requêtes en attente de traitement. Comme O_2 a délivré α' avant α, elle transmet $notify'_\alpha$ à la copie secondaire O_3 et effectue le traitement de α' sans avoir traité α. Lorsque O_2 délivre $notify_\alpha$, il est trop tard: O_2 et O_3 ont traité les requêtes α et α' dans un ordre différent! Si ces requêtes ne sont pas commutatives, les états des deux copies deviennent incohérents.

a.2.2 Si $prim(O)$ défaille après avoir délivré le message de synchronisation $notify$, les copies correctes de O ne sont pas affectées. Si au cours de ce traitement, O_i invoque d'autres objets, on retrouve le cas de la défaillance d'une copie d'un client dupliqué semi-activement.

a.3. Si $prim(O)$ défaille pendant la transmission de la réponse β, la réponse risque de ne pas être délivrée au client.

Les copies secondaires n'ont aucun moyen de distinguer ce cas du cas a.2.2. Par conséquent, la nouvelle copie primaire de O va systématiquement transmettre la réponse quand elle détecte la défaillance de la copie primaire initiale. La propriété d'intégrité garantit que le client ne délivre pas la réponse plusieurs fois.

Défaillance de la copie primaire d'un client dupliqué semi-activement On retrouve exactement les mêmes problèmes que pour un client dupliqué activement.

Particularités

Outre les problèmes hérités de la duplication active (par ex. nécessité de l'agrément uniforme) ou de la duplication passive (par ex. élection d'une copie primaire), la duplication semi-active se distingue par son besoin de la sémantique vue-synchrone.

4.5.4 Duplication coordinateur-cohorte

Principe

La duplication coordinateur-cohorte est définie ainsi:

- pour un objet serveur:
  - délivrance des requêtes: *toutes les copies délivrent le même ensemble de requêtes;*
  - traitement des requêtes: *la copie primaire est la seule à traiter les requêtes;*
  - émission des réponses: *la copie primaire est la seule à émettre les réponses;*
- pour un objet client:
  - émission des requêtes: *la copie primaire est la seule à émettre les requêtes;*
  - délivrance des réponses: *toutes les copies délivrent le même ensemble de réponses.*

Cette définition complète la définition présentée à la section 2.5.5. La première définition ne décrivait le principe de la duplication coordinateur-cohorte que pour un objet serveur.

Spécification

Soit un objet O dupliqué "coordinateur-cohorte". Le principe de la duplication coordinateur-cohorte se traduit, en termes des paramètres définis à la section 4.3.4, de la façon suivante:

- pour un objet serveur:
  - délivrance des requêtes: $dst(O) = v_k(O)$ et $ord(O) = FIFO$;
  - traitement des requêtes: $cpu(O) = prim(O)$;
  - émission des réponses: $src(O) = prim(O)$;
- pour un objet client:
  - émission des requêtes: $src(O) = prim(O)$;
  - délivrance des réponses: $dst(O) = v_k(O)$ et $ord(O) = FIFO$;

Participation d'un objet dupliqué "coordinateur-cohorte" à une invocation

La figure 4.14 illustre la participation de l'objet O à une invocation, en tant que serveur (a.) et en tant que client (b.).

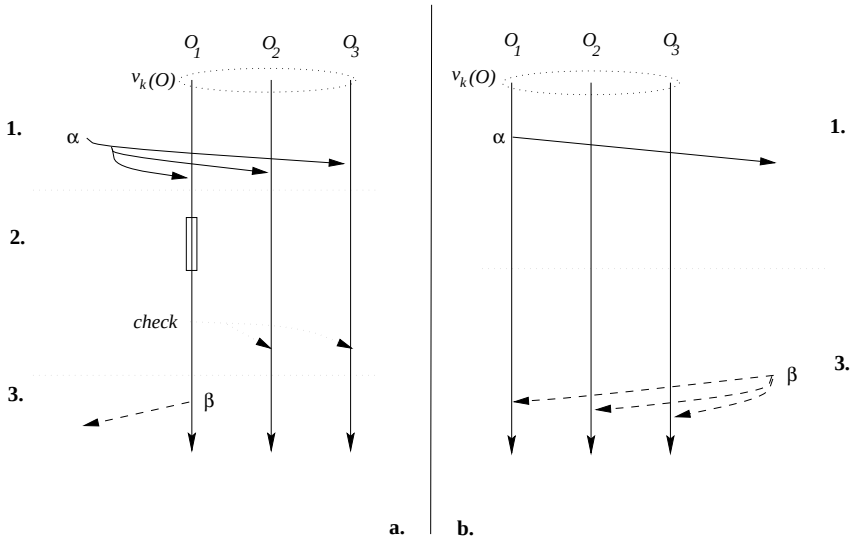

Figure 4.14: a. serveur dupliqué "coordinateur-cohorte", b. client dupliqué "coordinateur-cohorte"

Lorsque O est serveur (cf. figure 4.14-a), il participe aux trois phases de l'invocation. Chaque copie O_i délivre la requête α (phase 1). Seule la copie primaire O_1 produit la réponse β en traitant α (phase 2), puis transmet la réponse β vers le client (phase 3).

Lorsque O est client (cf. figure 4.14-b), il ne participe qu'aux phases 1 et 3. Seule la copie primaire O_1 transmet la requête α (phase 1). Chaque copie O_i délivre la réponse β transmise par le serveur (phase 3).

Tolérance aux fautes d'un objet dupliqué "coordinateur-cohorte"

Comme la duplication semi-active, la duplication coordinateur-cohorte est une stratégie hybride, à mi-chemin entre la duplication active et la duplication passive. Comme la duplication passive, la duplication coordinateur-cohorte utilise la post-synchronisation pour synchroniser les copies secondaires sur l'état de la copie primaire.

Défaillance de la copie primaire d'un serveur dupliqué "coordinateur-cohorte" Chaque cas de défaillance correspond à une phase précise de l'invocation:

a.1. Si $prim(O)$ défaille pendant la transmission de la requête α, la propriété d'agrément uniforme garantit que les copies correctes de O restent cohérentes. Cette remarque est aussi valable pour la défaillance d'une copie secondaire.

a.2. Si $prim(O)$ défaille pendant le traitement de α, il faut distinguer deux cas:

a.2.1 Si $prim(O)$ défaille avant de délivrer le message de synchronisation *check*, le traitement effectué par $prim(O)$ est perdu mais pas la requête α.

Si le traitement de α comporte une invocation d'un objet tiers, la nouvelle copie primaire refait systématiquement l'invocation de l'objet tiers. L'état de ce dernier

risquerait de devenir incohérent si la propriété d'uniformité ne le protégeait pas des requêtes redondantes. Il s'agit ici d'une variante du problème du nombre variable de requêtes.

a.2.2 Si $prim(O)$ défaille après avoir délivré le message de synchronisation *check*, les copies secondaires ont également délivré *check*. Elles ont donc pris connaissance de la requête α, de la réponse β et du changement d'état occasionné par le traitement de α.

La nouvelle copie primaire transmet systématiquement la réponse β au client.

a.3. Si $prim(O)$ défaille pendant la transmission de la réponse β, la réponse risque de ne pas être délivrée au client.

Les copies secondaires n'ont aucun moyen de distinguer ce cas du cas a.2.2. Par conséquent, la nouvelle copie primaire de O va systématiquement transmettre la réponse quand elle détecte la défaillance de la copie primaire initiale. La propriété d'intégrité garantit que le client ne délivre pas la réponse plusieurs fois.

Défaillance de la copie primaire d'un client dupliqué "coordinateur-cohorte" On retrouve exactement les mêmes problèmes que pour un client dupliqué passivement.

4.5.5 Synthèse

Le tableau 4.4 résume, pour les quatres stratégies de duplication considérées, les valeurs des paramètres de l'invocation entre objets dupliqués. Si O est un serveur dupliqué, $dst(O)$ et

stratégies de duplication	**paramètres de l'invocation entre objets dupliqués**						
	serveur dupliqué O				client dupliqué O		
	phase 1		phase 2	phase 3	phase 1	phase 3	
	$dst(O)$	$ord(O)$	$cpu(O)$	$src(O)$	$src(O)$	$dst(O)$	$ord(O)$
active	$v_k(O)$	$TOTAL$	$v_k(O)$	$v_k(O)$	$v_k(O)$	$v_k(O)$	$TOTAL$
passive	$prim(O)$	$FIFO$	$prim(O)$	$prim(O)$	$prim(O)$	$prim(O)$	$FIFO$
semi-active	$v_k(O)$	$FIFO$	$v_k(O)$	$prim(O)$	$v_k(O)$	$v_k(O)$	$FIFO$
coord.-cohorte	$v_k(O)$	$FIFO$	$prim(O)$	$prim(O)$	$prim(O)$	$v_k(O)$	$FIFO$

Tableau 4.4: Valeurs des paramètres pour les quatres stratégies de duplication

$ord(O)$ concernent la transmission de la requête (phase 1), $cpu(O)$ concerne le traitement de la requête (phase 2) et $src(O)$ concerne la transmission de la réponse (phase 3). Si O est un client dupliqué, $src(O)$ concerne la transmission de la requête (phase 1) alors que $dst(O)$ et $ord(O)$ concernent la transmission de la réponse (phase 3).

Symétrie entre les phases 1 et 3 Les colonnes correspondant à la phase 1 d'un serveur sont identiques à celles correspondant à la phase 3 d'un client. Les paramètres $dst(O)$ et $ord(O)$ prennent les mêmes valeurs lorsque O est serveur (phase 1) et lorsque O est client (phase 3). Ceci est la conséquence de la modélisation symétrique de l'invocation entre objets dupliqués.

La colonne correspondant à la phase 3 d'un serveur est identique à la colonne correspondant à la phase 1 sauf pour la duplication semi-active. Dans ce cas, $src(O) = prim(O)$ si O est serveur, alors que $src(O) = v_k(O)$ si O est client. Pour chacune des autres stratégies, les valeurs des deux colonnes sont identiques: le même ensemble de copies émet respectivement la requête ou la réponse selon que O est respectivement client ou serveur.

Stratégies hybrides vs stratégies de base Le tableau 4.4 permet de faire le point sur les similitudes entre les stratégies hybrides (duplication semi-active et duplication coordinateur-cohorte) et les stratégies de base (duplication active et duplication passive). Les 16 combinaisons correspondant aux quatre stratégies de duplication sont représentées graphiquement à l'annexe A.

La duplication semi-active se rapproche de la duplication active si l'on considère les paramètres $dst(O)$, $cpu(O)$ et $src(O)$ (lorsque O est client). Elle se rapproche de la duplication passive si l'on considère les paramètres $ord(O)$ et $src(O)$ (lorsque O est serveur).

La duplication coordinateur-cohorte se rapproche de la duplication active si l'on considère le paramètre $dst(O)$. Elle se rapproche de la duplication passive si l'on considère les paramètres $src(O)$, $ord(O)$ et $cpu(O)$.

4.6 Conclusion

Résumé

L'invocation entre objets dupliqués pose le problème de l'encapsulation de la duplication. Ce problème regroupe, les difficultés liées à l'intégration de la notion de duplication avec celle d'invocation, et les difficultés liées à la prise en compte des défaillances.

L'intégration de la notion de duplication avec celle d'invocation se heurte au problème de l'encapsulation de la pluralité et au problème de l'encapsulation des stratégies de duplication. Encapsuler la pluralité d'un objet dupliqué O revient à désigner l'objet O par un mécanisme indépendant de l'ensemble des copies de O, et indépendant de la stratégie de duplication de O. Encapsuler la stratégie de duplication d'un objet O revient à cacher, la primitive de communication qu'il faut utiliser pour transmettre un message à O, et à filtrer les messages en provenance de O.

L'analyse de quelques exemples d'invocation met en évidence une modélisation *asymétrique* de l'invocation entre objets dupliqués. Cette modélisation suppose que la réponse à une invocation est transmise comme un retour d'appel de procédure à distance. Le principal inconvénient de la modélisation asymétrique est qu'elle ne permet pas d'exprimer le problème de l'encapsulation de la pluralité d'un client dupliqué. Ce chapitre propose une alternative appelée modélisation *symétrique*, dont la particularité est de considérer la transmission de la réponse exactement comme la transmission de la requête. La modélisation symétrique permet de prendre en compte le problème de l'encapsulation de la pluralité de tout objet dupliqué, qu'il soit client ou serveur.

Aux problèmes évoqués ci-dessus, s'ajoutent les difficultés liées à la prise en compte des défaillances. Dans un système réparti asynchrone, les informations sur les défaillances sont nécessairement approximatives. Dans cette étude, l'abstraction choisie est celle d'un groupe d'objets à partition primaire doté de la sémantique vue synchrone.

À partir de la modélisation symétrique et de l'abstraction de groupe, une spécification générique de l'invocation entre objets dupliqués a été construite. Les paramètres de cette spécification sont déterminés par les stratégies de duplication des objets participant à l'invocation. Les valeurs de ces paramètres sont obtenues en appliquant la spécification sur chacune des quatre stratégies de duplication considérées dans cette thèse.

Commentaires

Ce chapitre a décrit de manière approfondie les problèmes étudiés au cours de ce travail de thèse. Cette description a permis de définir une spécification, basée sur une modélisation symétrique originale de l'invocation entre objets dupliqués. Le chapitre suivant s'appuie sur cette spécification pour élaborer des solutions aux problèmes décrits dans ce chapitre.

Chapitre 5

N2M, un service d'invocation pour objets dupliqués

5.1 Introduction

Ce chapitre présente N2M, un service d'invocation pour objets dupliqués[1], conçu dans le contexte de cette thèse, afin de répondre aux problèmes décrits au chapitre précédent.

Présentation générale

Le service N2M réalise l'encapsulation de la duplication des objets qui utilisent ce service pour communiquer. En d'autres termes, la duplication de tout objet O utilisant N2M, est rendue complètement transparente aux objets communiquant avec O.

Pour encapsuler la duplication, le service N2M définit l'abstraction de *représentants symétriques*. Cette abstraction consiste à construire, de manière symétrique: un représentant du serveur sur chaque nœud où réside une copie du client, et un représentant du client sur chaque nœud où réside une copie du serveur. Grâce aux représentants symétriques, les objets utilisant N2M communiquent exclusivement par invocations, et ils ne perçoivent ni la répartition ni la duplication.

Les représentants symétriques sont mis en œuvre par les *objets communication*, les objets qui *fournissent* le service N2M. Par opposition, les objets qui *utilisent* le service N2M, sont appelés *objets application*. Les objets communication mettent en œuvre les stratégies de duplication qui sont utilisées pour dupliquer les objets application.

Il existe deux sortes d'objets communication: les encapsulateurs et les messagers. Un *encapsulateur* est associé à chaque copie d'un objet application. Un encapsulateur est un véritable filtre à invocations: il contrôle toutes les requêtes et toutes les réponses, entrant ou sortant de la copie associée. Un *messager* est un représentant local d'un objet dupliqué O. Le rôle d'un messager de O consiste à transmettre les requêtes et les réponses, émises par les objets résidant sur le nœud du messager, vers les copies de l'objet application O.

La dimension générique de l'invocation est prise en compte grâce à la conception d'encapsulateurs

[1] Le schéma de communication correspondant à un multicast est souvent désigné par l'expression anglaise "1 to n". De la même manière, l'expression "n to m" caractérise la communication entre n copies d'un client dupliqué et m copies d'un serveur dupliqué. Le logo N2M fait référence à cette expression.

et de messagers spécialisés. Ainsi, chaque stratégie de duplication est réalisée à l'aide d'une classe de messagers et d'une classe d'encapsulateurs. Ces classes mettent en œuvre des algorithmes satisfaisant les propriétés définies au chapitre précédent.

Organisation

La section 5.2 présente l'abstraction de représentants symétriques. La section 5.3 décrit comment les encapsulateurs et les messagers permettent de modéliser cette abstraction. La section 5.4 présente les solutions au problème de l'encapsulation de la pluralité. La section 5.5 décrit les solutions au problème de l'encapsulation des stratégies de duplication. La section 5.6 présente l'application de ces solutions aux quatre stratégies de duplication. La section 5.7 conclut le chapitre.

5.2 Représentants symétriques

Les représentants symétriques constituent la principale abstraction que définit le service d'invocation N2M afin de réaliser l'encapsulation de la duplication. La section 5.2.1 présente la notion de représentants symétriques. La section 5.2.2 décrit leur rôle au cours du déroulement d'une invocation. La section 5.2.3 compare les représentants symétriques aux mandataires définis dans [Shapiro 86].

5.2.1 Notion de représentants symétriques

L'objectif du service N2M est de réaliser l'encapsulation de la duplication, afin de permettre à des objets dupliqués de communiquer exclusivement à l'aide d'invocations. Pour atteindre cet objectif, le service N2M définit la notion de **représentants symétriques** (*symmetric representatives*). Cette notion est illustrée sur la figure 5.1: un client dupliqué C invoque un serveur dupliqué S. Les n copies du client C sont notées $C_1, C_2, ..., C_n$. Les m copies du serveur sont notées $S_1, S_2, ..., S_m$. Chaque copie réside sur un nœud distinct (rectangle en grisé). Le client et le serveur sont dupliqués activement: chaque copie du client doit invoquer le serveur, et chaque copie du serveur doit être invoquée par le client.

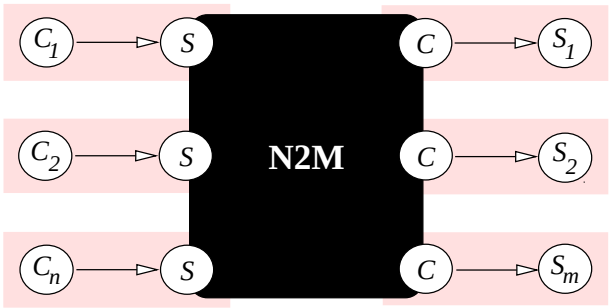

Figure 5.1: Notion de représentants symétriques

Sur chaque nœud où réside une copie du client C, le service N2M construit un représentant du

serveur S. Un représentant du serveur est une abstraction donnant l'illusion à toute copie C_i que S est un objet local et non dupliqué. Pour cette raison, un représentant du serveur est représenté sur la figure 5.1, exactement comme le serait l'objet S s'il n'était pas dupliqué.

De manière *symétrique*, N2M construit un représentant du client sur chaque nœud où réside une copie du serveur. Un représentant du client est une abstraction donnant l'illusion à toute copie S_j que C est un objet local et non dupliqué. Pour cette raison, un représentant du client est représenté sur la figure 5.1, exactement comme le serait l'objet C s'il n'était pas dupliqué.

Pour invoquer le serveur S, toute copie C_i invoque (flèche à pointe creuse) le représentant local de S. Le service N2M "propage" cette invocation sur tous les nœuds abritant une copie de S. Sur chacun de ces nœuds, le représentant de C invoque (flèche à point creuse) S_j, la copie locale de S. Tout se passe comme si l'invocation de toute copie S_j était le prolongement d'une invocation faite par une copie C_i. Grâce aux représentants symétriques, toute copie C_i ou S_j perçoit l'invocation comme une invocation locale.

Sur la figure 5.1, chaque flèche à pointe creuse représente une invocation locale: la requête d'invocation et la réponse sont échangées à l'aide du mécanisme d'invocation intégré à tout langage de programmation orientée-objets[2].

Une invocation exprime une communication bi-directionnelle entre un client et un serveur. Le client transmet une requête au serveur qui traite la requête, puis transmet la réponse correspondante au client. Pourtant, une invocation est habituellement représentée par une flèche allant du client vers le serveur. Cette représentation vient du fait qu'une invocation est toujours commencée à l'initiative du client. Le serveur ne fait que réagir à l'invocation.

L'exemple de la figure 5.1 met en évidence deux catégories distinctes d'objets: les objets qui *utilisent* le service N2M et les objets qui *réalisent* le service N2M.

Les objets qui utilisent le service N2M sont appelés **objets application** (*application objects*). Les copies de C et de S sont des objets application. Grâce à la notion de représentants symétriques, les objets application ont l'impression d'appartenir au même espace adressage. Ils communiquent exclusivement par invocations et ne perçoivent ni la répartition, ni la duplication. Cette propriété des objets application facilite la programmation d'applications réparties tolérantes aux fautes. Le programmeur peut se concentrer sur la mise en œuvre de la sémantique de l'application, sans se préoccuper des aspects relatifs à la répartition et à la duplication.

Les objets qui réalisent le service N2M sont appelés **objets communication** (*communication objects*). Les représentants de C et de S sont des objets communication. Les objets communication gèrent tous les aspects relatifs à la répartition et à la duplication afin de mettre en œuvre l'abstraction d'espace d'adressage unique pour le compte des objets application.

Les objets communication mettent en œuvre l'abstraction de représentants symétriques. Ils réalisent l'encapsulation de la duplication des objets application:

- les copies du client (resp. serveur) n'ont aucun moyen de percevoir que le serveur (resp. client) est constitué par un groupe de copies: la pluralité du client et la pluralité du serveur sont encapsulées;
- l'influence des stratégies de duplication du client et du serveur sur la communication est cachée aux copies du client et à celles du serveur: les stratégies de duplication du client et du serveur sont encapsulées.

[2]Ce mécanisme correspond à l'appel de procédure des langages de programmation procéduraux.

5.2.2 Rôle des représentants symétriques

La figure 5.2 reprend l'exemple de la figure 5.1 afin d'illustrer le rôle des représentants symétriques dans le déroulement d'une invocation. Pour mieux les distinguer des objets application, les représentants symétriques sont représentés à l'aide de rectangles noirs aux coins arrondis. Les représentants de S (resp. C) portent l'étiquette $rep(S)$ (resp. $rep(C)$).

La séquence des étiquettes numériques de la figure 5.2 permet de suivre le déroulement[3] de l'invocation de S par C. Sur cet exemple, le client C et le serveur S sont dupliqués activement.

