Caractérisation de la sûreté de fonctionnement de systèmes à base d'intergiciel

Eric Marsden

Numéro d'ordre : XXXX - Année 2004

Thèse

préparée au
Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS

en vue de l'obtention du
Doctorat de l'Institut National Polytechnique de Toulouse
Spécialité : Informatique et Télécommunications

par **Éric Marsden**

Caractérisation de la Sûreté de Fonctionnement de Systèmes à base d'Intergiciel

Soutenue le 27 février 2004 devant le jury :

Président	**Jean**	**Arlat**
Directeur de thèse	**Jean-Charles**	**Fabre**
Rapporteurs	**Gilles**	**Muller**
	João Gabriel	**Silva**
Examinateurs	**Philippe**	**David**
	Virginie	**Watine**

Cette thèse a été préparée au LAAS-CNRS
7, Avenue du Colonel Roche, 31077 Toulouse Cedex 4

Rapport LAAS Numéro XXXXX

aux vols de nuit

Avant-propos

Les travaux présentés dans ce mémoire ont été réalisés au *Laboratoire d'Analyse et d'Architecture des Systèmes* du *Centre National de la Recherche Scientifique* (LAAS-CNRS). J'exprime toute ma reconnaissance à Messieurs Jean-Claude Laprie et Mallik Ghallab, qui ont assuré la direction du LAAS-CNRS pendant mon séjour, pour m'avoir accueillis au sein de ce laboratoire. Je remercie également Messieurs David Powell et Jean Arlat, responsables successifs du groupe de recherche *Tolérance aux fautes et Sûreté de Fonctionnement informatique* (TSF) du LAAS-CNRS, pour m'avoir reçu dans ce groupe.

J'exprime ma vive reconnaissance à Monsieur Jean-Charles Fabre, Professeur à l'Institut National Polytechnique de Toulouse et Directeur de Recherche au CNRS, pour avoir encadré ces travaux de thèse, ainsi que pour le soutien, l'attention et la confiance qu'il m'a toujours accordés.

Je remercie Monsieur Jean Arlat, Directeur de Recherche au CNRS, pour l'honneur qu'il me fait en présidant mon Jury de thèse, ainsi que :

- Monsieur João Gabriel Silva, Professeur à l'Université de Coïmbra (Portugal),
- Monsieur Gilles Muller, Professeur à l'École des Mines de Nantes,
- Monsieur Jean-Charles Fabre, Directeur de Recherche au CNRS,
- Monsieur Philippe David, Ingénieur à l'ESA,
- Madame Virgine Watine, Ingénieur à Thales,

pour l'honneur qu'ils me font en participant à ce Jury, et particulièrement Messieurs João Silva et Gilles Muller qui ont accepté la charge d'être rapporteurs. J'exprime ma gratitude à João Silva pour qui la langue dans laquelle est rédigé ce manuscrit a représenté une charge de travail supplémentaire.

Ces travaux ont été partiellement financés par la Communauté Européenne dans le cadre du projet IST *Dependable Systems of Systems*. La participation à ce projet a été une expérience enrichissante, et j'ai tiré de nombreux enseignements des travaux mênés avec les différents partenaires du projet.

Je remercie Joëlle Penavayre, secrétaire du groupe TSF, pour sa constante disponibilité et son aide précieuse lors des tâches administratives. Je n'oublie pas l'ensemble des membres des services techniques et administratifs du LAAS-CNRS, dont la compétence contribue grandement aux conditions de travail excellentes rencontrées dans ce laboratoire.

Je tiens à saluer Nicolas Perrot et Christophe Attias qui ont contribué, au cours de leurs stages de fin d'études, à la réalisation de la plate-forme expérimentale à partir de laquelle ont été obtenus les résultats présentés dans ce mémoire. Je remercie Manuel Rodríguez Moreno pour des discussions profitables sur l'injection de fautes, et François Taïani et Marc-Olivier Killijian pour des bavardages concernant les systèmes répartis. Moi dire *thank you* à Annaïg Rabah, Maman et Benjamin Lussier pour leurs relectures attentives qui ont permis d'améliorer la fluidité de ce manuscrit.