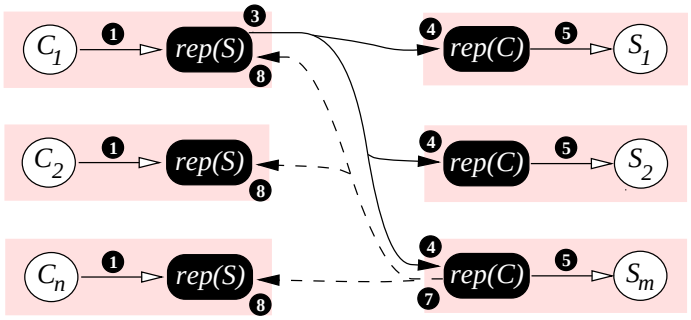

Figure 5.2: Rôle des représentants symétriques

La duplication active confère un rôle identique à chaque copie, ce qui simplifie l'exposé sans nuire à sa généralité.

Chaque copie C_i invoque S comme si ce dernier était un objet local. En réalité, C_i invoque le représentant $rep(S)$ (étiquette 1). Cet objet communication connaît la stratégie de duplication de S: il est capable de localiser les copies de S et de leur transmettre la requête. Comme *toutes* les copies de C invoquent S, il peut paraître surprenant que le représentant $rep(S)$ résidant sur le nœud de C_1 (on le note $rep(S)(C_1)$), soit le *seul* à émettre la requête (étiquette 3) en direction des copies du serveur.

Les représentants $rep(S)$ agissent comme de véritables filtres à invocations. Le filtrage des invocations sert à garantir la propriété d'intégrité (cf. section 4.3.4) lors de la transmission de la requête. Le filtrage des invocations peut être réalisé selon deux approches: le *pré-filtrage* (ou filtrage à l'émission) et le *post-filtrage* (ou filtrage à la réception). Ces approches sont détaillées à la section 5.5.

Sur cet exemple, les représentants $rep(S)$ utilisent le pré-filtrage. Chaque $rep(S)(C_i)$ est invoqué par la copie C_i mais seul le coordinateur $rep(S)(C_1)$ émet la requête. Sur les nœuds des copies de S, le représentant $rep(C)(S_j)$ reçoit la requête émise par $rep(S)(C_1)$ (étiquette 4), puis, il invoque S_j (étiquette 5). Au terme de cette invocation, chaque représentant $rep(C)(S_j)$ obtient une réponse qu'il doit transmettre aux copies du client.

Les représentants de C utilise également le pré-filtrage pour transmettre la réponse aux copies de C. Ainsi, le coordinateur[4] $rep(C)(S_m)$ est le seul à émettre la réponse (étiquette 7). Sur les nœuds des copies de C, le représentant $rep(S)(C_i)$ reçoit la réponse (étiquette 8) émise par

[3] Le lecteur attentif aura remarqué des "trous" dans la numérotation. Ces trous correspondent à des étapes invisibles sur cette figure, et qui seront dévoilées sur la figure 5.4.

[4] Tout représentant peut être coordinateur.

$rep(C)(S_m)$, et il la communique à C_i par retour de l'invocation initiale (étiquette 1).

5.2.3 Représentants vs mandataires

Un représentant s'apparente à un mandataire (cf. section 3.4.2). Un représentant d'un objet dupliqué O réalise l'encapsulation de la duplication de O. Tout objet communiquant avec O ne peut distinguer $rep(O)$, le représentant local de O, de l'objet dupliqué O dont les copies sont réparties sur des nœuds distants. De façon analogue, un mandataire d'un serveur dupliqué O réalise l'encapsulation de la duplication de O. Le mandataire donne l'illusion, à tout client de O résidant sur le nœud du mandataire, que l'objet O est un objet local et non dupliqué.

Cependant, un représentant se distingue d'un mandataire par deux aspects:

- la notion de représentant est symétrique alors que celle de mandataire est asymétrique;
- le comportement d'un représentant dépend à la fois du client *et* du serveur alors que celui d'un mandataire ne dépend que du serveur.

Symétrie des représentants vs asymétrie des mandataires

Les représentants sont *symétriques*: il y a un représentant du serveur sur le nœud de chaque copie du client *et* un représentant du client sur le nœud de chaque copie du serveur. Cette abstraction s'inscrit dans le contexte de la modélisation symétrique de l'invocation entre objets dupliqués. Elle permet de contrôler finement le schéma de communication entre les copies du client et celles du serveur. Précisément, le message transportant la réponse à une invocation est totalement *indépendant* du message transportant la requête. L'indépendance des messages signifie que la réponse ne suit pas *systématiquement* le trajet inverse de la requête. Le schéma de communication entre le client et le serveur est alors déterminé par les stratégies de duplication du client et du serveur.

Sur la figure 5.2, la requête va de C_1 vers S_1, S_2, S_m, tandis que la réponse va de S_m vers C_1, C_2, C_n. Comme S (resp. C) est dupliqué activement, la requête (resp. réponse) est transmise à toutes les copies de S (resp. C). Le pré-filtrage garantit la propriété d'intégrité lors de la transmission de la requête (resp. réponse).

Contrairement aux représentants, les mandataires sont *asymétriques*: il y a un mandataire du serveur sur le nœud de chaque copie du client mais il n'y a pas de mandataire du client sur les nœuds des copies du serveur. Un mandataire représente *toujours* un serveur. Cette abstraction s'inscrit dans le contexte de la modélisation asymétrique (usuelle) de l'invocation entre objets dupliqués. Le schéma de communication entre les copies du client et du serveur est figé: le message transportant la réponse à une invocation suit systématiquement le trajet inverse du message transportant la requête. Les copies du serveur jouent un rôle privilégié: elles sont conscientes de la répartition puisqu'elles doivent être capables de communiquer avec des mandataires distants. Le schéma de communication entre le client et le serveur est contrôlé par les copies du serveur et par les mandataires. Les copies du client perçoivent l'invocation du serveur comme une invocation locale: la duplication du serveur est encapsulée par l'abstraction de mandataire.

La figure 5.3 illustre la différence entre représentants et mandataires, en exprimant l'exemple de la figure précédente avec des mandataires. Sur le nœud de chaque copie C_i, il y a un mandataire

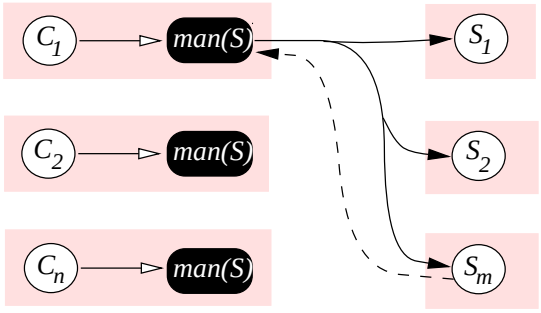

Figure 5.3: Asymétrie des mandataires

du serveur $man(S)$. Par contre, il n'y a pas de mandataire du client sur le nœud de chaque copie S_j. Chaque copie C_i invoque le mandataire $man(S)$ résidant sur le même nœud. Comme sur la figure 5.2, le pré-filtrage est utilisé pour garantir la propriété d'intégrité sur la transmission de la requête et sur la transmission de la réponse. Par conséquent, le mandataire $m(S)(C_1)$ est le seul à transmettre la requête vers les copies de S. Le message transportant la requête va donc de C_1 vers S_1, S_2, S_m. La copie S_m est la seule à transmettre la réponse. Comme la réponse doit suivre le trajet inverse de la requête, la réponse va de S_m vers C_1. Par conséquent, les copies C_2 et C_3 ne reçoivent pas de réponse!

Ce petit exemple montre que l'abstraction de mandataire ne permet pas d'utiliser le pré-filtrage pour garantir la propriété d'intégrité. Cette impossibilité correspond au fait que la modélisation asymétrique de l'invocation (sur laquelle est basée l'abstraction de mandataire) ne permet pas d'exprimer le problème de l'encapsulation de la pluralité d'un client dupliqué.

Comportement d'un représentant vs comportement d'un mandataire

Le comportement d'un mandataire est complètement déterminé par le serveur que le mandataire représente. En particulier, la stratégie de duplication du serveur définit le schéma de communication entre le mandataire et les copies du serveur. Le client interagit avec le mandataire à l'aide d'invocations locales. Il n'a donc aucune influence sur le comportement du mandataire.

Contrairement au mandataire, le comportement d'un représentant dépend à la fois du client et du serveur. L'analyse de la figure 5.2 permet de s'en convaincre.

Les représentants de S sont chargés de transmettre la requête d'invocation depuis les copies de C vers les copies de S. La transmission de la requête est conditionnée par les paramètres $dst(S)$, $ord(S)$ et $src(C)$ (cf. section 4.3.4). Les deux premiers paramètres dépendent de $str(S)$, la stratégie de duplication de S. Ils désignent respectivement l'ensemble des copies de S qui doivent délivrer la requête, et le critère d'ordre à utiliser pour cette délivrance. Pour pouvoir transmettre la requête, tout représentant de S doit donc connaître $str(S)$.

Le troisième paramètre dépend de $str(C)$, la stratégie de duplication du client. Il désigne l'ensemble des copies de C qui émettent la requête. Ce paramètre est important car il conditionne le filtrage des requêtes. En effet, les représentants de S effectuent le pré-filtrage des requêtes à cause de la duplication active du client C. Ainsi, tout représentant de S doit connaître $str(C)$ pour effectuer le pré-filtrage.

Comme tout représentant $rep(S)$ doit connaître $str(S)$ et $str(C)$, son comportement dépend à la fois du serveur S et du client C. La même remarque s'applique à tout représentant $rep(C)$. Pour s'en convaincre, il suffit de considérer la transmission de la réponse.

5.3 Encapsulateurs et messagers

Les encapsulateurs et les messagers sont des objets communication spécialisés qui réalisent l'abstraction de représentants symétriques. La section 5.3.1 présente la modélisation des représentants symétriques en termes d'encapsulateurs et de messagers. La section 5.3.2 décrit la notion d'encapsulateur. La section 5.3.3 présente la notion de messager.

5.3.1 Modélisation des représentants symétriques

Le comportement d'un représentant symétrique dépend à la fois du client et du serveur. Afin de respecter la propriété d'encapsulation des objets, le comportement dépendant du client, doit être isolé du comportement dépendant du serveur. Par conséquent, un représentant symétrique est modélisé en utilisant deux objets communication distincts: un *encapsulateur* et un *messager*. La figure 5.4 illustre la modélisation des représentants symétriques. Elle reprend l'exemple de la

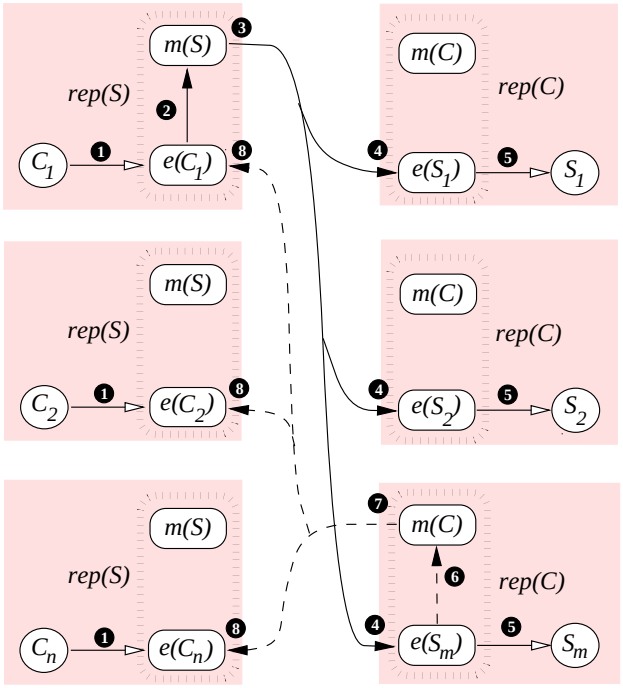

Figure 5.4: Modélisation des représentants symétriques

figure 5.2: le client C et le serveur S sont dupliqués activement et les invocations sont pré-filtrées.

Sur le nœud de chaque copie C_i, le représentant du serveur, $rep(S)$, est modélisé à l'aide de l'encapsulateur $e(C_i)$ et du messager $m(S)$. Le messager $m(S)$ modélise le comportement dépendant du serveur S. Il est chargé de transmettre la requête en respectant les valeurs

des paramètres imposées par $str(S)$, la stratégie de duplication du serveur S. L'encapsulateur $e(C_i)$ modélise le comportement dépendant du client C. Il est chargé de filtrer les requêtes d'invocation en tenant compte de $str(C)$, la stratégie de duplication du client C.

De façon symétrique, sur le nœud de chaque copie S_j, le représentant du client, $rep(C)$, est modélisé à l'aide de l'encapsulateur $e(S_j)$ et du messager $m(C)$. Le messager $m(C)$ modélise le comportement dépendant du client C. Il est chargé de transmettre la réponse en respectant les valeurs des paramètres imposées par $str(C)$. L'encapsulateur $e(S_j)$ modélise le comportement dépendant du serveur S. Il est chargé de filtrer les réponses à l'invocation en tenant compte de $str(S)$.

Cette modélisation des représentants symétriques repose sur l'interaction entre des messagers et des encapsulateurs, dont les classes peuvent être a priori quelconques. Pour que cette interaction soit possible, tout encapsulateur et tout messager doit présenter une interface standard. Les méthodes *outRequest*(), *inReply*() et *inRequest*() constituent l'interface standard d'un encapsulateur. Les méthodes *sendRequest*() et *sendReply*() constituent l'interface standard d'un messager.

La figure 5.4 permet de suivre le déroulement d'une invocation et d'avoir une première idée du rôle de chaque méthode évoquée ci-dessus. Plus de détails sont donnés à la section 5.3.2 et à la section 5.3.3. Une invocation se déroule toujours en trois phases:

1. **Transmission de la requête:** comme le client C est dupliqué activement, chaque copie C_i invoque le serveur S. Lorsqu'une copie C_i invoque S, la méthode *outRequest*() de l'encapsulateur $e(C_i)$ est appelée automatiquement[5]. Au cours de l'exécution de la méthode *outRequest*(), chaque $e(C_i)$ pré-filtre les requêtes d'invocation redondantes. En outre, le coordinateur $e(C_1)$ invoque la méthode *sendRequest*() du messager $m(S)$ (étiquette 2) pour lui transmettre la requête d'invocation, émise par C_1. Dès lors, pour chaque $e(C_i)$, l'exécution de *outRequest*() est bloquée en attente de la réponse à l'invocation.

   Le messager $m(S)$ effectue la transmission de la requête (étiquette 3) vers les encapsulateurs des copies de S en respectant les valeurs des paramètres fixés par $str(S)$. Ainsi, la requête est transmise aux encapsulateurs de toutes les copies de S à l'aide d'un multicast respectant l'ordre total.

2. **Traitement de la requête:** lorsque la requête d'invocation parvient sur le nœud d'une copie S_j, elle est délivrée à $e(S_j)$ par l'appel de la méthode *inRequest*() (étiquette 4). Puis, l'encapsulateur $e(S_j)$ utilise cette requête pour invoquer S_j (étiquette 5). Lorsque l'invocation se termine, $e(S_j)$ récupère la réponse produite par S_j.

3. **Transmission de la réponse:** au cours de l'exécution de la méthode *inRequest*(), chaque $e(S_j)$ pré-filtre les réponses redondantes. En outre, le coordinateur $e(S_m)$ invoque la méthode *sendReply*() du messager $m(C)$ (étiquette 6) pour lui transmettre la réponse produite par S_m.

   Le messager $m(C)$ effectue la transmission de la réponse (étiquette 7) vers les encapsulateurs des copies de C en respectant les valeurs des paramètres fixés par $str(C)$. Ainsi, la réponse est transmise aux encapsulateurs de toutes les copies de C à l'aide d'un multicast respectant l'ordre total.

[5]Ce mécanisme est expliqué plus loin à la section 5.4.3.

Lorsque la réponse parvient sur le nœud d'une copie C_i, elle est délivrée à $e(C_i)$ par l'appel de la méthode $inReply()$ (étiquette 8). Cette délivrance provoque la terminaison de la méthode $outRequest()$: C_i obtient ainsi la réponse à l'invocation initiale (étiquette 1).

5.3.2 Notion d'encapsulateur

Un **encapsulateur** (*encapsulator*) associé à un objet application O, est un objet communication dont le rôle consiste à *filtrer* toutes les invocations auxquelles O participe, soit comme client, soit comme serveur. L'encapsulateur associé à O est noté $e(O)$.

Un encapsulateur peut être vu comme une sorte de capsule ou de coquille enveloppant un objet application. Il est donc situé sur le même nœud que son objet application. Sa durée de vie correspond à celle de son objet application: il est créé et détruit en même temps que son objet application.

Les traitements que réalisent un encapsulateur $e(O)$ varient d'une part, selon la classe de l'encapsulateur, et d'autre part selon que O est client ou serveur pour cette invocation.

Rôle de l'encapsulateur d'un client

Lorsque O est client, les traitements effectués par $e(O)$ correspondent à l'exécution des méthodes $outRequest()$ et $inReply()$.

La méthode $outRequest()$ réalise le traitement que l'encapsulateur effectue *avant* de transmettre la requête d'invocation au messager du serveur. Cette méthode est appelée automatiquement par N2M à chaque fois que le client O invoque un serveur. Tout se passe comme si O avait invoqué $e(O)$. Les paramètres d'entrée de $outRequest()$ sont: le nom du serveur, le nom de l'opération et la liste des paramètres éventuels. La valeur de retour de $outRequest()$ est la réponse renvoyée par le serveur invoqué par O. Le traitement réalisé par $outRequest()$ est structuré comme suit:

1. traitement spécifique à la classe de $e(O)$;
2. invocation du messager du serveur (méthode $sendRequest()$);
3. attente de la réponse à l'aide d'un sémaphore de synchronisation privé à l'invocation;
4. terminaison de $outRequest()$: la réponse constitue la valeur de retour de $outRequest()$.

La méthode $inReply()$ correspond au traitement que l'encapsulateur effectue *avant* de délivrer la réponse au client. Cette méthode est invoquée au moment où la réponse à l'invocation parvient sur le nœud de O. Le traitement consiste principalement à débloquer la méthode $outRequest()$ en attente sur le sémaphore de synchronisation.

Rôle de l'encapsulateur d'un serveur

Lorsque O est serveur, les traitements effectués par $e(O)$ correspondent à l'exécution de la méthode $inRequest()$. Cette méthode correspond au traitement qu'un encapsulateur effectue *avant* de délivrer la requête au serveur. Elle est invoquée au moment où une requête d'invocation parvient sur le nœud de O.

L'encapsulateur $e(O)$ délivre la requête d'invocation transmise par un messager distant, puis il invoque l'objet O à l'aide de cette requête. Lorsque l'invocation de O est terminée, $e(O)$ récupère la réponse de cette invocation et la transmet au messager du client (méthode $sendReply()$).

5.3.3 Notion de messager

Un **messager** (*mailer*) associé à un objet application O, est un objet communication chargé de *transmettre*, vers les encapsulateurs des copies de O, les messages (i.e. requêtes et réponses) émis par les encapsulateurs des objets application locaux. On note $m(O)$ le messager associé à O.

Un messager est une sorte de mandataire symétrique: contrairement à un mandataire, un messager $m(O)$ transmet un message (requête ou réponse) *vers* l'objet O mais ne délivre pas de message provenant de O. Un messager $m(O)$ est créé sur chaque nœud où il existe un objet application suceptible d'envoyer des messages à O.

Le caractère symétrique du rôle d'un messager se retrouve dans les traitements qu'il effectue. Lorsque O est serveur, $m(O)$ est chargé de transmettre la requête d'invocation vers les encapsulateurs des copies de O. Ce traitement est assuré par la méthode $sendRequest()$. Lorsque O est client, $m(O)$ est chargé de transmettre la réponse à l'invocation vers les encapsulateurs des copies de O. Ce traitement est assuré par la méthode $sendReply()$.

5.4 Encapsulation de la pluralité dans le service N2M

Cette section décrit les solutions du service N2M, au problème de l'encapsulation de la pluralité. La section 5.4.1 situe la problématique dans le contexte du service N2M. La section 5.4.2 présente la structure des objets application dupliqués. La section 5.4.3 décrit les mécanismes permettant de désigner ces objets. La section 5.4.4 fait le point sur les solutions présentées.

5.4.1 Introduction

Encapsuler la pluralité d'un objet dupliqué O consiste à cacher, aux objets communiquant avec O, le fait que O est un *groupe* d'objets répartis sur plusieurs nœuds. Comme l'a montré la section 4.3.2, le problème de l'encapsulation de la pluralité peut être ramené à celui de la désignation d'un objet dupliqué. Si O peut être désigné à l'aide d'un identificateur indépendant du nombre de ses copies, de leur répartition, et de sa stratégie de duplication alors la pluralité de O est encapsulée. Au cours d'une invocation, le problème de la désignation d'un objet dupliqué se pose pendant la transmission de la requête (désignation du serveur) et pendant la transmission de la réponse (désignation du client).

Dans le contexte du service N2M, la pluralité des objets application dupliqués est encapsulée par les objets communication. Plus précisément, un objet application dupliqué est construit autour d'un groupe d'encapsulateurs dont l'interface est réalisé, sur chaque nœud, par un messager. Cette structure permet de construire un mécanisme de désignation basé sur les identificateurs de groupe.

5.4.2 Structure des objets application dupliqués

De manière générale, un objet dupliqué est constitué par un ensemble de copies réparties sur plusieurs nœuds. Dans le contexte de N2M, chaque copie d'un objet application dupliqué est un objet application. Par conséquent[6], les copies d'un objet application ne sont pas conscientes qu'elles gèrent un état dupliqué.

Le lien entre les copies d'un objet application dupliqué est réalisé par les encapsulateurs associés à ces copies. En effet, tous les encapsulateurs associés à une copie d'un objet application O appartiennent à un **groupe d'encapsulateurs** (*encapsulator group*), noté $g(O)$. La figure 5.5

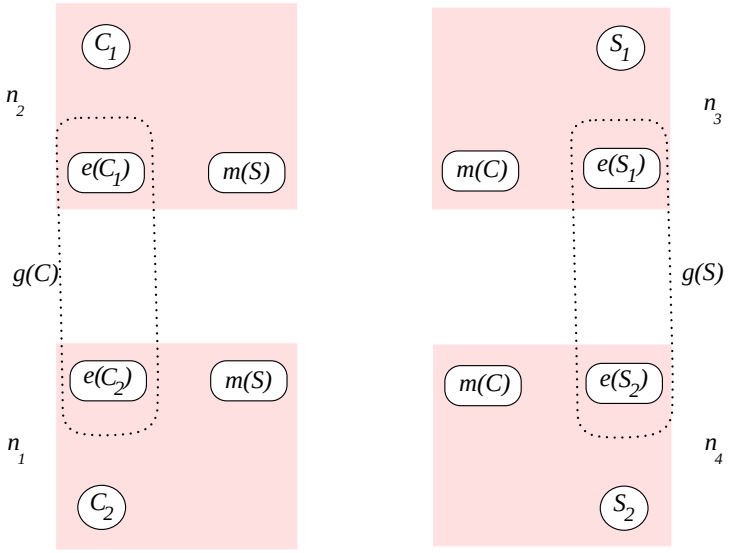

Figure 5.5: Structure des objets application dupliqués

illustre la structure des objets application dupliqués. Deux objets application dupliqués, C et S, sont représentés. Les encapsulateurs des copies de C sont notés $e(C_1)$ et $e(C_2)$. Ils forment le groupe $g(C)$, illustré par un contour en pointillé. De même, les encapsulateurs des copies de S forment le groupe $g(S)$.

Un groupe d'encapsulateurs réalise la sémantique décrite à la section 4.4.1. Il se caractérise par:

- une *composition dynamique* exprimée sous la forme d'une séquence de vues délivrée à tous les membres du groupe. Chaque vue contient la liste des identificateurs des encapsulateurs appartenant au groupe à cet instant logique. Une nouvelle vue est construite à chaque fois qu'un encapsulateur rejoint ou quitte le groupe.
- *un identificateur unique et global* à tout le système.

Cette sémantique peut se résumer par les méthodes $join()$, $viewChange()$ et $leave()$. Elles correspondent aux traitements que tout encapsulateur doit réaliser pour respectivement, rejoindre un groupe, traiter un changement de vue et quitter un groupe. Pour réaliser ces méthodes, les

[6] Un objet application ne perçoit ni la répartition, ni la duplication (cf. section 5.2.1).

classes d'encapsulateurs s'appuient sur un service de gestion de groupes *indépendant* de N2M. Des classes d'objets communication spécialisés assurent l'interface entre N2M et le service de gestion de groupes. Dès lors, le service N2M peut utiliser n'importe quel service de gestion de groupes, à condition qu'il permette de réaliser la sémantique décrite ci-dessus. Une mise en œuvre de N2M basée sur le service de groupes fourni par ISIS est décrite au chapitre 6.

Comme on l'a vu à la section 5.2.1, la notion de messager symétrique joue un rôle crucial dans le schéma de communication avec un objet application dupliqué. Un messager permet de communiquer avec un objet application dupliqué tout en ignorant le nombre et la répartition des copies de cet objet, ainsi que sa stratégie de duplication. Sur la figure 5.5, il y a un messager $m(S)$ sur chaque nœud où réside une copie de C, c.-à-d. sur les nœuds n_1 et n_2.

Tout objet désireux de transmettre un message (requête ou réponse) à l'objet S s'adresse au messager local. Par exemple, tout message en provenance d'une copie C_i est transmis à S en passant d'abord par $e(C_i)$, puis par $m(S)$. Ce dernier transmet le message au groupe $g(S)$ en suivant le protocole défini par la stratégie de duplication de S. Si une copie de S désire transmettre un message à C, elle devra s'adresser au messager de C résidant sur le même nœud.

5.4.3 Désignation des objets application dupliqués

La désignation des objets application dupliqués comporte deux niveaux:

- une désignation *locale* chargée d'assurer l'interface entre les objets application et les objets communication résidant sur un même nœud;
- une désignation *globale* chargée d'identifier uniquement les objets communication de façon à pouvoir leur transmettre les messages correspondant aux invocations (i.e. requêtes et réponses).

Désignation locale

La désignation locale est mise en œuvre par des objets noms spécialisés appelés références. Une **référence** (*reference*) sur un objet application dupliqué O, est un objet nom dont l'état permet de retrouver le messager local associé à O. On la note $ref(O)$. Lorsqu'un client invoque O, il invoque en réalité $ref(O)$. Pour le client, O et $ref(O)$ sont indiscernables. Le client a ainsi l'impression d'invoquer un objet application local et non dupliqué.

Une référence $ref(O)$ représente l'objet application O dans l'espace d'adressage du nœud sur lequel elle réside. Elle complète ainsi le rôle du messager. La notion de référence permet de cacher complètement les objets communication (i.e. encapsulateurs et messagers) aux objets application.

Désignation globale

La désignation globale est mise en œuvre par des objets noms spécialisés appelés identificateurs globaux. Un **identificateur global** (*global identifier*) est un objet nom permettant de désigner un objet communication (i.e. encapsulateur ou messager) de manière unique. Il est composé de trois informations:

- l'identificateur du groupe;
- l'identificateur du nœud;
- la nature de l'objet communication: encapsulateur ou messager.

Ces trois informations suffisent pour désigner de manière unique un objet communication. En effet, un objet communication ne participe qu'à la mise en œuvre d'un *seul* objet application dupliqué. Un encapsulateur est associé à un objet application unique, copie d'un objet application dupliqué. De même, un messager ne sert d'interface qu'à un seul objet application dupliqué. Par conséquent, tout objet communication n'est lié qu'à un seul groupe d'encapsulateurs.

Étant donnés un nœud n_i et un objet application dupliqué O, il y a au plus un encapsulateur $e(O_i)$ et au plus un messager $m(O)$ sur le nœud n_i. Si $e(O_i)$ existe, il est unique car il ne peut y avoir plus d'une copie de O sur n_i. Si $m(O)$ existe, il est unique car tout objet application dupliqué est représenté au plus une fois sur un nœud donné.

Sur chaque nœud, la table des objets communication permet de retrouver un objet communication à partir de son identificateur global. Cette table comporte trois colonnes:

- l'identificateur de groupe;
- l'adresse de l'encapsulateur de la copie locale;
- l'adresse du messager.

Sur l'exemple de la figure 5.5, la table des objets communication du nœud n_1 a l'allure suivante:

identificateur de groupe	adresse de l'encapsulateur	adresse du messager
$g(C)$	$e(C_2)$	-
$g(S)$	-	$m(S)$

Tableau 5.1: Un exemple de table des objets communication

Désignation des objets au cours d'une invocation

La figure 5.6 représente les quatre premières étapes d'une invocation au cours de laquelle C_1, une copie d'un client C, invoque l'opération *op*() d'un serveur dupliqué S. Cette vue simplifiée suffit pour illustrer l'utilisation des niveaux de désignation au cours d'une invocation. Les étapes 1 et 2 montrent comment une invocation d'un objet application dupliqué est interceptée par l'encapsulateur de l'appelant. Les étapes 3 et 4 illustrent le rôle de la désignation au cours de la transmission d'une requête. Elles s'appliquent également, par symétrie, à la transmission de la réponse.

1. $ref(S).op()$
   La copie C_1 invoque la référence de S (représentée par un cercle en trait discontinu). Toute invocation d'une référence est interceptée afin d'être redirigée vers l'encapsulateur

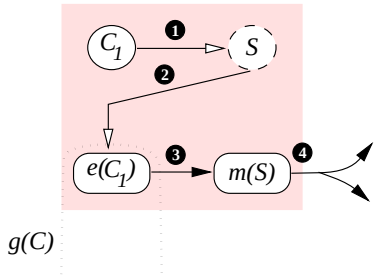

Figure 5.6: Désignation des objets au cours d'une invocation

de l'objet appelant. Cette interception permet de cacher complètement les objets communication aux objets application. L'illusion d'espace d'adressage unique est alors totale! Le mécanisme d'interception peut être mis en œuvre selon deux approches: par génération de code lors de la compilation, ou en utilisant un mécanisme d'exceptions à l'exécution.

Avant d'invoquer l'encapsulateur de C_1, le mécanisme d'interception réifie l'invocation: il construit un objet $reified(op)$ contenant le nom de l'opération invoquée ainsi que les paramètres éventuels.

2. $e(C_1).outRequest(ref(S), reified(op))$
Le mécanisme d'interception invoque $e(C_1)$. En utilisant $ref(S)$ et la table des objets communication, l'encapsulateur $e(C_1)$ détermine $m(S)$, l'adresse du messager du serveur.

3. et 4. $m(S).sendRequest(reified(op))$
L'encapsulateur $e(C_1)$ invoque le messager $m(S)$. Selon la stratégie de duplication de S, le messager $m(S)$ transmet (étape 4) la requête à toutes les copies (en utilisant l'identificateur de groupe) ou bien uniquement à la copie primaire (en utilisant l'identificateur global).

5.4.4 Synthèse

La solution apportée par N2M au problème de l'encapsulation de la pluralité repose sur trois principes:

1. un objet application dupliqué est structuré à partir d'un groupe d'encapsulateurs;
2. le messager symétrique permet de cacher la composition du groupe d'encapsulateurs aux autres objets communication;
3. le mécanisme d'interception des invocations permet de cacher aux objets application l'existence des objets communication.

5.5 Encapsulation des stratégies de duplication dans le service N2M

Cette section présente les solutions de N2M, au problème de l'encapsulation des stratégies de duplication. La section 5.5.1 situe la problématique dans le contexte du service N2M. La sec-

tion 5.5.2 décrit la désignation des messages liés aux invocations. La section 5.5.3 présente le rôle des messagers dans le choix des multicasts utilisés pour transmettre les messages. Les deux sections suivantes décrivent deux approches pour réaliser le filtrage des invocations: le pré-filtrage (cf. section 5.5.4) et le post-filtrage (cf. section 5.5.5). La section 5.5.6 compare ces deux approches. La section 5.5.7 fait la synthèse des solutions présentées.

5.5.1 Introduction

Encapsuler la stratégie de duplication d'un objet application dupliqué O consiste à cacher, aux objets communiquant avec O, le protocole qu'il faut utiliser pour communiquer avec O. Ce protocole est défini par la stratégie de duplication de O afin de garantir la cohérence des copies de O. La section 4.3.2 a montré que deux conditions doivent être satisfaites pour encapsuler la stratégie de duplication de O. D'une part, le choix de la primitive de communication utilisée pour transmettre des messages à O doit être transparent aux objets communiquant avec O. D'autre part, les messages en provenance de O doivent être filtrés.

La première condition permet de résoudre le problème de l'ordre de délivrance des messages. La seconde condition permet de résoudre le problème du nombre variable de messages. Au cours d'une invocation, ces deux problèmes se posent pendant la transmission de la requête (nombre variable de requêtes et ordre de délivrance des requêtes) et pendant la transmission de la réponse (nombre variable de réponses et ordre de délivrance des réponses).

Dans le contexte du service N2M, les objets communication réalisent l'encapsulation des stratégies de duplication des objets application dupliqués. Les encapsulateurs mettent en œuvre deux stratégies de filtrage (le pré-filtrage et le post-filtrage), tandis que les messagers se chargent de cacher le choix des primitives de communication.

5.5.2 Désignation des messages liés aux invocations

Sur le plan de la communication, une invocation correspond à un échange de messages entre un client et serveur. Les messages allant du client vers le serveur transportent les requêtes d'invocation, tandis que les messages en sens inverse transportent les réponses. Lorsque le client et le serveur sont dupliqués, plusieurs messages distincts peuvent transporter la même requête (ou la même réponse).

Cette sémantique a été décrite au chapitre 4 avec les notations suivantes. Soient un client dupliqué C et un serveur dupliqué S, tels que:

- α désigne la requête d'invocation;
- β désigne la réponse;
- $\alpha(C_i)$ désigne le message transportant l'exemplaire de la requête α émis par la copie C_i;
- $\beta(S_j)$ désigne le message transportant l'exemplaire de la réponse β émis par la copie S_j;

Deux messages distincts $\alpha(C_k)$ et $\alpha(C_{k'})$ transportent la même requête d'invocation. Par conséquent, un seul d'entre eux doit être délivré aux copies du serveur. Il en est de même pour les messages transportant les réponses. Pour réaliser cette sémantique, les invocations et les messages s'y rapportant, doivent être désignés de manière unique.

Pour désigner une invocation de manière unique, trois informations suffisent:

- l'identificateur du client;
- l'identificateur du serveur;
- un numéro d'invocation attribué par le client.

Dans N2M, l'identificateur du client (resp. serveur) correspond à l'identificateur unique du groupe auxquels appartiennent les encapsulateurs des copies du client (resp. serveur). Le numéro d'invocation est un numéro de séquence, généré par l'encapsulateur de la copie du client qui effectue une nouvelle invocation. Si plusieurs copies effectuent la même invocation, le déterminisme des copies garantit que le même numéro sera généré par tous les encapsulateurs.

À partir de l'identificateur d'une invocation, il est aisé de construire l'identificateur d'un message transportant une requête ou une réponse, liée à cette invocation. L'identificateur unique d'un message est composé de:

- l'identificateur de l'invocation;
- la nature du message: requête ou réponse.
- l'identificateur global de l'encapsulateur ayant émis le message.

5.5.3 Choix des multicasts

Pour transmettre les messages (requêtes et réponses) aux groupes d'encapsulateurs, les messagers utilisent des multicasts garantissant la sémantique vue-synchrone et l'uniformité. Cette sémantique permet de mettre en œuvre la spécification présentée à la section 4.4.3. Cette spécification s'apparente à celle d'un multicast fiable uniforme ordonné, dont le critère d'ordre varie selon la stratégie de duplication considérée. Les messagers permettent de cacher sous une interface standard (i.e. méthodes *sendRequest*() et *sendReply*()) le choix du critère d'ordre. Ils encapsulent le critère d'ordre, comme ils encapsulent l'identité et la composition du groupe d'encapsulateurs auquel est destiné le message.

Les multicasts sont mis en œuvre par un service indépendant de N2M. La seule contrainte sur ce service est qu'il doit pouvoir interagir avec le service de gestion de groupe sur lequel sont basés les groupes d'encapsulateurs. Cette approche présente l'intérêt de pouvoir réutiliser, dans N2M, des algorithmes de multicasts développés dans d'autres contextes. Dans la mise en œuvre présentée au chapitre 6, le service de multicasts est fourni par le logiciel ISIS.

L'inconvénient de cette approche est que les algorithmes chargés d'ordonner les messages ne perçoivent pas la notion d'invocation. Ainsi, deux messages *distincts* $\alpha(C_k)$ et $\alpha(C_{k'})$ risquent d'être ordonnés inutilement. Ces messages contiennent chacun un exemplaire de la même requête, et l'un des deux sera donc ignoré. On notera que l'occurence de ce problème dépend de la technique de filtrage utilisée. Si les messages redondants sont filtrés à l'émission (pré-filtrage) le problème ne pose pas. Par contre, si les messages sont filtrés à la réception (post-filtrage), le problème se pose.

5.5.4 Pré-filtrage

Comme on l'aura remarqué sur les exemples précédents, le pré-filtrage est basé sur la notion de coordinateur. Étant donné un objet application dupliqué O, lorsque les copies de O émettent une requête ou une réponse, leurs encapsulateurs enregistrent ce message. Puis, l'un des encapsulateurs, appelé coordinateur, transmet le message vers ses destinataires.

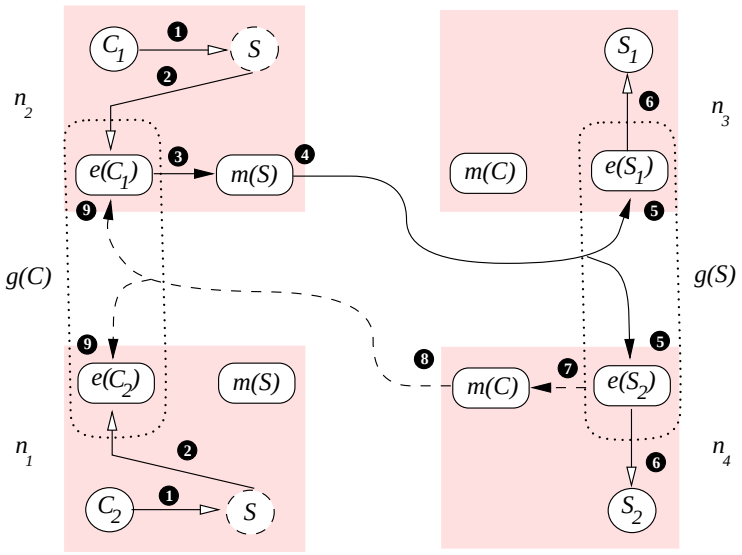

Figure 5.7: Principe du pré-filtrage

Au cours d'une invocation, le pré-filtrage peut être utilisé pour la transmission de la requête comme pour la transmission de la réponse. C'est le cas notamment de l'exemple présenté sur la figure 5.7. Le choix d'utiliser ou non le pré-filtrage dépend de la mise en œuvre de la stratégie de duplication.

Dans N2M, le pré-filtrage est réalisé par un algorithme intégré aux méthodes *outRequest*(), *inReply*() et *inRequest*(). Une méthode supplémentaire, appelée *coordinatorElected*() décrit le traitement à effectuer lorsqu'un nouveau coordinateur est élu.

Méthode *outRequest*()

La figure 5.8 présente le pseudo-code[7] de la méthode *outRequest*(). La méthode *outRequest*() de l'encapsulateur $e(O_i)$ est appelée lorsque la copie O_i invoque un objet serveur.

Comme il s'agit d'une nouvelle invocation, un nouvel identificateur unique est créé par instanciation de la classe $InvocationUID$ (ligne 2). Les identificateurs du serveur et de la méthode, relatifs à la nouvelle invocation, correspondent respectivement aux paramètres $invokedServer$ et $invokedMethod$ (ligne 1). Ces deux paramètres sont utilisés pour construire une instance de la classe $OutgoingEntry$ (ligne 3). Cette classe décrit les objets stockés dans $outgoingTable$, la table des invocations sortantes, i.e. les invocations pour lesquelles O est client.

[7] La syntaxe adoptée est proche de celle de C++.

```
method outRequest(invokedServer, invokedMethod)                          1
        newInvocationUID := InvocationUID :: new();                      2
        newEntry := OutgoingEntry :: new(invokedServer, invokedMethod,
        newInvocationUID);                                               3
        outgoingTable.add(newEntry);                                     4
        if e(O_i).isCoordinator                                          5
          then serverMailer := invokedServer.mailer;                     6
               serverMailer.sendRequest(newEntry.request);               7
        end if                                                           8
        (newEntry.semaphore).P                                           9
        return(newEntry.reply);                                         10
end method outRequest                                                   11
```

Figure 5.8: Pré-filtrage: méthode $outRequest()$

Chaque instance de la classe $OutgoingEntry$ stocke des informations à propos d'une invocation sortante. Ces informations sont:

- l'identificateur unique de l'invocation: la méthode $invocationUID()$ permet d'obtenir cet identificateur;
- l'état courant de l'invocation: cet état peut prendre les valeurs:
  - $DONE$: l'invocation est terminée;
  - $PENDING$: l'invocation est en cours;
  - $AWAITED$: l'invocation est attendue[8]

  Les méthodes $status()$ et $setStatus()$ permettent respectivement de consulter et de modifier cet état;
- le sémaphore de synchronisation (initialisé à 0) permettant d'attendre la réponse: chaque appel à $outRequest()$ se bloque sur ce sémaphore jusqu'à ce que la réponse à l'invocation parvienne à l'encapsulateur. La méthode $semaphore()$ permet d'accéder à ce sémaphore;
- la requête d'invocation: une requête d'invocation comprend l'identité du serveur invoqué ainsi que celle de la méthode. Les méthodes $request()$ et $setRequest()$ permettent respectivement de consulter et d'initialiser cette information;
- la réponse à l'invocation: une réponse correspond à l'objet que le serveur renvoie au client au terme de l'invocation. Les méthodes $reply()$ et $setReply()$ permettent respectivement de consulter et d'initialiser cette information;

La nouvelle instance de $OutgoingEntry$ est alors ajoutée à la table $outgoingTable$ (ligne 4): la nouvelle invocation est enregistrée au niveau de chaque encapsulateur. Si $e(O_i)$ est le coordinateur (ligne 5), la requête d'invocation est transmise au messager du serveur (lignes 6 et 7). Le messager du serveur est obtenu à partir du paramètre $invokedServer$. Le messager du serveur se charge de transmettre la requête aux encapsulateurs du serveur, en respectant la stratégie

[8] Ce dernier cas est expliqué plus loin.

de duplication du serveur. L'avant-dernière instruction de *outRequest*() (ligne 9) correspond à l'invocation de l'opération $P()$ qui permet le bloquage sur le sémaphore de synchronisation.

Méthode *inReply*()

La figure 5.9 présente le pseudo-code décrivant la méthode *inReply*(). Cette méthode est appelée lorsqu'un message transportant une réponse parvient à l'encapsulateur $e(O_i)$. Il s'agit alors de

```
method inReply(aReplyMessage)                                              1
    outgoingEntry := outgoingTable.search(aReplyMessage.invocationUID);    2
    if outgoingEntry <> NIL                                                3
      then if outgoingEntry.status = PENDING                               4
             then outgoingEntry.setReply(aReplyMessage.replyObject);       5
                  (outgoingEntry.semaphore).V;                             6
           end if                                                          7
      else newEntry := OutgoingEntry :: new2
           (aReplyMessage.invocationUID, aReplyMessage.replyObject);       8
           outgoingTable.add(newEntry);                                    9
           newEntry.setStatus := AWAITED                                  10
    end if                                                                11
end method inReply                                                        12
```

Figure 5.9: Pré-filtrage: méthode *inReply*()

déterminer si cette réponse a déjà été reçue par l'encapsulateur. Pour s'en assurer, on recherche l'entrée correspondante dans *outgoingTable*, la table des invocations sortantes (ligne 2).

Si cette entrée existe, on consulte l'etat courant de l'invocation (ligne 4). Si aucune réponse n'a encore été reçue pour cette invocation (état $PENDING$), il faut considérer la réponse que l'on vient de recevoir. Par conséquent, la réponse est stockée dans l'entrée correspondante de *outgoingTable* (ligne 5) et on libère le sémaphore de synchronisation (ligne 6). L'exécution de la méthode *outRequest*() reprend et la réponse est retournée au client (ligne 10 de la figure 5.8).

Si l'entrée n'existe pas, la réponse reçue correpondant à une requête *future* de O_i. En d'autres termes, $e(O_i)$ a reçu une réponse à une invocation dont il n'a pas *encore* enregistré la requête. Cette situation survient quand la copie O_i est beaucoup plus lente que le coordinateur: ce dernier a transmis une requête α, et obtenu une réponse alors que O_i n'a pas encore émis α. Par conséquent, une nouvelle entrée est créée dans *outgoingTable* avec l'état $AWAITED$ (i.e. invocation attendue) (ligne 8 à ligne 12). La méthode de création de l'instance de *OutgoingEntry* est différente (méthode *new*2()) de celle utilisée dans *outRequest*() puisque les paramètres sont différents.

Méthode *inRequest*()

La figure 5.10 présente le pseudo-code décrivant la méthode *inRequest*(). Cette méthode est appelée lorsqu'un message transportant une requête est reçu par l'encapsulateur $e(O_i)$.

Contrairement aux deux méthodes précédentes, la méthode *inRequest*() est utilisée lorsque O joue le rôle de serveur dans une invocation. Par conséquent, cette méthode s'appuie sur une

```
method inRequest(aRequestMessage)                                              1
        incomingEntry := incomingTable.search(aRequestMessage.invocationUId);  2
        if incomingEntry <> NIL                                                3
          then if e(O_i).isCoordinator                                         4
                 then if incomingEntry.status = DONE                           5
                        then clientMailer := (incomingEntry.client).mailer;    6
                             clientMailer.sendReply(incomingEntry.reply)       7
                      end if                                                   8
                 else if incomingEntry.status = AWAITED                        9
                        then newEntry.setStatus(DONE);                         10
                      end if                                                   11
               end if                                                          12
          else newEntry := incomingTable.add(aRequestMessage);                 13
               newEntry.setStatus(PENDING);                                    14
               replyObject := O_i.(aRequestMessage.op);                        15
               newEntry.setReply(replyObject);                                 16
               newEntry.setStatus(DONE);                                       17
               if e(O_i).isCoordinator                                         18
                 then clientMailer := (aRequestMessage.client).mailer;         19
                      clientMailer.sendReply(replyObject)                      20
               end if                                                          21
        end if                                                                 22
end method inRequest                                                           23
```

Figure 5.10: Pré-filtrage: méthode $inRequest()$

table des invocations entrantes, appelée *incomingTable*. Pour chaque invocation, cette table stocke les mêmes informations que *outgoingTable*, exception faite de la requête d'invocation et du sémaphore de synchronisation. Comme dans le cas de *inReply*(), on détermine tout d'abord si la requête contenue dans *aRequestMessage* n'a pas déjà été reçue (ligne 2).

S'il s'agit d'une nouvelle requête (lignes 13 à 23), une nouvelle entrée est créée dans la table *incomingTable*, avec l'état *PENDING*, i.e. invocation en cours (lignes 13 et 14). Puis, l'encapsulateur $e(O_i)$ invoque son objet application afin d'obtenir la réponse à l'invocation (ligne 15). La réponse obtenue est stockée dans l'entrée correspondante de *incomingTable* et l'état de l'invocation est mis à *DONE*. Puis, si $e(O_i)$ est le coordinateur, il transmet la réponse au messager du client (lignes 19 et 20).

Si la requête a déjà été reçue par $e(O_i)$ (lignes 3 à 12), le traitement diffère selon que $e(O_i)$ est coordinateur ou non. Si $e(O_i)$ est coordinateur et que l'invocation est terminée (i.e. une réponse est disponible), $e(O_i)$ transmet cette réponse via le messager du client (ligne 5 à 8). Si $e(O_i)$ n'est pas coordinateur et que la requête était attendue (état *AWAITED* décrit précédemment), alors l'état de l'invocation est mis à *DONE* (lignes 9 à 12).

```
method viewChange(aGroupView)                                              1
        if coordinator <> aGroupView.first                                 2
          then coordinator := aGroupView.first;                            3
               if e(Ci).isCoordinator                                      4
                 then e(Ci).coordinatorElected()                           5
               end if                                                      6
        end if                                                             7
end method viewChange                                                      8

method coordinatorElected()                                                9
        for outgoingEntry ∈ outgoingTable                                  10
            if outgoingEntry.status = PENDING                              11
              then serverMailer := (outgoingEntry.server).mailer;          12
                   serverMailer.sendRequest(outgoingEntry.request)         13
            end if                                                         14
        end for                                                            15
        for incomingEntry ∈ incomingTable                                  16
            if incomingEntry.status = DONE                                 17
              then clientMailer := (incomingEntry.client).mailer;          18
                   clientMailer.sendReply(incomingEntry.reply);            19
            end if                                                         20
        end for                                                            21
end method coordinatorElected                                              22
```

Figure 5.11: Pré-filtrage: méthodes *viewChange*() et *coordinatorElected*()

Élection du coordinateur

La figure 5.11 présente le pseudo-code décrivant la méthode *viewChange*() et la méthode *coordinatorElected*(). Ces méthodes réalisent les traitements relatifs à l'élection d'un nouveau

coordinateur.

Chaque encapsulateur mémorise l'identité du coordinateur dans une variable interne appelée *coordinator*. L'élection d'un nouveau coordinateur a lieu dès que la défaillance du coordinateur précédent est détectée. Cette élection se réduit à choisir le premier élement de la vue, une liste ordonnée d'identificateurs d'encapsulateurs.

La méthode *viewChange*() est appelée à chaque changement de vue. On détermine si le coordinateur a changé en comparant la tête de liste au contenu de la variable *coordinator* (ligne 2). Si c'est le cas, la variable *coordinator* est mise à jour (ligne 3). Puis, le nouveau coordinateur exécute la méthode *coordinatorElected*().

Cette méthode réalise les traitements que le nouveau coordinateur doit effectuer juste après avoir été élu. D'une part, le nouveau coordinateur réémet (lignes 10 à 15) toutes les requêtes correspondant aux invocations sortantes non encore terminées (état $PENDING$). Il est en effet possible que le coordinateur précédent ait défailli juste avant de transmettre une requête vers un serveur. D'autre part, le nouveau coordinateur réémet (lignes 16 à 22) les réponses correspondant aux invocations entrantes terminées (état $DONE$). Il en effet possible que le coordinateur précédent ait défailli juste avant de transmettre une réponse vers un client.

5.5.5 Post-filtrage

Le post-filtrage est plus simple à réaliser que le pré-filtrage. Lorsqu'un encapsulateur $e(O_i)$ reçoit un message μ (requête ou réponse), le message μ est délivré à O_i si et seulement si c'est la première fois que $e(O_i)$ reçoit μ. Ce principe permet de garantir la propriété d'intégrité, comme le montre la figure 5.12.

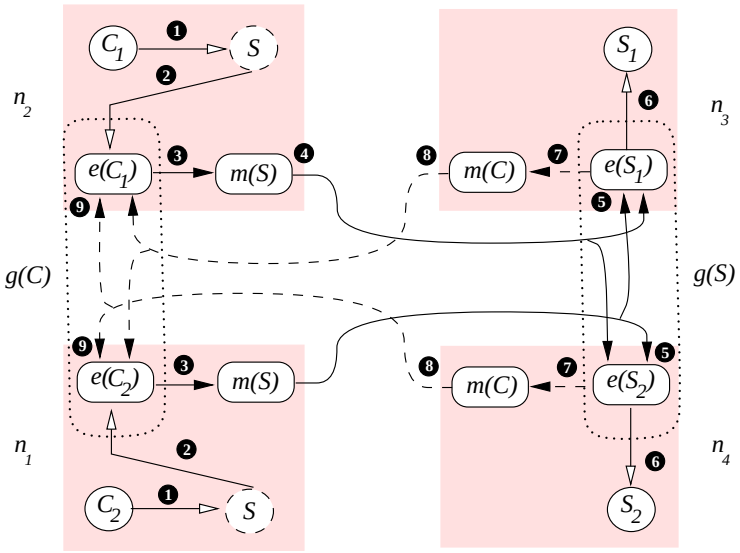

Figure 5.12: Principe du post-filtrage

Le contexte de cet exemple est similaire à celui de la figure 5.6: un objet dupliqué activement C invoque un objet dupliqué activement S. Comme le post-filtrage est utilisé, *chaque* encapsu-

lateur $e(C_i)$ transmet (étiquette 4) une requête au messager $m(S)$ résidant sur le même nœud. Par conséquent, chaque encapsulateur $e(S_j)$ reçoit (étiquette 5) deux exemplaires de la même requête, mais il n'invoque (étiquette 6) sa copie S_j qu'une seule fois. Le post-filtrage appliqué à la transmission de la réponse se déroule de manière analogue.

Les traitements relatifs au post-filtrage sont réalisés par les méthodes *inRequest*(), *inReply*() et *outRequest*(). La méthode *inReply*() est identique à celle utilisée pour le pré-filtrage.

Méthode *outRequest*()

La figure 5.13 présente le pseudo-code décrivant la méthode *outRequest*().

```
method outRequest(invokedServer, invokedMethod)                          1
        newInvocationUID := InvocationUID :: new();                      2
        newEntry := OutgoingEntry :: new(invokedServer, invokedMethod,
        newInvocationUID);                                               3
        outgoingTable.add(newEntry);                                     4
        serverMailer := invokedServer.mailer;                            5
        serverMailer.sendRequest(newEntry.request);                      6
        (newEntry.semaphore).P                                           7
        return(newEntry.reply);                                          8
end method outRequest                                                    9
```

Figure 5.13: Post-filtrage: méthode *outRequest*()

On retrouve les mêmes traitements que pour le pré-filtrage (cf. figure 5.7) avec la distinction du cas du coordinateur en moins. En effet, cette fois tous les encapsulateurs transmettent la requête au messager du serveur.

Méthode *inRequest*()

La figure 5.13 présente le pseudo-code décrivant la méthode *inRequest*().

Dans ce cas aussi, les traitements sont identiques au pré-filtrage (cf. figure 5.10), avec les deux alternatives distinguant le cas du coordinateur en moins.

5.5.6 Pré-filtrage vs post-filtrage

Le pré-filtrage correspond à une approche préventive (on évite les messages redondants) tandis que le post-filtrage correspond à une approche curative (on autorise les messages redondants tout en éliminant leurs effets indésirables). Il paraît donc clair qu'en termes de ressources informatiques (bande passante du réseau notamment), le pré-filtrage est plus économique. Toutefois, cette affirmation est à nuancer lorsque le coordinateur défaille.

Dans ce cas, le nouveau coordinateur retransmet (méthode *coordinatorElected*()) certains messages. Un sous-ensemble de ces messages a déjà été transmis à leurs destinataires par l'ancien coordinateur. Non seulement ces retransmissions sont inutiles, mais en outre, elles supposent

```
method inRequest(aRequestMessage)                                                    1
    incomingEntry := incomingTable.search(aRequestMessage.invocationUId);            2
    if incomingEntry <> NIL                                                          3
      then if incomingEntry.status = DONE                                            4
             then clientMailer := (incomingEntry.client).mailer;                     5
                  clientMailer.sendReply(incomingEntry.reply)                        6
             else if incomingEntry.status = AWAITED                                  7
                    then newEntry.setStatus(DONE);                                   8
                  end if                                                             9
           end if                                                                    10
      else newEntry := incomingTable.add(aRequestMessage);                           11
           newEntry.setStatus(PENDING);                                              12
           replyObject := Oi.(aRequestMessage.op);                                   13
           newEntry.setReply(replyObject);                                           14
           newEntry.setStatus(DONE);                                                 15
           clientMailer := (aRequestMessage.client).mailer;                          16
           clientMailer.sendReply(replyObject)                                       17
    end if                                                                           18
end method inRequest                                                                 19
```

Figure 5.14: Post-filtrage: méthode $inRequest()$

que les destinataires post-filtrent les messages redondants. En d'autres termes, lorsque le coordinateur défaille, le pré-filtrage s'appuie sur le post-filtrage pour permettre au système de poursuivre son exécution.

Le post-filtrage est moins vulnérable aux défaillances que le pré-filtrage. Comme tous les encapsulateurs d'un groupe ont le même rôle, aucun traitement particulier n'est nécessaire en cas de défaillance de l'un d'entre eux. Le coût reste le même, qu'il y ait ou pas de défaillance.

En conclusion, le pré-filtrage peut être vu comme une sorte d'optimisation du post-filtrage, tant que le coordinateur ne défaille pas. Le prix de cette optimisation est payé au moment de l'initialisation du nouveau coordinateur. Par conséquent, les stratégies de duplication pour lesquelles le pré-filtrage s'appliquent (i.e. duplication active et duplication semi-active) peuvent être mise en œuvre selon deux variantes: une variante standard utilisant le post-filtrage, et une version optimisée utilisant le pré-filtrage.

5.5.7 Synthèse

La solution apportée par le service N2M au problème de l'encapsulation des stratégies de duplication repose sur trois principes:

- l'utilisation par les messagers de multicasts vue-synchrones, uniformes et ordonnés pour transmettre les messages (i.e. requêtes et réponses);

- les messagers symétriques permettent de cacher complètement le choix du critère d'ordre qui varie selon la stratégie de duplication du destinataire du message;

- les encapsulateurs mettent en œuvre le pré-filtrage et le post-filtrage pour garantir la propriété d'intégrité lors de la transmission des messages;

5.6 Application aux stratégies de duplication

Le service N2M utilise les solutions présentées dans ce chapitre pour mettre en œuvre les quatre stratégies de duplication étudiées dans cette thèse. Chaque stratégie de duplication est réalisée à l'aide d'une classe d'encapsulateurs et d'une classe de messagers.

La classe d'encapsulateurs réalise la gestion du groupe de copies ainsi que le filtrage des invocations. La classe de messagers met en œuvre la transmission des messages (requêtes et réponses) vers le groupe d'encapsulateurs. Chacune des quatre sections suivantes présente brièvement les classes d'objets communication mettant en œuvre une stratégie de duplication.

5.6.1 Duplication active

La duplication active est réalisée dans le service N2M selon deux variantes: l'une utilisant le post-filtrage, l'autre utilisant le pré-filtrage. La première variante correspond à la classe d'encapsulateurs *AREncaps* tandis que la seconde correspond à la classe d'encapsulateurs *PreFAREncaps*. Ces deux variantes utilisent la classe de messagers *ARMailer*. Cette dernière utilise un multicast garantissant l'ordre total pour transmettre les messages (requêtes et réponses) au groupe d'encapsulateurs.

5.6.2 Duplication passive

La duplication passive est réalisée dans le service N2M à l'aide de la classe d'encapsulateurs *PREncaps* et de la classe de messagers *PRMailer*.

La classe *PREncaps* utilise le post-filtrage pour garantir la propriété d'intégrité lorsqu'une nouvelle copie primaire est élue (cf. section 4.5.2). Le pré-filtrage n'est pas applicable à la duplication passive puisque la copie primaire est la seule à transmettre les messages. L'élection de la copie primaire est basée sur la vue du groupe d'encapsulateurs. Comme la vue est une liste ordonnée d'identificateurs globaux d'encapsulateurs, une solution simple consiste à choisir, comme nouvelle copie primaire, l'encapsulateur correspondant au premier élément de cette liste. Cette technique est utilisée également pour la duplication semi-active et la duplication coordinateur-cohorte, toutes deux étant également des stratégies de duplication basées sur une copie primaire.

La classe *PRMailer* n'utilise pas de multicast pour transmettre les messages puisque ceux-ci ne sont destinés qu'à la copie primaire. Par contre, elle utilise un multicast pour déterminer l'identité de la copie primaire. En effet, les instances de *PRMailer* sont des messagers: ils n'appartiennent pas au groupe d'encapsulateurs et ils ne sont pas informés des changements de vue du groupe d'encapsulateurs.

5.6.3 Duplication semi-active

Comme la duplication active, la duplication semi-active est réalisée dans le service N2M selon deux variantes: l'une utilisant le post-filtrage, l'autre utilisant le pré-filtrage. La première

variante correspond à la classe d'encapsulateurs *SAREncaps* tandis que la seconde correspond à la classe d'encapsulateurs *PreFSAREncaps*. Ces deux variantes utilisent la classe de messagers *SARMailer*. Cette dernière utilise un multicast garantissant l'ordre FIFO pour transmettre les messages (requêtes et réponses) au groupe d'encapsulateurs.

5.6.4 Duplication coordinateur-cohorte

La duplication coordinateur-cohorte est réalisée dans le service N2M à l'aide de la classe d'encapsulateurs *CCREncaps* et de la classe de messagers *CCRMailer*. Comme pour la duplication passive, la classe *CCREncaps* utilise le post-filtrage pour garantir la propriété d'intégrité lorsqu'une nouvelle copie primaire est élue (cf. section 4.5.4). Comme pour la duplication semi-active, la classe *CCRMailer* utilise un multicast garantissant l'ordre FIFO pour transmettre les messages (requêtes et réponses) au groupe d'encapsulateurs.

5.7 Conclusion

Résumé

Le service N2M est un service d'invocation pour objets dupliqués. Les objets qui utilisent N2M sont appelés *objets application*. Les objets qui réalisent N2M sont appelés *objets communication*. Les objets communication modélisent l'abstraction de *représentants symétriques* afin d'encapsuler la duplication des objets application. Lorsqu'un objet application dupliqué C invoque un objet application dupliqué S, le service N2M construit un représentant local du serveur S sur chaque nœud où réside une copie du client C, et un représentant local de C sur chaque nœud où réside une copie de S. Les copies du client et celles du serveur ne perçoivent pas la duplication qui est gérée entièrement par les objets communication.

Un objet application dupliqué est construit à partir d'un groupe d'encapsulateurs. Un encapsulateur est un objet communication associé à une copie d'un objet application. Son rôle est de filtrer les messages (requêtes et réponses) entrant ou sortant de la copie auquel il est associé. Le filtrage des messages est destiné à garantir la propriété d'intégrité. Il peut être réalisé selon deux approches. Le pré-filtrage consiste à filtrer les messages au niveau de l'émetteur. Le post-filtrage consiste à filtrer les messages au niveau du destinataire.

Pour communiquer avec un objet application dupliqué O, un objet application doit s'adresser, via son encapsulateur, au messager local de O. Un messager est un objet communication chargé de représenter un objet application dupliqué sur un nœud. Son rôle est de cacher, aux interlocuteurs de O, la composition du groupe des copies de O, ainsi que la propriété d'ordre imposée par la stratégie de duplication de O.

Tous ces mécanismes ont été utilisés pour définir des classes d'encapsulateurs et des classes de messagers permettant de réaliser les quatre stratégies de duplication étudiées dans cette thèse.

Commentaires

Ce chapitre a présenté comment le service N2M proposait de résoudre le problème de l'encapsulation de la duplication. Le chapitre suivant présente une mise en œuvre de N2M basée sur ISIS.

Chapitre 6

Mise en œuvre du service N2M dans l'environnement GARF-v2

6.1 Introduction

Ce chapitre décrit la mise en œuvre du service N2M dans GARF-v2, un environnement logiciel visant à faciliter la tâche du programmeur d'applications réparties et tolérantes aux fautes.

Présentation générale

Le projet de *Génération Automatique d'applications Résistantes aux Fautes*[1] (GARF) consiste à réaliser un environnement logiciel destiné à faciliter la programmation d'applications réparties et tolérantes aux fautes. La tolérance aux fautes des applications développées avec GARF est réalisée exclusivement par logiciel, en se basant sur les dispositifs matériels qui composent un système réparti (i.e. ordinateurs interconnectés par un réseau).

Les travaux entrepris dans le contexte du projet GARF ont conduit à la définition d'un modèle de programmation et à la mise en œuvre de l'environnement GARF-v1. À ce point du projet, une seule stratégie de duplication (la duplication active) avait été étudiée et réalisée. En outre, la duplication ne s'appliquait qu'à des s-composants (des objets uniquement serveurs, cf. section 2.4).

Le développement de l'environnement GARF-v2, dans le cadre de cette thèse, a permis de lever ces restrictions. L'environnement GARF-v2 met en œuvre le service N2M ainsi que d'autres mécanismes indispensables (par ex. changement de vue, transfert d'état) qui ne figuraient pas dans GARF-v1. Le développement de GARF-v2 a bénéficié de l'expérience acquise avec GARF-v1. De nombreuses classes d'objets développées pour GARF-v1 ont été réutilisées dans la mise en œuvre de GARF-v1.

Organisation

La suite du chapitre est organisée comme suit. La section 6.2 décrit brièvement le projet GARF. La section 6.3 présente l'architecture de l'environnement GARF-v2. Chacune des quatre sections suivantes décrivent une couche de GARF-v2. La section 6.8 conclut ce chapitre.

[1] Résistance aux fautes est synonyme de tolérance aux fautes.

6.2 Présentation du projet GARF

La section 6.2.1 résume les objectifs du projet, tandis qu'un aperçu de la démarche suivie et des résultats du projet est donné à la section 6.2.2.

6.2.1 Objectifs

L'objectif du projet GARF était la génération automatique d'applications tolérantes aux fautes. L'idée consistait à générer une application A_{rf} résistante aux fautes, à partir d'une application A non résistante aux fautes, conçue selon le modèle client-serveur. L'application A_{rf} devait être réalisée par duplication des composants logiciels de A. Générer A_{rf} à partir de A consistait à utiliser des mécanismes de groupe et de multicast vue-synchrone (comme ceux que fournit ISIS [Birman 93]) pour réaliser la duplication des composants de A.

L'idée de la génération automatique était motivée par une double constatation. D'une part, la réalisation d'une application résistante aux fautes est une tâche complexe demandant des compétences pointues. D'autre part, comme les algorithmes réalisant la tolérance aux fautes sont à la fois complexes et indépendants de la sémantique des applications, il est parfaitement judicieux de vouloir réutiliser les composants logiciels réalisant ces algorithmes.

En visant la génération automatique, le projet GARF cherchait à simplifier la tâche du programmeur en appliquant un principe fondamental du génie logiciel (la réutilisabilité du logiciel) à la réalisation d'applications tolérantes aux fautes.

6.2.2 Démarche et résultats

Le projet GARF a consisté en la réalisation de quatre tâches principales:

1. la conception d'un modèle de programmation de l'application A qui soit adapté à la génération automatique;

2. la conception des composants logiciels génériques réalisant la tolérance aux fautes et servant de supports à la génération automatique;

3. l'identification des étapes de la génération automatique de A_{rf} à partir de A;

4. la mise en œuvre d'un prototype et d'une application pilote destinée à expérimenter ce prototype.

Modèle de programmation de l'application A

Le modèle de programmation de l'application est basé sur des objets. En effet, le concept d'objet exprime de façon naturelle le modèle client-serveur. L'application A est programmée comme une application centralisée: le programmeur se concentre sur les fonctionnalités de l'application A et ne se préoccupe ni de la répartition ni de la tolérance aux fautes. Les objets utilisés pour la programmation de A sont en fait les objets application décrits à la section 5.2.1.

Composants logiciels génériques

Les composants logiciels génériques sont décrits par des classes d'objets communication[2]. Les objet communication sont chargés de gérer la répartition et la duplication des objets données.

Le modèle du service N2M (cf. section 5.2.1) est une évolution du modèle initial de GARF. Le modèle initial de GARF ne comportait pas de messagers symétriques. La notion de messager était asymétrique et tout à fait comparable à la notion de mandataire. L'étude des communications entre objets dupliqués avec des stratégies différentes, menée dans le contexte de cette thèse, a conduit à la définition du modèle de communication de N2M, basé sur des messagers symétriques.

Étapes de la génération automatique

Le choix d'un modèle objet pour exprimer à la fois l'application A et les composants génériques simplifie considérablement la génération automatique. L'application A est rendue tolérante aux fautes en dupliquant les objets application critiques.

Dupliquer un objet application O consiste à lui associer un encapsulateur et une classe de messagers qui implémentent la duplication. L'encapsulateur de O est créé à l'exécution au moment de l'instanciation de O. Un messager est instancié sur chaque nœud ou réside un objet communiquant avec O. Par conséquent, avant le début de l'exécution de l'application, le programmeur doit associer une classe d'encapsulateurs et une classe de messagers à chaque classe d'objets application critiques.

En résumé, la génération automatique de A_{rf} à partir de A consiste en les étapes suivantes:

1. programmation des classes d'objets application de l'application A;
2. Pour chaque classe d'objets application critiques:
   (a) choix d'une classe d'encapsulateurs et d'une classe de messagers réalisant la tolérance aux fautes souhaitée;
   (b) association de ces deux classes à la classe d'objet application (pour plus de détails, se référer à la section 6.4);

À ce point, l'application A_{rf} peut être exécutée.

Mise en œuvre d'un prototype

Les environnements GARF-v1 et GARF-v2 ont tous deux été programmés en Smalltalk et en C. Les objets application et les objets communication sont programmés en Smalltalk. Les objets communication utilisent les services (i.e. gestion de groupes, multicasts) de la plate-forme ISIS à travers une interface externe écrite en C (cf. section 6.7).

Les deux environnements ont été utilisés pour développer une application pilote: le Gestionnaire Réparti d'Agendas Dupliqués (GRAD) [Mazouni 94]. Malgré une sémantique relativement sim-

[2]La terminologie "objets application/objets communication" remplace la terminologie "objet données/objets comportementaux", adoptée initialement dans GARF. Cette nouvelle terminologie permet de distinguer le modèle de communication de N2M, du modèle de communication initial de GARF.

ple, le GRAD a permis d'identifier et de résoudre de nombreux problèmes liés à la répartition et à la tolérance aux fautes.

6.3 Architecture de l'environnement GARF-v2

L'architecture de l'environnement GARF-v2 est représentée sur la figure 6.1. Les "boîtes" grises correspondent aux quatre couches de GARF-v2: les couches Application, Runtime, Communication et Réseau. Les deux autres "boites"correspondent aux logiciels sur lesquels GARF-v2 s'appuie: ISIS (cf. section 2.6.1) et le système d'exploitation UNIX[3] augmenté du protocole de communication TCP/IP.

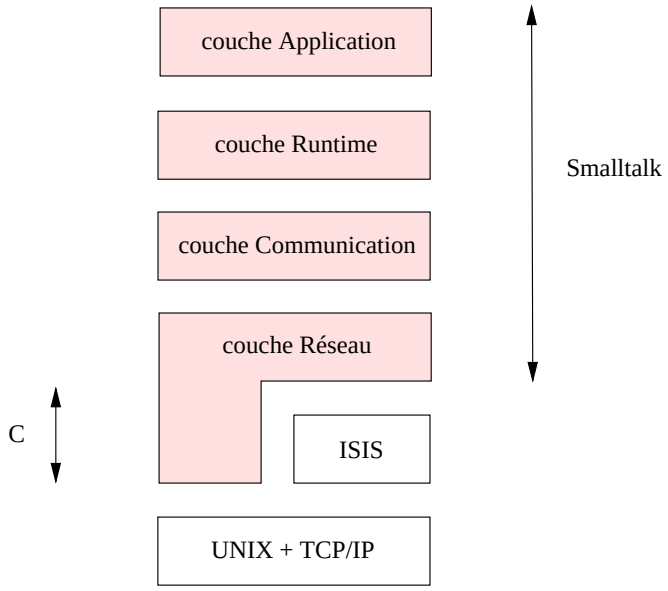

Figure 6.1: Architecture de GARF-v2

Chaque couche est réalisée à l'aide d'un ensemble de classes d'objets programmées en Smalltalk. À ce titre, GARF-v2 est une extension de l'environnement Smalltalk[4] permettant de programmer des applications réparties et tolérantes aux fautes. Il est à noter que le langage Smalltalk n'a *pas* été modifié. La couche Réseau comprend en outre une interface à ISIS, programmée en langage C.

Chaque couche joue un rôle spécifique dans la mise en œuvre du service N2M. La couche Application est la couche d'interface: elle fournit la classe abstraite[5] dont tout objet application doit hériter. La couche Runtime réalise le mécanisme d'interception des invocations. Elle assure ainsi la liaison entre les objets application et les objets communication. Ces derniers sont mis en œuvre par les classes de la couche Communication. Enfin, la couche Réseau comprend des

[3] Il s'agit du système Sun OS 5.x, conforme à la norme System V Release 4.

[4] Il s'agit du produit VisualWorks 2.0 développé par ParcPlace Systems, Inc.

[5] Une **classe abstraite** (*abstract class*) est une classe dont on ne crée pas d'instances. Une classe abstraite sert uniquement à définir des variables et des méthodes qui sont héritées par ses sous-classes.

classes, utilisées par les objets communication, qui réalisent l'infrastructure de communication (interface avec ISIS principalement).

6.4 Couche Application

La couche Application met en œuvre la notion d'objet application en définissant la classe abstraite *ApplicationObject*. Pour bénéficier des avantages de la génération automatique et pour pouvoir utiliser le service N2M, toute classe d'objets d'application doit être programmée (étape 1 de la génération automatique (cf. section 6.2.2)), comme une sous-classe de *ApplicationObject*. Outre cette contrainte, la programmation d'une classe d'objets application ne diffère pas de la programmation Smalltalk usuelle.

L'étape 2 de la génération automatique consiste à surcharger la méthode de classe **garfNew:**, définie par *ApplicationObject*, pour chaque classe d'objets application dont on veut dupliquer les instances. Pour une classe d'objets application donnée, la méthode **garfNew:** définit:

- le nombre et la répartition des copies de chaque instance;
- la classe des encapsulateurs des copies de chaque instance;
- la classe du messager de chaque instance.

Cette méthode est appelée automatiquement lorsque la couche Runtime intercepte une invocation visant à créer un objet application dupliqué. En Smalltalk, on instancie une classe en invoquant une méthode de classe spécifique (par ex. **new**). L'invocation interceptée par la couche Runtime est réifiée puis passée en paramètre à la méthode **garfNew:**.

La figure 6.2 donne un exemple[6] de méthode **garfNew:**. Le traitement peut être décomposé en quatre étapes:

1. création du groupe d'encapsulateurs (lignes 3 à 7):
   L'invocation de la méthode de classe **replicasOn:uidKey:** (ligne 7) permet de créer un groupe dont les membres sont des instances de *AREncaps*, la classe d'encapsulateurs réalisant la duplication active. La réponse à cette invocation est un identificateur unique de groupe qui est stocké dans la variable *newGroupUid*. La méthode **replicasOn:uidKey:** est définie par la classe *Encaps*, la super-classe de *AREncaps* (cf. section 6.6.1). Le paramètre $listOfNodes$ désigne la liste des nœuds sur lesquels les membres du groupe vont être créés. Le paramètre $groupUidKey$ est une chaîne de caractères servant à construire un identificateur unique de groupe.

   La valeur de $groupUidKey$ est construite par la méthode de classe **getKeyFrom:** (ligne 6) à partir du paramètre $aCreationInvocation$. La méthode de classe **getKeyFrom:**, définie par *ApplicationObject*, doit être surchargée par chaque classe d'objets application. Elle utilise les paramètres de l'invocation de création pour générer une chaîne de caractères unique à chaque instanciation.

2. création du messager (ligne 8):
   L'invocation de la méthode de classe **to:** permet de créer une instance de *ARMailer*. Cette

[6] Pour faciliter la compréhension du code Smalltalk, les noms de méthodes sont en gras (par ex. **garfNew:**) et les noms de classes commencent par une majuscule (par ex. *List*).

```
garfNew: aCreationMessage                                              1
        | listOfNodes newMailer groupUidKey newGroupUid |              2

        listOfNodes := List new .                                      3
        listOfNodes add: 'lsesun2.epfl.ch' .                           4
        listOfNodes add: 'lsesun3.epfl.ch' .                           5
        groupUidKey := self getKeyFrom: aCreationInvocation .          6
        newGroupUid := AREncaps replicasOn: listOfNodes
                                uidKey: groupUidKey .                  7

        newMailer := ARMailer to: newGroupUid .                        8

        AREncaps buildAndBind: self
                 sending: aCreationInvocation
                 to: newMailer .                                       9

        ↑ ObjectReference of: newMailer .                              10
```

Figure 6.2: Un exemple de méthode garfNew:

classe met en œuvre un messager pour la duplication active. L'identificateur du groupe d'encapsulateurs est passé en paramètre au messager créé.

3. création des copies (ligne 9):
   L'invocation de la methode de classe **buildAndBind:sending:to:** indique à chaque membre du groupe d'encapsulateurs qu'il doit créer une copie de l'objet application et contrôler les invocations de cette copie. Cette méthode est définie par la classe *Encaps*.

   Le premier paramètre précise la classe de la copie à créer. Il s'agit de la classe sur laquelle la méthode **garfNew:** est invoquée. Par conséquent, la pseudo-variable *self* désigne cette classe[7]. Le second paramètre est l'invocation de création initiale, interceptée par la couche Runtime. Chaque encapsulateur va donc adresser l'invocation contenue dans *aCreationInvocation* à la classe désignée par *self*. Le troisième paramètre désigne le messager à utiliser pour délivrer ce message d'initialisation du groupe d'encapsulateurs.

4. création de la référence sur le messager (ligne 10):
   L'invocation de la méthode de classe **of:** permet de créer une instance de la classe *ObjectReference* et d'initialiser cette référence avec le paramètre.

   La classe *ObjectReference* met en œuvre la notion de référence décrite au chapitre 5. Cette référence est utilisé comme résultat de la méthode **garfNew:**. Par conséquent, le créateur de l'objet application créé suite à l'appel de **garfNew:**, obtient cette référence qu'il considère comme étant l'objet qu'il vient de créer.

La figure 6.3 permet de fixer les idées. Sur le nœud *lsesun1.epfl.ch*, une copie C_1 d'un client C, a instancié la classe d'un serveur S. Cette opération d'instanciation a provoqué l'exécution de la méthode **garfNew:**, associée à la classe de S. Au terme de cette exécution:

[7] Dans le code d'une méthode m, la pseudo-variable *self* désigne l'objet auquel on a demandé l'invocation de m.

- le groupe d'encapsulateurs $g(S)$ et les copies S_1 et S_2, ont été créés sur les nœuds *lsesun2.epfl.ch* et *lsesun3.epfl.ch*;
- le messager $m(S)$ a été créé sur le nœud *lsesun1.epfl.ch*.

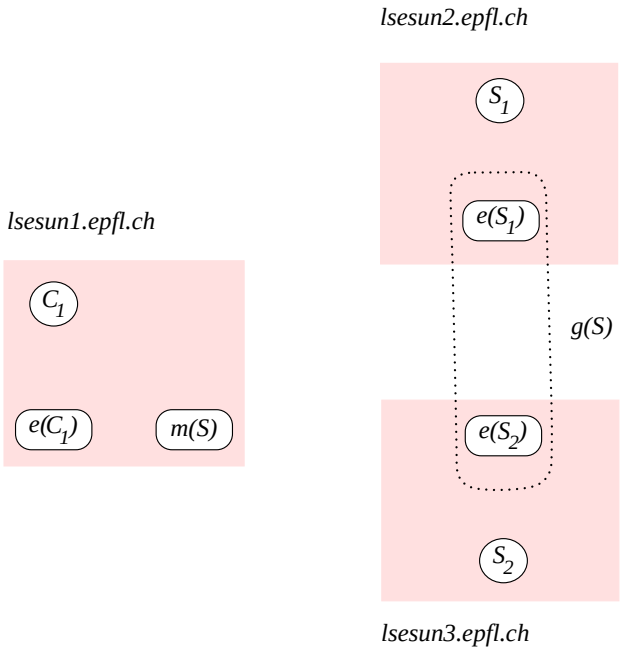

Figure 6.3: Un exemple illustrant le rôle de **garfNew:**

6.5 Couche Runtime

La couche Runtime réalise le mécanisme d'interception des invocations. Elle intervient à chaque création d'un objet application et à chaque invocation impliquant un objet application. Toute invocation visant à l'instanciation d'une classe d'objets application est transformée en une invocation de la méthode **garfNew:** de cette classe. De façon analogue, toute invocation d'un objet application est transformée en une invocation de la méthode **outRequest:** de l'encapsulateur associé à l'objet application invoqué.

L'idée consiste à substituer chaque objet application et chaque classe d'objets application par une référence. Ainsi, toute invocation destinée à une classe ou à un objet parvient en fait à sa référence qui intercepte cette invocation. Le mécanisme d'interception des invocations s'appuie sur la gestion des exceptions dans le langage Smalltalk. Cette méthode est communément utilisée [Pascoe 86, Foote 89] pour réaliser des extensions à Smalltalk. En Smalltalk, lorsqu'un objet est invoqué avec une méthode qu'il ne définit pas, la méthode **doesNotUnderstand:** est appelée avec, comme paramètre, l'invocation illicite réifiée. La méthode **doesNotUnderstand:** est définie par la classe *Object*, la racine de l'arborescence d'héritage en Smalltalk. Son traitement consiste à lancer un outil de mise au point (*debugger*).

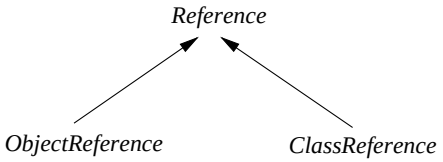

Figure 6.4: La classe $Reference$ et ses sous-classes

L'interception des invocations peut être réalisée en redéfinissant la méthode **doesNotUnderstand:** pour les références, et en faisant en sorte que ce soit la seule méthode qu'elles comprennent. Ainsi, toute invocation d'une référence provoquera une exception traitée par la méthode **doesNotUnderstand:** redéfinie. La couche Runtime met en place ce mécanisme en définissant la classe abstraite $Reference$. Cette classe a la particularité de ne pas avoir de super-classe: on assure ainsi qu'elle n'hérite aucune méthode. La classe $Reference$ a deux sous-classes (cf. figure 6.4) $ObjectReference$ et $ClassReference$ modélisant respectivement le comportement d'une référence sur un objet application et le comportement d'une référence sur une classe d'objets application.

La référence sur un objet application est créée par la méthode **garfNew:** au moment de la création de l'objet. La méthode **doesNotUnderstand:** est programmée pour invoquer la méthode **outRequest:** de l'encapsulateur associé à l'objet auquel est destiné l'invocation interceptée.

Les références sur les classes d'objets application sont créées explicitement par le programmeur dès qu'il souhaite tester son application. Cette création est réalisée par un utilitaire qui substitue chaque classe par sa référence dans le dictionnaire $SmalltalkDictionary$. Ce dernier est un objet défini par l'environnement Smalltalk pour associer les noms de classes aux objets représentant les classes[8]. Dès que la substitution est effective, tout invocation destinée à une classe d'objets application parvient en fait à sa référence. La méthode **doesNotUnderstand:** de la classe $ClassReference$ est programmée de façon à invoquer la méthode **garfNew:** si l'invocation correspond à une création d'objets. Dans le cas contraire, l'invocation initiale est effectuée normalement sur la classe d'objets application.

La figure 6.5 illustre le rôle de la couche Runtime au cours d'une invocation. Elle permet notamment de mieux comprendre le mécanisme d'interception des invocations. Sur cet exemple, un client dupliqué C invoque un serveur dupliqué S, mais, pour simplifier, seuls deux nœuds sont représentés: celui de la copie C_1 et celui de la copie S_1. La copie C_1 invoque la référence de S, instance de la classe $ObjectReference$, représentée par un cercle en trait discontinu portant l'etiquette S.

Cette invocation déclenche une exception qui se traduit par l'appel de la méthode **doesNotUnderstand:** avec l'argument *anInvocationRequest* qui est un objet représentant l'invocation interceptée. Comme il a été dit précédemment, la méthode **doesNotUnderstand:** invoque l'encapsulateur de C_1 avec la méthode **outRequest:**. L'invocation est alors traitée par la couche Communication.

[8] En Smalltalk, les classes sont aussi des objets.

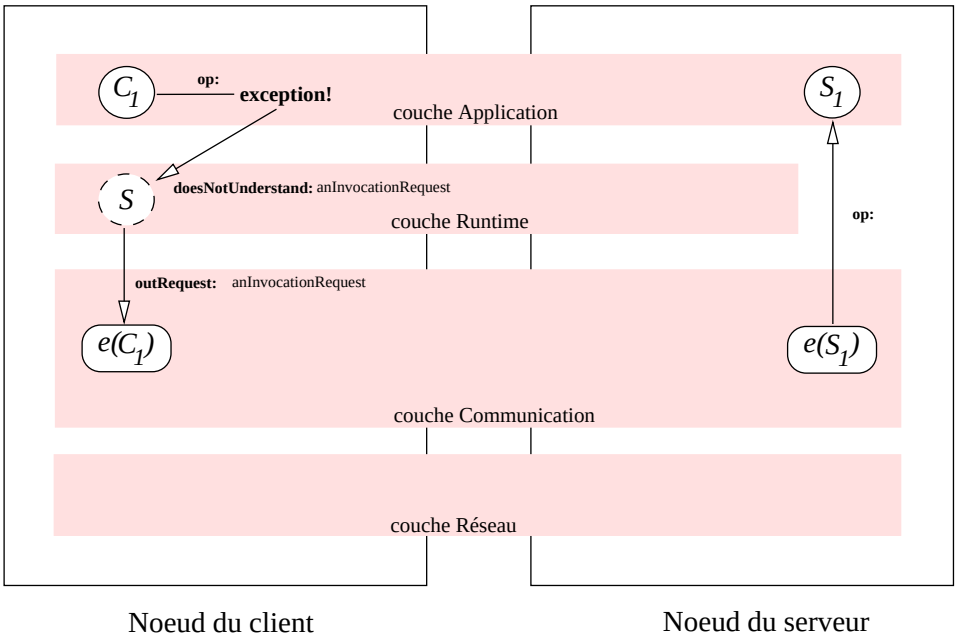

Figure 6.5: Rôle de la couche Runtime au cours d'une invocation

6.6 Couche Communication

La couche Communication constitue le cœur de la mise en œuvre du service N2M dans GARF-v2. Elle regroupe les classes chargées de la mise en œuvre des objets communication (cf. section 6.6.1) ainsi que les classes réalisant la gestion des invocations (cf. section 6.6.2).

6.6.1 Mise en œuvre des objets communication

La mise en œuvre des objets communication repose sur les classes *ComObjectUid*, *ComObject*, *Encaps* et *Mailer*. Les liens d'héritage entre ces classes sont présentés sur la figure 6.6. Les classes *ComObject* et *ComObjectUid* sont des sous-classes de la classe *Object*, racine de l'arbre d'héritage en Smalltalk. Les classes *Encaps* et *Mailer* sont des sous-classes de la classe *ComObject*. Les sous-classes de *Encaps* et de *Mailer* réalisent les stratégies de duplication. Elles ont été présentées à la section 5.6.

On notera que *PreFAREncaps*, la classe d'encapsulateurs qui met en œuvre le pré-filtrage dans le contexte de la duplication active, est une sous-classe de *AREncaps*, la classe d'encapsulateurs qui met en œuvre le post-filtrage dans le contexte de la duplication active. Il en est de même pour les classes *PreFSAREncaps* et *SAREncaps* dans le contexte de la duplication semi-active.

La classe *ComObjectUid* réalise la notion d'identificateur global (cf. section 5.4.3). Un identificateur global est constitué par un identificateur de groupe, un identificateur de nœud, et par une indication de la catégorie de l'objet communication (i.e. encapsulateur ou messager). Chacune de ces composantes est représentée par une variable d'instance. En outre, la classe *ComObjectUid* définit des méthodes réalisant des traitements usuels: création, consultation et

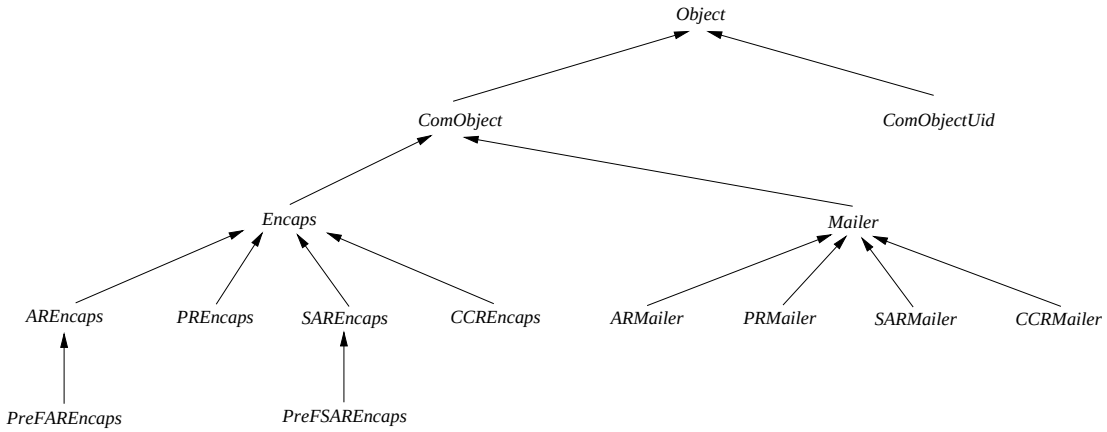

Figure 6.6: Les classes chargées de la mise en œuvre des objets communication

comparaison.

La classe abstraite *ComObject* réalise la notion d'objet communication (i.e. encapsulateur et messager). Les variables et les méthodes significatives de la classe *ComObject* sont présentées sur la figure 6.7:

- La variable $objectUid$ sert à stocker l'identificateur global d'un objet communication. Cet identificateur est une instance de la classe $ComObjectUid$.
- La variable *dispatcher* stocke une référence sur le distributeur associé à chaque objet communication. Ce distributeur est une instance de la classe *SQDispatcher* décrite à la section 6.7.1.
- La variable $toolkitAdapter$ désigne l'adaptateur, l'unique instance de la classe $IsisAdapter$ présente sur chaque nœud (cf. section 6.7.2).
- La méthode abstraite[9] **deliver:** est appelée par le distributeur (variable *dispatcher*) à chaque fois qu'un message doit être délivré à l'objet communication.

La classe abstraite *Encaps* réalise la notion d'encapsulateur. Les variables et les méthodes significatives de la classe *Encaps* sont présentées sur la figure 6.7:

- La variable *incomingTable* (resp. *outgoingTable*) correspond à la table des invocations entrantes (resp. sortantes) (cf. section 5.5.4). Chaque élément de cette table est une instance de la classe $IncomingEntry$ (resp. $OutgoingEntry$) (cf. section 6.6.2). La variable $invocationCounter$ correspond au compteur permettant de générer le numéro d'invocation (cf. section 5.5.2).
- La variable $groupView$ sert à stocker la vue courante du groupe d'encapsulateurs. La vue est définie comme une liste d'identificateurs de nœuds, puisque deux encapsulateurs du même groupe sont nécessairement sur des nœuds différents.

[9] Une méthode **abstraite** (*abstract method*) est une méthode "vide" dont la responsabilité de la mise en œuvre est laissée à chaque sous-classe.

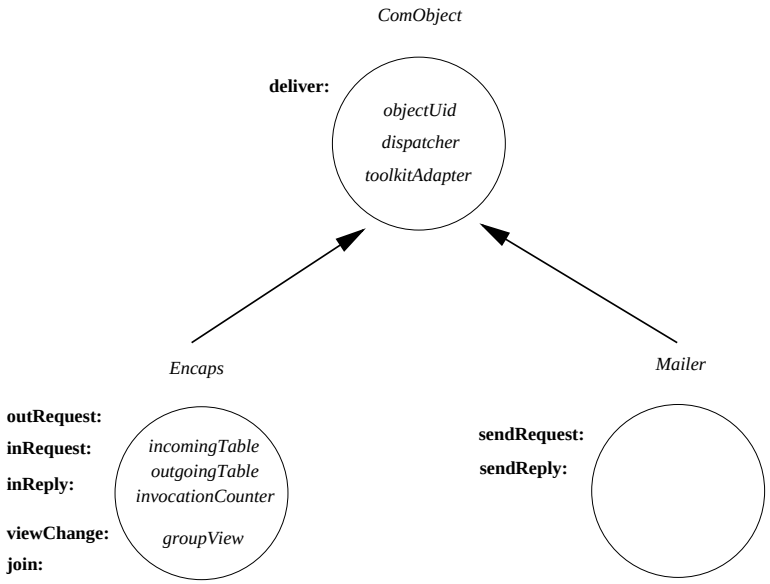

Figure 6.7: Les classes *ComObject*, *Encaps* et *Mailer*

- L'interface standard d'un encapsulateur est réalisé par les méthodes abstraites **outRequest:**, **inReply:** et **inRequest:**.
- La méthode abstraite **viewChange:** permet d'effectuer un traitement spécifique (par ex. élection d'une copie primaire) déclenché par la délivrance d'une nouvelle vue du groupe.
- La méthode **join:** est appelée au moment de l'initialisation d'un encapsulateur afin qu'il rejoigne un groupe. Elle consiste principalement à invoquer l'adaptateur afin que ISIS puisse effectuer l'opération de jonction (cf. section 6.7.2).
- La méthode **deliver:**, définie par la classe *ComObject*, est programmée par la classe *Encaps* de façon à appeler:
  - la méthode **inRequest:** lorsqu'un message transportant une requête (instance de la classe *RequestMessage*) est reçue par le distributeur.
  - la méthode **inReply:** lorsqu'un message transportant une réponse (instance de la classe *ReplyMessage*) est reçu.
  - la méthode **viewChange:** lorsqu'un message transportant une nouvelle vue est reçu.

La classe abstraite *Mailer* définit l'infrastructure commune à tous les messagers. L'interface standard d'un messager est réalisé par les méthodes abstraites **sendRequest:** et **sendReply:**. Chaque sous-classe réalise ces méthodes en invoquant l'adaptateur afin que ISIS puisse effectuer le multicast.

6.6.2 Gestion des invocations

La gestion des invocations est réalisée par les classes dont l'arbre d'héritage est représenté sur la figure 6.8.

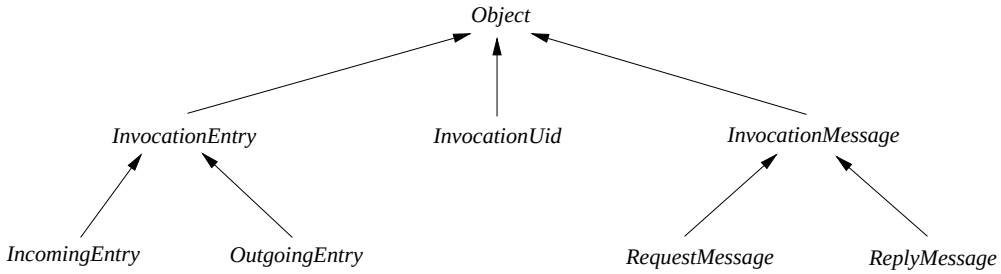

Figure 6.8: Les classes chargées de la gestion des invocations

Les instances de la classe *InvocationUid* permettent d'identifier de façon unique les invocations. Chaque instance comporte l'identificateur du client, celui du serveur, et le numéro de l'invocation. Cette dernière information correspond à la valeur du compteur stockée dans la variable *invocationCounter* définie par la classe *ComObject*. La classe *InvocationUid* définit des méthodes réalisant des traitements usuels: création, consultation et comparaison.

La classe abstraite *InvocationEntry* modélise les informations communes aux éléments des tables *incomingTable* et *outgoingTable*. Les informations spécifiques à une table sont modélisées par la sous-classe correspondante. La structure de ces tables est décrite à la section 5.5.4.

La classe *InvocationMessage* modélise les informations communes aux messages transportant les requêtes et les réponses. Il s'agit de l'identificateur du message, instance de la classe *InvocationUId*. Une instance de la classe *RequestMessage* est constituée par la requête d'invocation à soumettre au serveur. Cette information est elle même une instance de *Message*, la classe prédéfinie qui modélise les invocations en Smalltalk. Une instance de la classe *ReplyMessage* est constituée par un objet représentant la réponse. La classe de cet objet dépend de la méthode invoquée.

Synthèse

La figure 6.9 résume le rôle de la couche Communication au cours d'une invocation. Il s'agit du même exemple que celui de la figure 6.5.

On retrouve la modélisation symétrique de l'invocation. Les flèches en trait continu correspond au trajet de la requête alors que les flèches en trait discontinu correspond au trajet de la réponse. Les flèches à pointe creuse représentent des invocations locales (cf. section 5.2.1). Les flèches en boucles correspondant aux appels des méthodes **inRequest:** et **inReply:** illustrent l'aspect réflexif de ces appels. En effet, lorsque le distributeur (rectangle étiqueté *aSQDispatcher*) appelle la méthode **deliver:** d'un encapsulateur, ce dernier appelle sa propre méthode **inRequest:** ou **inReply:** selon qu'il s'agit d'une requête ou d'une réponse.

6.7 Couche Réseau

La couche Réseau réalise le pilotage de chaque nœud (cf. section 6.7.1), et assure l'interface avec ISIS (cf. section 6.7.2). Elle est constituée par des classes Smalltalk et par un programme, écrit en C, s'exécutant dans un processus différent. La communication entre ce processus et l'image

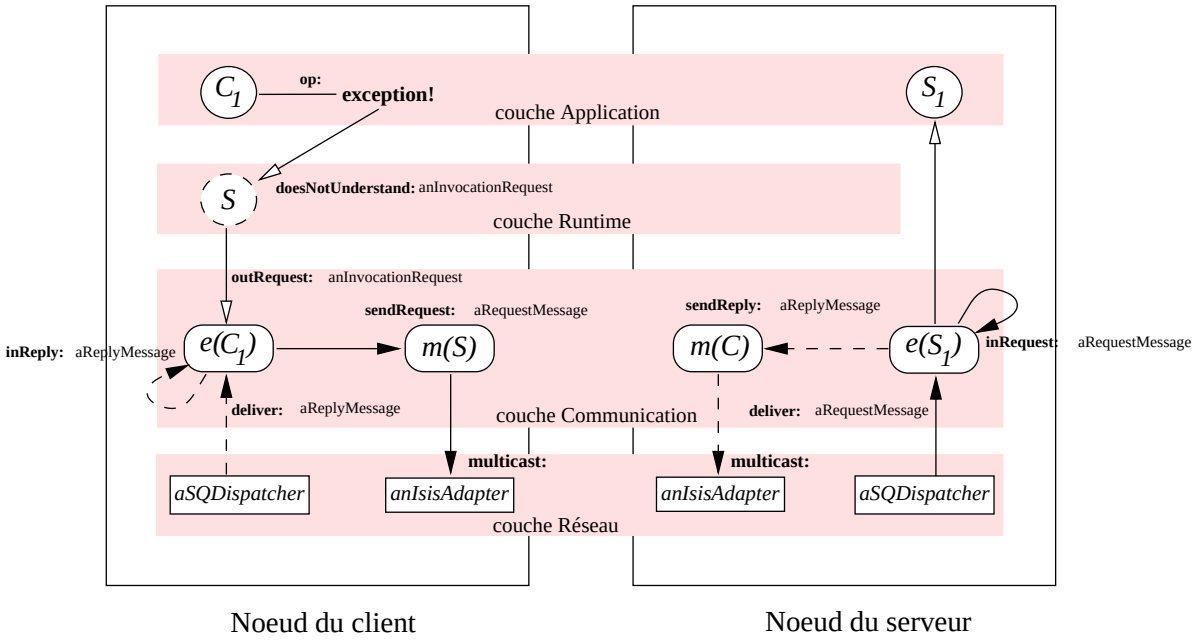

Figure 6.9: Rôle de la couche Communication au cours d'une invocation

Smalltalk (i.e. le processus contenant les objets Smalltalk) est réalisée à travers un socket UNIX.

6.7.1 Pilotage d'un nœud

Sur chaque nœud, le pilote, instance de la classe *NodeDriver*, coordonne la communication entre l'image Smalltalk et le processus externe. La figure 6.10 représente les objets participant au pilotage d'un nœud. On distingue:

- le pilote de nœud (*theNodeDriver*),
- l'adaptateur (*theISISAdapter*),
- le distributeur associé à l'adaptateur (*theESDispatcher*),
- la tampon associé au distributeur (*theExternalStream*),
- deux instances de la classe *SQDispatcher* associées à des objets communication non représentés sur la figure.

La classe abstraite *Dispatcher* et ses sous-classes mettent en œuvre la notion de distributeur. Un distributeur est constitué par un tampon et par une tâche (i.e. instance de la classe *Process*). Dès qu'il y a un message dans le tampon, la tâche le retire du tampon et appelle la méthode **deliver:** de l'objet associé au distributeur. Pour écrire un message dans le tampon, les clients d'un distributeur appelle la méthode **send:**.

Les sous-classes de *Dispatcher* correspondent à différentes mises en œuvre du tampon. La classe *SQDispatcher* réalise les distributeurs associés aux objets communication. Elle utilise une

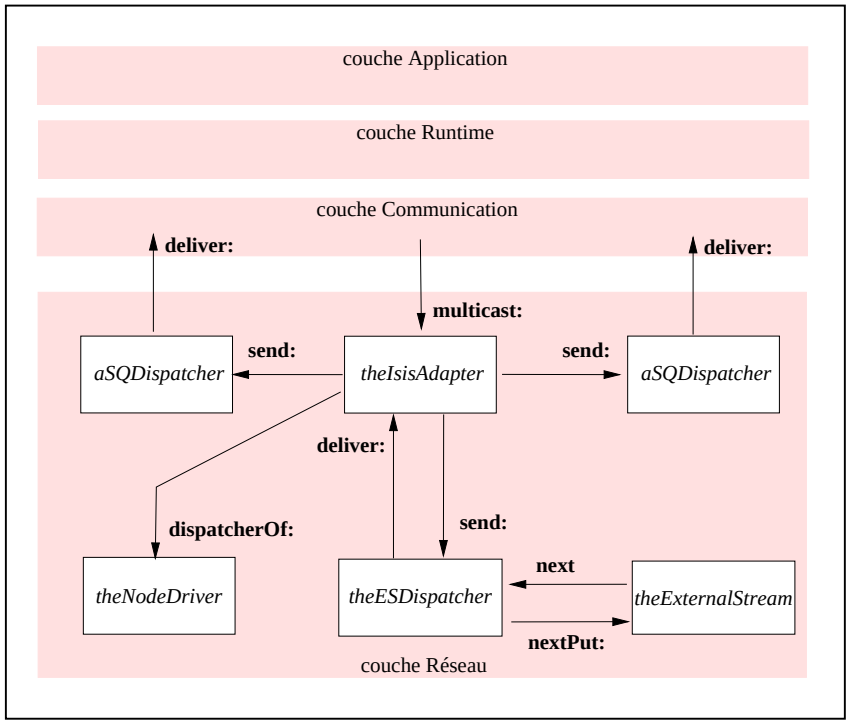

Figure 6.10: Pilotage d'un nœud

instance de la classe *SharedQueue* comme tampon. La classe *ESDispatcher* réalise le distributeur associé à l'adaptateur. Elle utilise comme tampon une instance de la classe *ExternalStream*. La classe *ExternalStream* modélise un flot d'octets bi-directionnel que l'on peut "connecter" à une source externe comme un fichier ou un socket.

La classe *IsisAdapter* réalise la notion d'adaptateur, la partie Smalltalk de l'interface avec ISIS. Sur chaque nœud, une instance de cette classe est chargée de:

- mettre en œuvre des méthodes d'interface à ISIS (par ex. **multicast:**);
- d'encoder et de décoder les vecteurs d'octets échangés entre l'image Smalltalk et le processus externe.

L'encodage consiste à construire un vecteur d'octets à partir d'une invocation de l'adaptateur. Selon la méthode appelée (par ex. **join:**, **multicast:**), le format du vecteur change. À chaque format, correspond une méthode spécialisée dans la construction du vecteur. Ce vecteur est ensuite passé au distributeur associé, pour qu'il le transmette, via le socket, au processus externe.

Le décodage est l'opération inverse. La méthode **deliver:** de la classe *IsisAdapter* est programmée de façon à reconnaître le format de chaque vecteur d'octets. Lorsque le format est identifié, une méthode spécialisée est appelée pour construire une invocation (instance de *SmalltalkMessage*).

Lorsque l'adaptateur *theISISAdapter* est invoqué (par ex. méthode **multicast:**) par un objet communication, il construit un vecteur d'octets et le transmet (méthode **send:**) au distributeur

theESDispatcher. Ce dernier écrit (méthode **nextPut:**) le vecteur dans le tampon *theExternalStream*.

Lorsqu'un vecteur d'octets arrive du processus externe, il est stocké dans le tampon *theExternalStream*. Le distributeur *theESDispatcher* lit ce vecteur (méthode **next:**) et le délivre (méthode **deliver:**) à l'adaptateur *theIsisAdapter*. Ce dernier décode le vecteur et construit une invocation. Puis, il invoque (méthode **dispatcherOf:**) le pilote *theNodeDriver* pour retrouver le distributeur associé au destinataire du message. Enfin, l'adaptateur *theIsisAdapter* transmet (méthode **send:**) le message au distributeur. Ce dernier peut alors délivrer le message à son destinataire.

6.7.2 Interface avec ISIS

L'interface avec ISIS est assurée par un programme C, s'exécutant dans un processus distinct. Ce programme est une application ISIS standard: il communique avec ISIS en utilisant la bibliothèque de fonctions C que ISIS met à la disposition du programmeur.

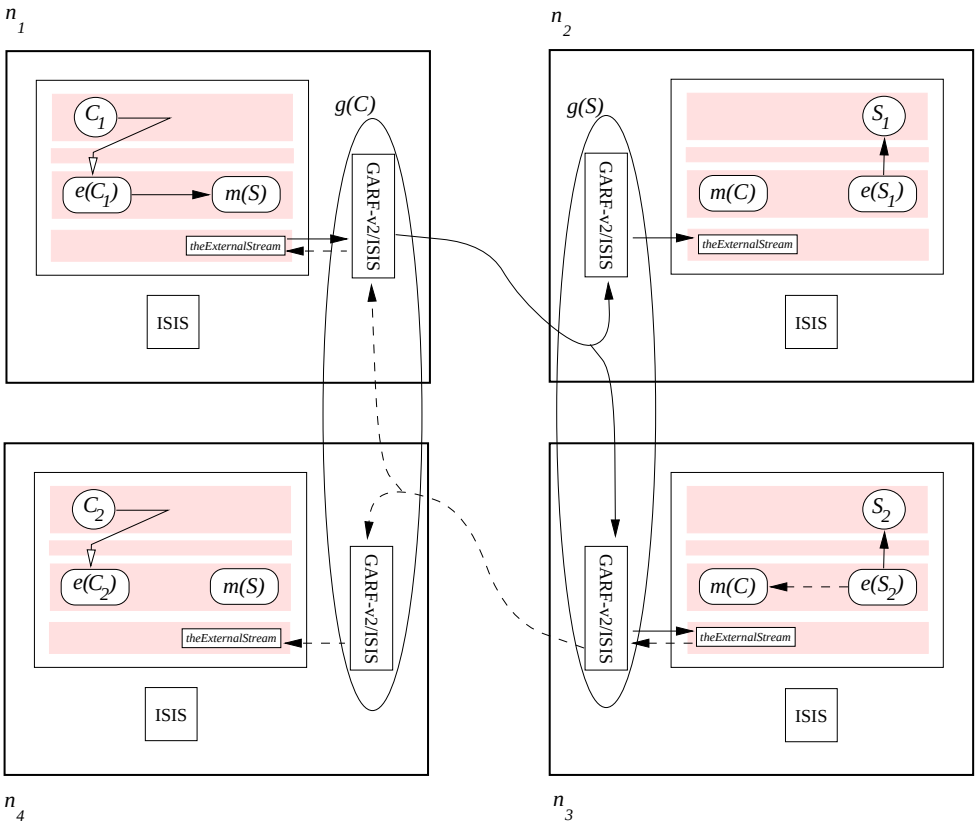

Figure 6.11: Interface avec ISIS

Le programme d'interface joue essentiellement un rôle d'adaptateur:

- d'une part, il interprète les vecteurs d'octets en provenance de l'image Smalltalk, et appelle les fonctions C correspondantes;

- d'autre part, il interprète les événements en provenance de ISIS, construit les vecteurs d'octets correspondant et les transmet à l'image via le socket.

L'aspect le plus important est l'utilisation des groupes de processus, réalisés par ISIS, pour construire des groupes d'encapsulateurs. Chaque groupe d'encapsulateurs est construit à partir d'un groupe de processus. La figure 6.11 illustre cette idée. Quatre nœuds sont représentés. Sur chaque nœud, on distingue trois processus UNIX: l'image Smalltalk, le processus d'interface GARF-v2/ISIS et le processus démon de ISIS.

Un client dupliqué activement, dont les copies C_1 et C_2 sont respectivement sur les nœuds n_1 et n_4, invoque un serveur dupliqué activement, dont les copies S_1 et S_2 sont respectivement sur les nœuds n_2 et n_3. Le groupe des encapsulateurs associés aux copies de C est modélisé par le groupe de processus $g(C)$ dont les membres sont les processus d'interface s'exécutant sur les nœuds n_1 et n_4. De même, Le groupe des encapsulateurs associés aux copies de S est modélisé par le groupe de processus $g(S)$ dont les membres sont les processus d'interface s'exécutant sur les nœuds n_2 et n_3.

Le pré-filtrage est utilisé à la fois pour les requêtes et les réponses. En effet, seul le coordinateur $e(C_1)$ invoque le messager $m(S)$. Cette invocation se traduit par un multicast transportant la requête et destiné au membres du groupe $g(S)$. De manière symétrique, la réponse n'est transmise que par le coordinateur $e(S_2)$.

6.8 Conclusion

Résumé

L'environnement GARF-v2 est une évolution de l'environnement GARF-v1, le premier prototype développé dans le contexte du projet GARF. Les deux prototypes utilisent le logiciel ISIS pour réaliser les abstractions de groupe et de multicast. La mise en œuvre du service N2M dans l'environnement GARF-v2 est structurée en quatre couches appelées couche Application, couche Runtime, couche Communication et couche Réseau. Chaque couche est programmée entièrement en Smalltalk, à l'exception de la couche Réseau dont une partie est écrite en langage C.

La couche Application réalise la notion d'objet application en définissant la classe abstraite *ApplicationObject*. Pour pouvoir utiliser le service N2M, le programmeur doit construire ces classes comme des sous-classes de *ApplicationObject*.

La couche Runtime réalise l'interface entre les objets application et les objets communication. Elle met en place un mécanisme d'interception des invocations basée sur la classe *Reference* et sur la gestion des exceptions en Smalltalk.

La couche Communication réalise la notion d'objets communication en définissant les classes abstraites *Encaps* et *Mailer*. Elle met en œuvre les quatre stratégies de duplication à l'aide de classes d'encapsulateurs et de messagers construites comme des sous-classes de ces deux classes de base.

La couche Réseau réalise l'infrastructure nécessaire aux objets communication pour effectuer des opérations concernant la gestion de groupe et les multicasts. Elle est constituée par des classes d'objets Smalltalk et par un programme s'exécutant dans un processus distinct. Ce processus est utilisé pour mettre en œuvre un groupe d'encapsulateurs à partir d'un groupe de processus.

Commentaires

L'environnement GARF-v2 est encore au stade de prototype. Une phase de tests et d'ajustements est encore nécessaire afin d'éprouver sa robustesse et ses performances. Cette phase est agendée comme une des perspectives de ce travail de thèse.

Chapitre 7

Conclusions

Ce chapitre dresse un bilan de ce travail de thèse et en présente les perspectives.

7.1 Bilan

Les contributions principales de cette thèse sont:

1. une classification des problèmes posés par l'invocation entre objets dupliqués, sous l'angle de l'encapsulation de la duplication;
2. une modélisation symétrique de l'invocation permettant d'exprimer l'encapsulation de la duplication du client et du serveur;
3. la conception du service d'invocation N2M qui réalise l'encapsulation de la duplication en se basant sur la modélisation symétrique;
4. une mise en œuvre du service N2M dans l'environnement GARF-v2 réalisé en Smalltalk, et basé sur ISIS.

L'analyse des problèmes posés par l'invocation entre objets dupliqués a permis de les classer sous l'angle de l'encapsulation de la duplication. Cette dernière a été définie comme étant la conjonction de l'encapsulation de la pluralité et de l'encapsulation des stratégies de duplication. L'encapsulation de la pluralité consiste à cacher le fait qu'un objet dupliqué est constitué par un groupe de copies. L'encapsulation des stratégies de duplication consiste à cacher les protocoles de communication utilisés pour garantir la cohérence des copies.

La modélisation classique de l'invocation est asymétrique. Elle présente l'inconvénient de ne pas permettre de réaliser l'encapsulation de la duplication d'un client. Par conséquent, une modélisation symétrique de l'invocation a été proposée au chapitre 4. Cette modélisation a la particularité de considérer la transmission de la requête et la transmission de la réponse comme deux instances du même problème. Elle permet ainsi de réaliser l'encapsulation de la duplication du client et du serveur.

Le service N2M est un service d'invocation pour objets dupliqués basé sur la modélisation symétrique. Il permet à des objets dupliqués de communiquer en utilisant des invocations locales, comme s'ils n'étaient pas dupliqués. Le service N2M atteint cet objectif en réalisant l'abstraction de représentants symétriques. Un représentant symétrique est une sorte de mandataire étendu,

représentant aussi bien le serveur sur les nœuds du client, que le client sur les nœuds du serveur. Les représentants symétriques sont construits à partir d'objets communication qui utilisent les notions de groupes d'objets et de multicasts afin de mettre en œuvre les stratégies de duplication.

L'abstraction de groupe augmentée de la sémantique vue-synchrone a été choisie pour exprimer les défaillances dans le cas des quatre stratégies de duplication. Ce choix a été fait dans un but de simplification. Il aurait été intéressant mais aussi complexe, d'essayer d'identifier la sémantique minimale dont a besoin chaque stratégie de duplication.

L'environnement GARF-v2 a permis d'éprouver la faisabilité du service N2M. Cependant, l'état actuel du prototype GARF-v2 n'a pas permis d'effectuer des mesures quantitatives dans le but de comparer les performances relatives des stratégies de duplication et des techniques de filtrage.

7.2 Perspectives

Une direction de recherche intéressante consiste à identifier la qualité minimale de l'information relative aux défaillances, nécessaire pour chacune des stratégies de duplication présentées. L'intérêt de cette recherche est d'identifier plus précisément les abstractions sur lesquelles le service N2M doit se baser. Ainsi, des mises en œuvre plus efficaces pourraient en résulter.

Sur un plan pratique, il parait intéressant de comparer les performances relatives du pré-filtrage et du post-filtrage, d'une part, et les performances relatives des différentes stratégies de duplication, d'autre part.

Depuis deux ans, la norme CORBA [OMG 95] connaît un succés toujours croissant, mais elle ne comporte aucune spécification de mécanismes de duplication. La mise en œuvre de N2M dans le contexte CORBA est certainement une direction de recherche prometteuse. Elle pourrait s'inscrire dans la continuation de travaux en cours, comme ceux décrits dans [Felber 96].

Liste des figures

2.1 Faute, erreur et défaillance 8
2.2 Classes de défaillances 9
2.3 Dépendances client-serveur entre les composants du système 20
2.4 Duplication des s-composants 21
2.5 Duplication des serveurs 22
2.6 Duplication des clients et des serveurs 23
2.7 Principe de la duplication active 25
2.8 Principe de la duplication passive 26
2.9 Principe de la duplication semi-active 28
2.10 Principe de la duplication coordinateur-cohorte 29

3.1 Un exemple de synchroniseur d'objets 39
3.2 Un exemple de mandataire 40
3.3 Un exemple d'objet dispersé 41
3.4 Un exemple de troupeau d'objets 42
3.5 Un exemple de communauté d'objets 44
3.6 Mise en œuvre d'un objet dupliqué à l'aide d'un objet fragmenté 46
3.7 Un exemple de groupe d'interfaces 47

4.1 Duplication du serveur: a. active, b. passive 58
4.2 Duplication du client: a. active, b. passive 60
4.3 Duplication du client et du serveur: a. active-active, b. active-passive, c. passive-active, d. passive-passive 62
4.4 Duplication active du client et du serveur: a. modélisation symétrique, b. modélisation asymétrique 65
4.5 Un exemple d'invocation d'un serveur dupliqué passivement par un client dupliqué activement 67
4.6 Partition d'un groupe de copies 70

4.7 Duplication passive du client et duplication active du serveur: deux scénarios de défaillances . . . 72
4.8 Duplication active du client et duplication passive du serveur: deux scénarios de défaillances . . . 75
4.9 Pré-synchronisation (a.) vs post-synchronisation (b.) . . . 79
4.10 a. serveur dupliqué activement, b. client dupliqué activement . . . 83
4.11 a. serveur dupliqué passivement, b. client dupliqué passivement . . . 85
4.12 a. serveur dupliqué semi-activement, b. client dupliqué semi-activement . . . 88
4.13 Importance de la propriété de vue-synchronisme pour un serveur semi-activement dupliqué . . . 89
4.14 a. serveur dupliqué "coordinateur-cohorte", b. client dupliqué "coordinateur-cohorte" 91

5.1 Notion de représentants symétriques . . . 96
5.2 Rôle des représentants symétriques . . . 98
5.3 Asymétrie des mandataires . . . 100
5.4 Modélisation des représentants symétriques . . . 101
5.5 Structure des objets application dupliqués . . . 105
5.6 Désignation des objets au cours d'une invocation . . . 108
5.7 Principe du pré-filtrage . . . 111
5.8 Pré-filtrage: méthode $outRequest()$ . . . 112
5.9 Pré-filtrage: méthode $inReply()$ . . . 113
5.10 Pré-filtrage: méthode $inRequest()$ . . . 114
5.11 Pré-filtrage: méthodes $viewChange()$ et $coordinatorElected()$ . . . 115
5.12 Principe du post-filtrage . . . 116
5.13 Post-filtrage: méthode $outRequest()$ . . . 117
5.14 Post-filtrage: méthode $inRequest()$ . . . 118

6.1 Architecture de GARF-v2 . . . 124
6.2 Un exemple de méthode garfNew: . . . 126
6.3 Un exemple illustrant le rôle de **garfNew:** . . . 127
6.4 La classe $Reference$ et ses sous-classes . . . 128
6.5 Rôle de la couche Runtime au cours d'une invocation . . . 129
6.6 Les classes chargées de la mise en œuvre des objets communication . . . 130
6.7 Les classes $ComObject$, $Encaps$ et $Mailer$ . . . 131
6.8 Les classes chargées de la gestion des invocations . . . 132
6.9 Rôle de la couche Communication au cours d'une invocation . . . 133
6.10 Pilotage d'un nœud . . . 134

6.11 Interface avec ISIS 135

A.1 Les 4 cas correspondant à la duplication active du serveur 154
A.2 Les 4 cas correspondant à la duplication passive du serveur 155
A.3 Les 4 cas correspondant à la duplication semi-active du serveur 156
A.4 Les 4 cas correspondant à la duplication coordinateur-cohorte du serveur 157

Liste des tableaux

2.1 Huits classes de détecteurs de défaillances 16

4.1 Duplication active, duplication passive: les quatre combinaisons possibles 61
4.2 Modélisation asymétrique: classification des problèmes liés à l'encapsulation de la duplication 63
4.3 Modélisation symétrique: classification des problèmes liés à l'encapsulation de la duplication 66
4.4 Valeurs des paramètres pour les quatres stratégies de duplication 92

5.1 Un exemple de table des objets communication 107

A.1 Récapitulation des 16 cas possibles 153

Bibliographie

[Agha 86] G. Agha. Actors: a model of concurrent computation in distributed systems. MIT Press, 1986.

[Anceaume 93] E. Anceaume. *Algorithmique de Fiabilisation de Systèmes Répartis.* PhD thesis, Université de Paris-sud (Paris XI), 1993.

[ANSA 91] ISA Project Core Team ANSA. *ANSA: Asssumptions, Principles and Structure.* In J.P. Warne, editor, Conference Proceedings of Software Engineering Environments. University College of Wales, March 1991.

[Bartlett 87] J. Bartlett, J. Gray & B. Horst. The evolution of fault-tolerant systems, chapter : Fault Tolerance in Tandem Computer Systems, pages 55–76. Springer-Verlag, 1987.

[Birman 85] K.P. Birman. *Replication and Fault-Tolerance in the ISIS system.* In Proceedings of the 10th ACM Symposium on Operating Systems Principles, pages 79–86, December 1985.

[Birman 93] K.P. Birman. *The Process Group Approach to Reliable Distributed Computing.* Communications of the ACM, vol. 36, no. 12, pages 36–53, December 1993.

[Birrell 84] A. D. Birrell & B. J. Nelson. *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems, vol. 2, no. 1, pages 39–59, February 1984.

[Birtwistle 73] G. Birtwistle, O. Dahl & K. Nygaard. SIMULA begin. Petrocelli Charter, 1973.

[Black 85] A.P. Black. *Supporting Distributed Applications: Experience with Eden.* ACM Operating System Review, vol. 19, no. 5, pages 181–193, December 1985. Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP).

[Black 87] A.P. Black, N. Hutchinson, E. Jul, H.M. Levy & L. Carter. *Distribution and Abstract Types in EMERALD.* IEEE Transactions on Software Engineering, vol. SE-13, no. 1, pages 65–76, January 1987.

[Black 93] A.P. Black & M.P. Immel. *Encapsulating Plurality.* In Oscar M. Niestraz, editor, Proceedings of the 7th European Object Oriented Programming Conference (ECOOP), number 707 in Lecture Notes in Computer Science, pages 56–79. Springer-Verlag, 1993.

[Booch 91] G. Booch. Object oriented design with applications. The Benjamin/Cummings Publishing Company, Inc, 1991.

[Budhijara 93] N. Budhijara, K. Marzullo, F.B. Schneider & S. Toueg. Distributed systems, chapter 8: The Primary-Backup Approach, pages 199–216. Addison-Wesley, 1993.

[Chandra 94] T. D. Chandra & S. Toueg. *Unreliable Failure Detectors for Reliable Distributed Systems.* Technical report TR94-1458, Dept. of Computer Science, Cornell Univ., Ithaca, New York 14853, October 1994.

[Chang 84] J. Chang & N.F. Maxemchuck. *Reliable Broadcast Protocols.* ACM Transactions on Computer Systems, vol. 2, no. 3, pages 251–273, August 1984.

[Chiba 93] S. Chiba & T. Masuda. *Designing an Extensible Distributed Language with a Meta-Level Architecture.* In O.M. Niestrasz, editor, Proceedings of the 7th European Object Oriented Programming Conference (ECOOP), number 707 in Lecture Notes in Computer Science, pages 483–501. Springer-Verlag, July 1993.

[Cooper 85] E.C. Cooper. *Replicated Distributed Programs.* ACM Operating Systems Review, vol. 19, no. 5, pages 63–78, December 1985.

[Cristian 91] F. Cristian. *Understanding Fault-Tolerant Distributed Systems.* Communications of the ACM, vol. 34, no. 2, pages 56–78, February 1991.

[Dwork 88] C. Dwork, N.A. Lynch & L. Stockmeyer. *Consensus in The Presence of Partial Synchrony.* Journal of the ACM, vol. 35, no. 2, pages 288–323, 1988.

[Eychenne 93] Y. Eychenne, M. Simatic, C. Baradel, L. Junot & V. Kohen. *The Use of Object Groups to Implement Dependability in a Process Control Supervision System.* In Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS), pages 660–665. IEEE, June 1993.

[Felber 96] P. Felber, B. Garbinato & R. Guerraoui. *The Design of a CORBA Group Communication Service.* In Proceedings of the Symposium on Reliable Distributed Systems (SRDS), 1996. to appear.

[Finke 93] S. Finke, P. Jahn, O. Langmack & K.-P. Lohr. *Distribution and Inheritance in the HERON Approach to Hetergoneous Computing.* In Proceedings of the 13th International Conference on Distributed Computing Systems (DCS), pages 399–408. IEEE Computer Society Press, May 1993.

[Foote 89] B. Foote & R.E. Johnson. *Reflective Facilities in Smalltalk-80.* In Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pages 327–335. ACM Press, 1989.

[Frolund 93] S. Frolund & G. Agha. *A Language Framework for Multi-Object Coordination.* In O.M. Niestrasz, editor, Proceedings of the 7th European Object Oriented Programming Conference (ECOOP), number 707 in Lecture Notes in Computer Science, pages 347–360. Springer-Verlag, July 1993.

[Garbinato 93] B. Garbinato, R. Guerraoui & K.R. Mazouni. *Programming Fault Tolerant Applications Using Two Orthogonal Object Levels.* In Proceedings of International Symposium on Computer and Information Sciences (ISCIS), 1993.

[Garbinato 94a] B. Garbinato, X. Défago, R. Guerraoui & K.R. Mazouni. *Abstractions pour la Programmation Concurrente dans GARF.* Calculateurs parallèles, June 1994.

[Garbinato 94b] B. Garbinato, R. Guerraoui & K.R. Mazouni. *Distributed Programming in GARF.* In R. Guerraoui, O. Nierstrasz & M. Riveill, editors, Object Based Distributed Programming, number 901 in Lecture Notes in Computer Science, pages 225–239. Springer Verlag, 1994.

[Goldberg 83] A.J. Goldberg & A.D. Robson. Smalltalk-80: The language and its implementation. Addison Wesley, 1983.

[Gourhant 92] Y. Gourhant. *An Object-Oriented Approach for Replication Management.* In Proceedings of the 2nd Workshop on the Management of Replicated Data, pages 74–77. IEEE Computer Society Press, November 1992.

[Guerraoui 94] R. Guerraoui, B. Garbinato & K.R. Mazouni. *The GARF Library of DSM Consistency Model.* In Proceedings of the ACM SIGOPS European Workshop, 1994.

[Guerraoui 96] R. Guerraoui, B. Garbinato & K.R. Mazouni. *Lessons fom Designing and Implementing GARF.* In Object Oriented Parallel and Distributed Computation, number 791 in Lecture Notes in Computer Science, pages 238–256. Springer-Verlag, 1996.

[Guttag 77] J. Guttag. *Abstract Data Types and the Development of Data Structures.* Communications of the ACM, vol. 20, no. 6, 1977.

[Hadzilacos 93] V. Hadzilacos & S. Toueg. Distributed systems, chapter 5: Fault-Tolerant Broadcasts and Related Problems, pages 97–145. Addison-Wesley, 2nd edition, 1993.

[IDS 93] Isis Distributed Systems Inc. IDS. Object groups: a response to the ORB 2.0 RFI. April 1993.

[Jul 93] E. Jul. *Separation of Distribution and Objects.* In R. Guerraoui, O. Niestrasz & M. Riveill, editors, Proceedings of the ECOOP Workshop on Object-Based Distributed Programming, number 791 in Lecture Notes in Computer Science, pages 46–54. Springer-Verlag, July 1993.

[Kernighan 77] B. Kernighan & D. Ritchie. The c programming language. Prentice-Hall, 1977.

[Laprie 92] J.C. Laprie, editor. Dependability: Basic concepts and terminology in english, french, german, italian and japanese, volume 5 of *Dependable Computing and Fault Tolerant Systems.* Springer-Verlag, 1992.

[Liskov 74] B. Liskov & S. Zilles. *Programming with Abstract Data Types.* SIGPLAN Notices, vol. 9, no. 4, pages 50–59, 1974.

[Little 91] M.C. Little. *Object Replication in a Distributed System.* PhD thesis, University of Newcastle-upon-Tyne (UK), September 1991.

[Maffeis 95] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming.* PhD thesis, University of Zurich (Switzerland), February 1995.

[Makpangou 92] M. Makpangou, Y. Gourhant, J-P. Le Narzul & M. Shapiro. *Fragmented Objects for Distributed Abstractions.* In T.L. Casavant & M. Singhal, editors, Advances in Distributed Computing: Concepts and Design. IEEE Computer Society Press, 1992.

[Mazouni 92] K.R. Mazouni. *A Fault Tolerant Distributed Diary Manager.* In Proceedings of the Digital Equipment Computer Society, pages 227–232, 1992.

[Mazouni 94] K.R. Mazouni, B. Garbinato & R. Guerraoui. *Programmation d'une application distribuée avec l'environnement GARF.* In Proceedings of the 3rd Maghrebian Conference on Software Engineering and Artificial Intelligence. Maghrebian Information Processing Society, 1994.

[Mazouni 95a] K.R. Mazouni, B. Garbinato & R. Guerraoui. *Building Reliable Client-Server Software Using Actively Replicated Objects.* In I. Graham, B. Magnusson, B. Meyer & J.-M. Nerson, editors, Proceedings of the Conference on Technology of Object-Oriented Languages and Systems (TOOLS), pages 37–51. Prentice-Hall, 1995.

[Mazouni 95b] K.R. Mazouni, B. Garbinato & R. Guerraoui. *Filtering Duplicated Invocations Using Symmetric Proxies.* In Proceedings of the International Workshop on Object Orientation in Operating Systems (IWOOOS), pages 118–126. IEEE Computer Society Press, August 1995.

[Mazouni 95c] K.R. Mazouni, B. Garbinato & R. Guerraoui. *Invocation Support for Replicated Objects.* Technical report 95/120, École Polytechnique Fédérale de Lausanne, Département d'Informatique, 1995.

[Meyer 87] B. Meyer. *Eiffel: Programming for Reusability and Extendability.* SIGPLAN Notices, vol. 22, no. 2, 1987.

[Meyer 88] B. Meyer. Object-oriented software construction. Prentice-Hall, 1988.

[Muganga 95] B. Muganga, F. Pacull & K.R. Mazouni. *Visual Programming of Fault-Tolerant Distributed Applications.* In IEEE Symposium on Visual Languages. IEEE Computer Society Press, 1995.

[OFTA 94] OFTA. Informatique tolérante aux fautes, volume 15. Masson, 1994.

[Olsen 91] M.H. Olsen, E. Oskiewicz & J.P. Warne. *A Model for Interface Groups.* In Proceedings of the 10th Symposium on Reliable Distributed Systems, pages 98–107, October 1991.

[OMG 91] OMG & X/Open. *The Common Object Request Broker: Architecture and Specification.* Technical report 91.12.1, Object Management Group, 1991.

[OMG 95] Object Management Group OMG. *The Common Object Request Broker: Architecture and Specification Revision 2.0*, 1995.

[Otway 87] D. Otway & E. Oskiewicz. *REX: a Remote Execution Protocol for Object-Oriented Distributed Applications.* In Proceedings of the 7th International Conference on Distributed Computing Systems (DCS), pages 113–118. IEEE Press, September 1987.

[Pascoe 86] G.A. Pascoe. *Encapsulators: a New Software Paradigm in Smalltalk-80.* In Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pages 341–346. ACM Press, 1986.

[Pinson 91] L. J. Pinson & R. S. Wiener. Objective-c object-oriented programming techniques. Addison-Wesley, 1991.

[Powell 91a] D. Powell, editor. Delta4: A generic architecture for dependable computing. Springer-Verlag, 1991.

[Powell 91b] D. Powell & P. Verissimo. Delta4: A generic architecture for dependable computing, chapter 6: Distributed Fault-Tolerance, pages 89–123. Springer-Verlag, 1991.

[Renesse 94] R. Van Renesse & K.P. Birman. *Fault-Tolerant Programming using Process Groups.* In F. Brazier & D. Johansen, editors, Distributed Open Systems. IEEE Computer Society Press, 1994.

[Sabel 95] L.S. Sabel & K. Marzullo. *Election vs Consensus in Asynchronous Systems.* Technical report TR95-1488, Cornell University, Computer Science Department, 1995.

[Schneider 90] F.B. Schneider. *Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial.* ACM Computing Surveys, vol. 22, no. 4, pages 299–319, December 1990.

[Sens 94] P. Sens. *Conception et Mise en œuvred'une Plate-forme Logicielle de Tolérance aux Fautes pour le Support d'Applications Réparties.* PhD thesis, Université Pierre et Marie Curie (Paris VI), December 1994.

[Shapiro 86] M. Shapiro. *Structure and Encapsulation in Distributed Systems: The Proxy Principle.* In Proceedings of the 6th International Conference on Distributed Computing Systems (DCS), pages 198–204. IEEE Computer Society Press, May 1986.

[Shapiro 89] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin & C. Valot. *SOS: an Object-Oriented Operating System - Assessment and Perspectives.* Computing Systems, vol. 2, no. 4, pages 287–337, 1989.

[Shrivastava 94] S.K. Shrivastava & D.L. McCue. *Structuring Fault-Tolerant Object Systems for Modularity in a Distributed Environment.* IEEE Transactions on Parallel and Distributed Systems, vol. 5, no. 4, pages 421–432, April 1994.

[Stroustrup 94] B. Stroustrup. The c++ programming language, second edition. Addison-Wesley, December 1994.

[Taylor 92] C.J. Taylor. *A Status Report on Open Distributed Processing.* First-Class (Object Management Group newsletter), vol. 2, no. 2, pages 11–13, July 1992.

Annexe A

Récapitulation des 16 cas possibles

Cette section récapitule les 16 cas possibles lorsque l'on considère quatre stratégies de duplication pour le client et le serveur. Les 16 cas sont basés sur le même exemple. Un client C invoque un serveur S et aucune défaillance ne se produit. La requête est notée α et la réponse est notée β. Le serveur S est dupliqué en 3 exemplaires, ses copies sont notées S_1, S_2, S_3. Le client S est également dupliqué en 3 exemplaires et ses copies sont notées C_1, C_2, C_3. Le tableau A.1 permet de retrouver rapidement la figure correspondant à chacun des cas.

$str(S)$	$str(C)$			
	active	**passive**	**semi-active**	**coord.-cohorte**
active	figure A.1-a	figure A.1-b	figure A.1-c	figure A.1-d
passive	figure A.2-a	figure A.2-b	figure A.2-c	figure A.2-d
semi-active	figure A.3-a	figure A.3-b	figure A.3-c	figure A.3-d
coord.-cohorte	figure A.4-a	figure A.4-b	figure A.4-c	figure A.4-d

Tableau A.1: Récapitulation des 16 cas possibles

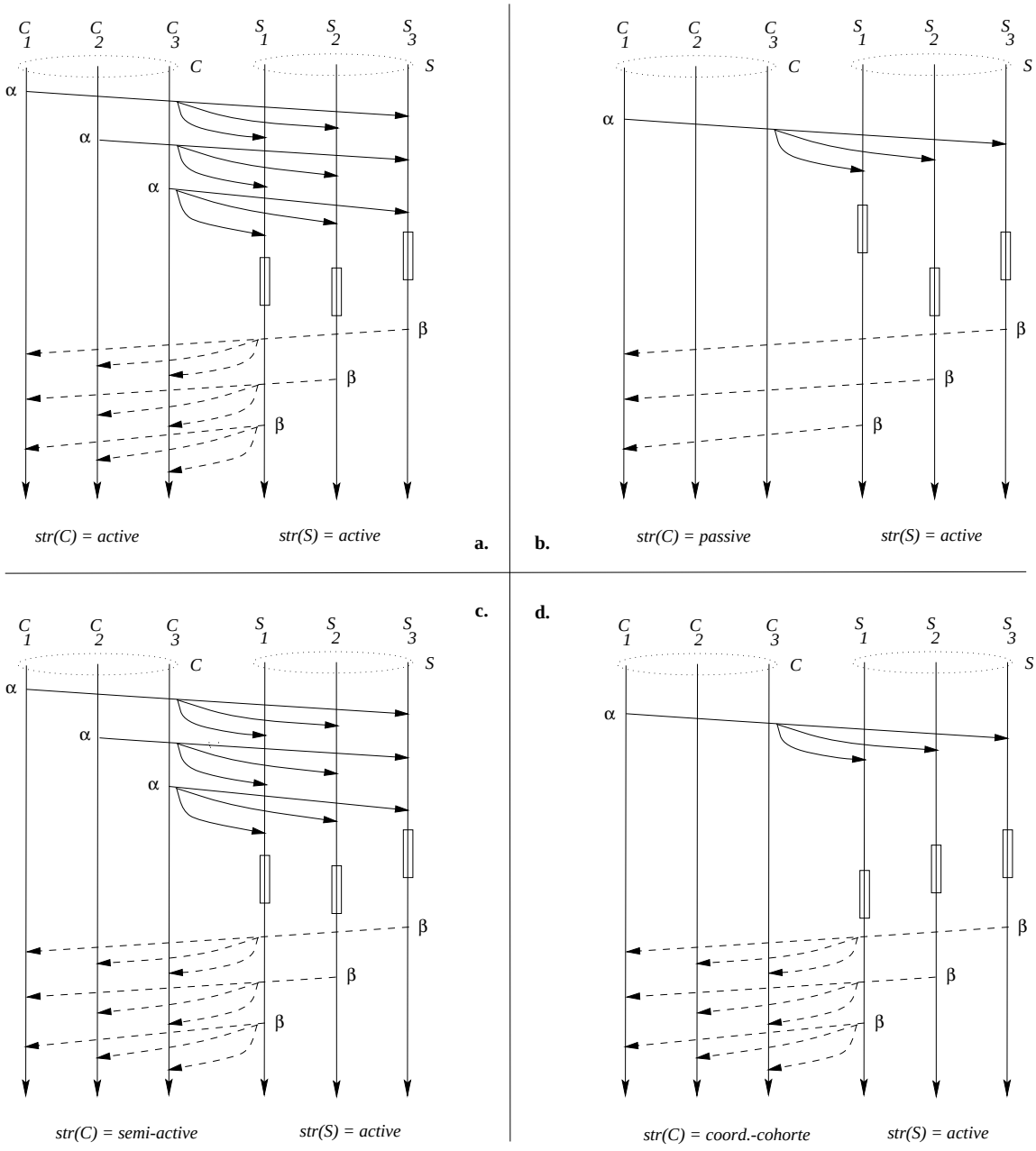

Figure A.1: Les 4 cas correspondant à la duplication active du serveur

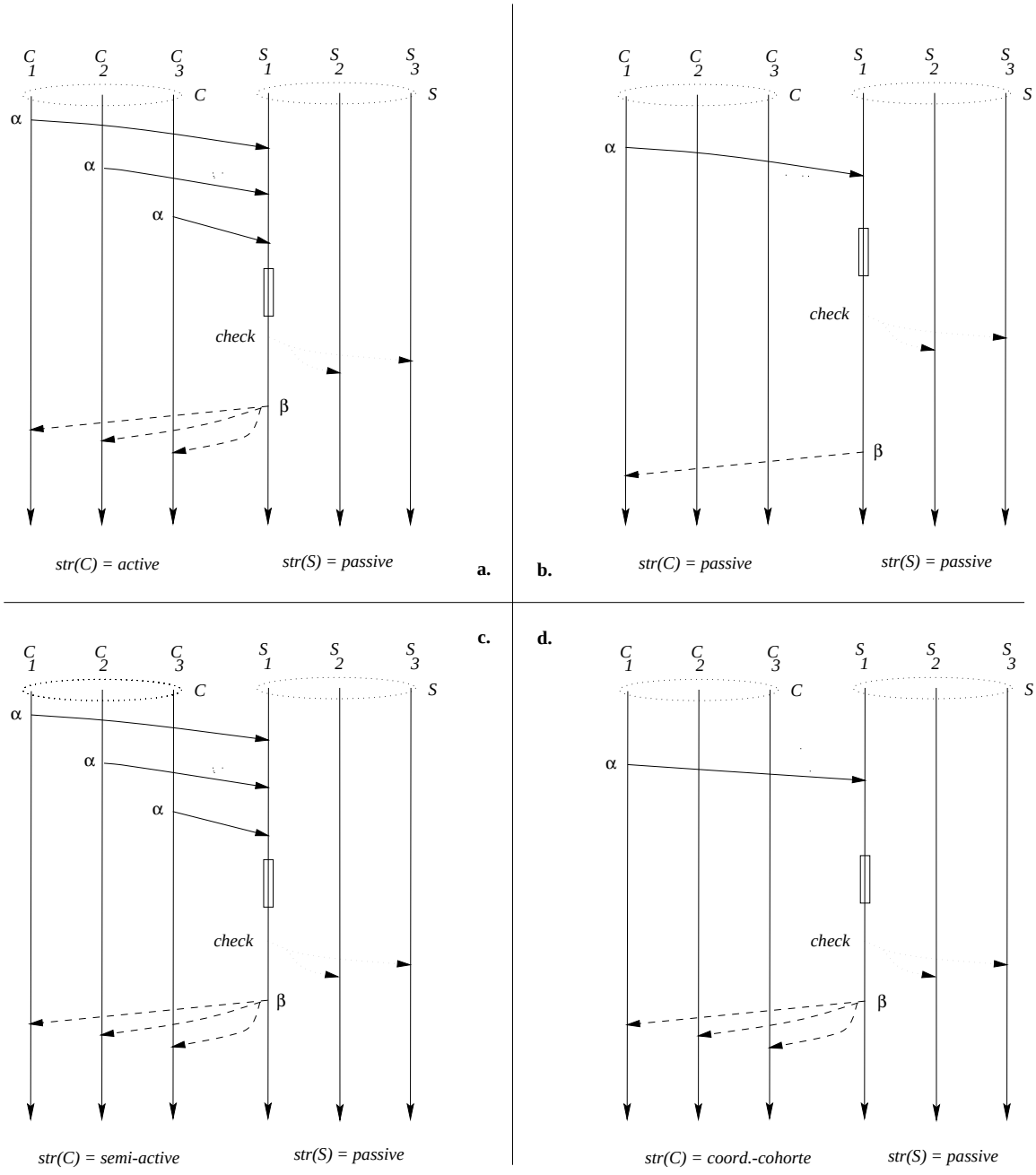

Figure A.2: Les 4 cas correspondant à la duplication passive du serveur

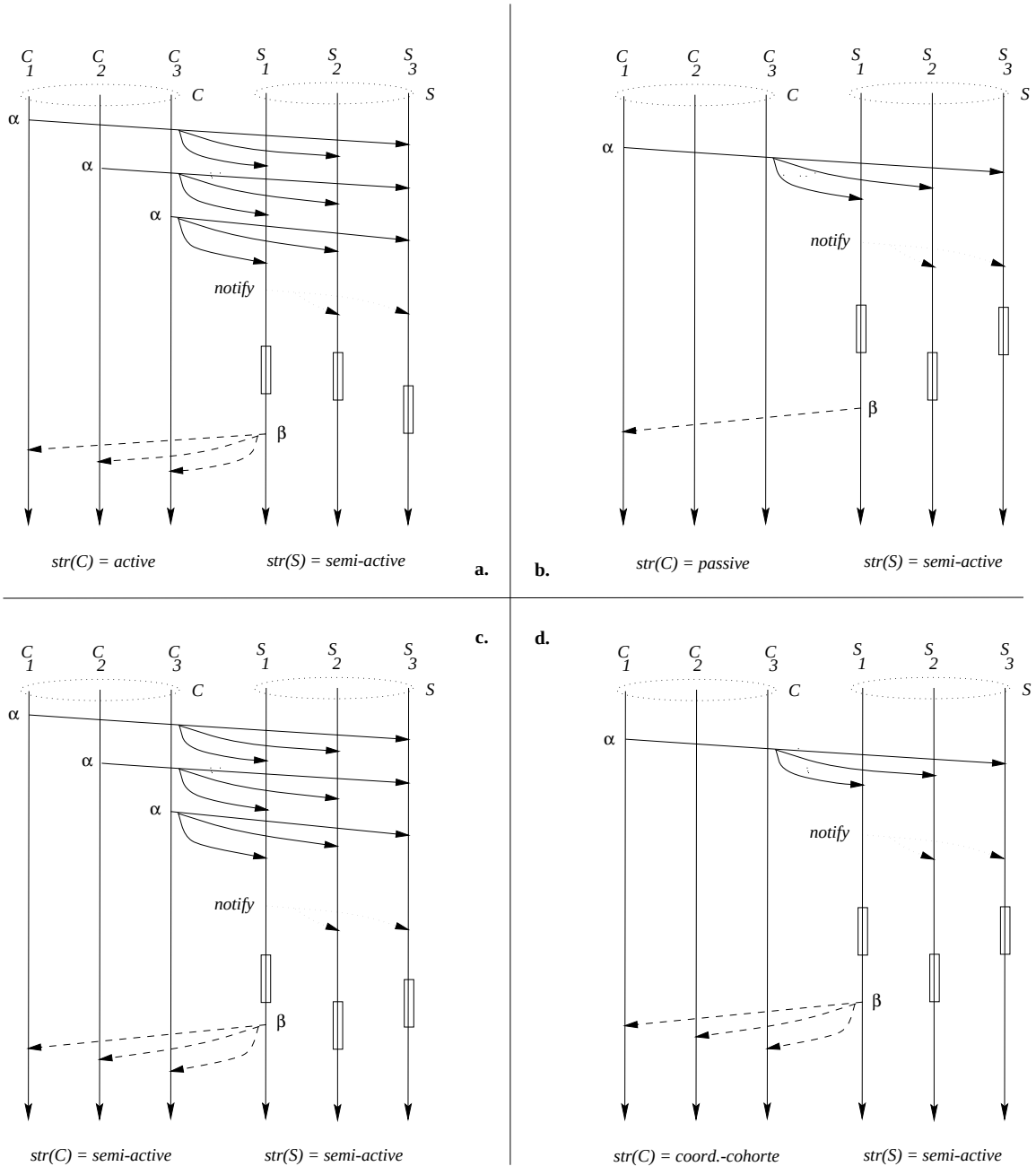

Figure A.3: Les 4 cas correspondant à la duplication semi-active du serveur

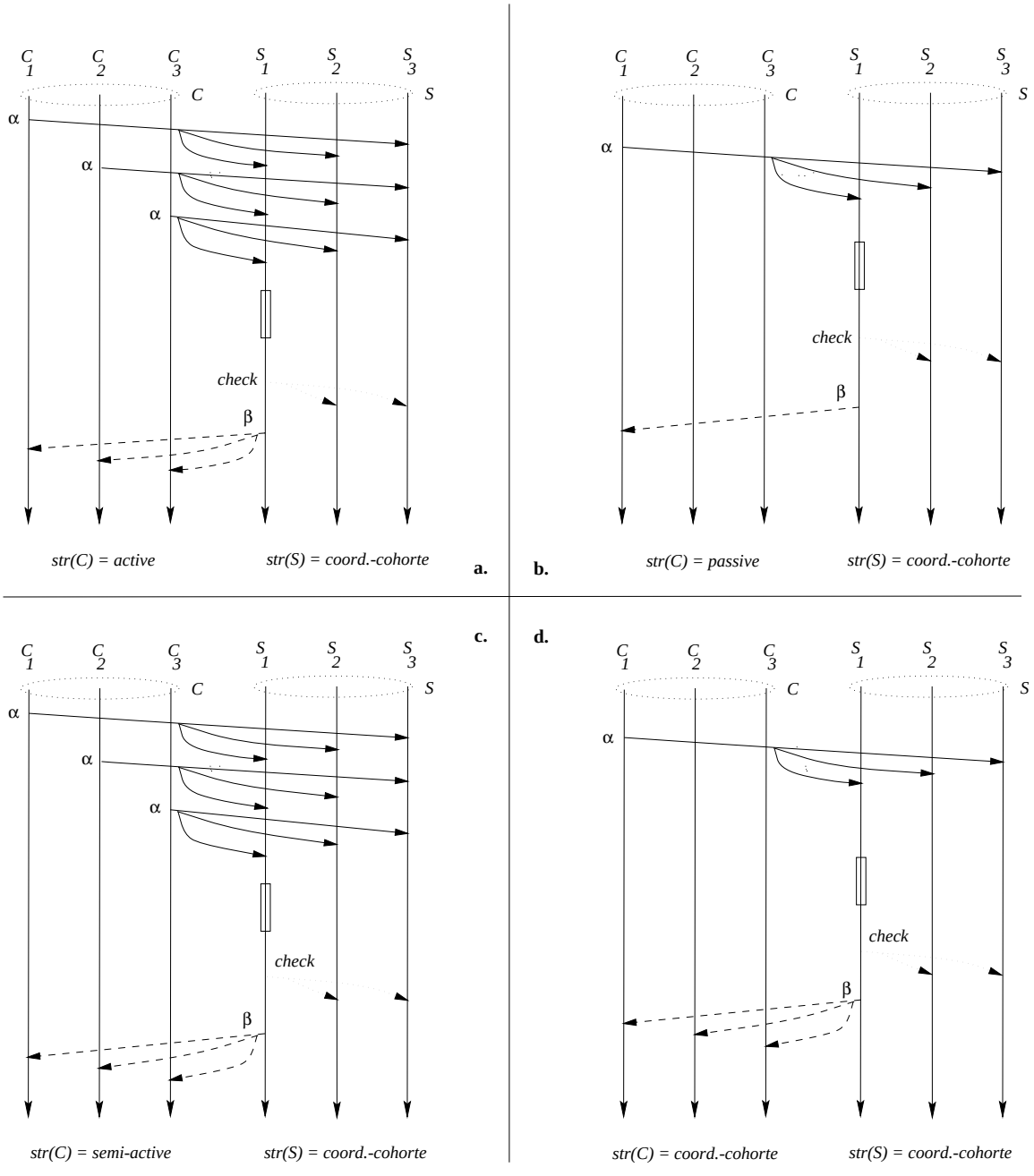

Figure A.4: Les 4 cas correspondant à la duplication coordinateur-cohorte du serveur

Annexe B

Curriculum Vitae

Je suis né le 23 juillet 1965 à Alger (Algérie). En 1983, j'entre à l'Institut National de formation en Informatique (INI) à Oued-Smar (banlieue d'Alger) pour entreprendre des études d'ingénieur en informatique. En 1988, mon diplôme en poche, je quitte l'Algérie pour la France. Je passe alors deux ans à l'université Pierre et Marie Curie de Paris où j'obtiens une maîtrise en informatique et un DEA en systèmes informatiques.

L'année de DEA a été un tournant dans mon cursus car c'est à ce moment que j'ai connu le domaine des systèmes répartis. En 1991, je viens à l'École Polytechnique Fédérale de Lausanne pour rejoindre l'équipe du Laboratoire de Systèmes d'Exploitation, dirigée par André Schiper. Dès mon arrivée, j'ai travaillé sur le développement d'applications réparties tolérantes aux fautes [Mazouni 92] à l'aide du logiciel ISIS. La continuation naturelle de ces travaux a été la définition du projet GARF. Ce projet s'est déroulé de 1993 à 1995, et a fait l'objet de nombreuses publications [Garbinato 93, Mazouni 94, Garbinato 94a, Guerraoui 94, Garbinato 94b, Muganga 95, Guerraoui 96]. Dès l'été 1994, je me suis consacré à l'étude de la communication entre les objets dupliqués [Mazouni 95a, Mazouni 95c, Mazouni 95b].

Outre mes activités de recherches, j'ai assuré les séances d'exercices pour le cours de Systèmes d'Exploitation de 1992 à 1995. Depuis, je donne ce cours aux étudiants de la section de Systèmes de Communication. Enfin, de 1991 à 1994, je me suis occupé de mettre en place et de gérer le parc des stations UNIX du LSE.