Je profite de l'occasion pour saluer les membres de la Confrérie du Bureau 10, autant les titulaires que sont Mourad *lance-lance*, *Man* Jaja, Noura *solitude* × *multitude*, Ayda *respect*, Anis *chhhhut*, et le jeune Benjdor ; que les invités d'honneur *chérie* Sahboune, Annaïg *lourd-à-porter*, et Paulo 3ème. Je voudrais également témoigner de mon amitié pour mes collègues Juanito *haut-parleur*, *Radio* Oulfa, Saly, Clô, Crina *chui-pas-ton-baby*, meta-François, Marco *détecteur*, GNU/Ludovix et Guillaume *ès-résistant*, avec lesquels j'ai partagé avec plaisir ces années de travail.

Enfin, je transmets toute ma reconnaissance à `melbourne`, `sydney`, `adelaide`, `brisbane`, `hobart`, `perth` et `canberra` du *CoFFEE-rack*, auprès desquelles j'ai passé de nombreuses soirées agréables ; sans leur contribution infatigable ces travaux n'auraient pu être réalisés.

Table des matières

Table des figures

Introduction générale

Les secteurs industriels comme l'espace et l'avionique garantissent aujourd'hui de très hauts niveaux de sûreté de fonctionnement. Ces niveaux ont pu être atteints en utilisant des processus de développement particulièrement rigoureux, et en maîtrisant chaque étape de la conception à la fabrication du système.

Avec la maturation et l'ouverture à la concurrence de ces secteurs, de nouvelles contraintes de maîtrise des coûts favorisent la transition de cette « culture de l'exploit » vers une culture industrielle, où les méthodes de développement, la maîtrise du cycle de vie et le contrôle des coûts deviennent des facteurs incontournables.

Cette pression sur les coûts et les délais incite aujourd'hui les développeurs à privilégier la réutilisation de logiciels, plutôt que de procéder à des développements spécifiques pour chaque projet. Cette tendance à l'utilisation de composants logiciels « sur étagère », souvent développés par des tiers, est renforcée par des besoins technologiques de plus en plus complexes, liés en particulier à l'intégration des systèmes dans des réseaux de communication. L'utilisation de composants sur étagère permet aux industriels de se concentrer sur leur domaine de compétence, tout en s'épargnant l'effort de réaliser des fonctionnalités déjà développées dans d'autres secteurs.

Cette tendance à la réutilisation, ainsi que l'interconnexion croissante des systèmes, ont favorisé l'émergence de standards d'interface, qui permettent l'interopérabilité de systèmes même lorsqu'ils sont développés par des organisations indépendantes. Ainsi, la plate-forme CORBA a été standardisée pour répondre au besoin d'intégration de systèmes dans un contexte distribué. Cet *intergiciel de communication* facilite l'interaction entre des applications disparates, s'exécutant sur des plates-formes matérielles et logicielles hétérogènes. Pour les intégrateurs de systèmes répartis, ces technologies sont particulièrement attractives pour plusieurs raisons, technologiques et économiques : elles constituent un moyen rapide d'intégration de nouvelles technologies ; elles permettent une plus grande souplesse dans le déploiement des systèmes ; et elles constituent un moyen privilégié d'ouverture à d'autres systèmes informatiques.

Toutefois, cette attractivité est conditionnée par des craintes relatives à la robustesse des composants intergiciels, dont on ne peut pas garantir qu'ils aient bénéficiés de la rigueur du processus de développement utilisé dans le contexte de systèmes critiques. Les intégrateurs de systèmes répartis semi-critiques ont donc besoin d'assurances

supplémentaires sur la qualité et la robustesse des composants qu'ils intègrent au sein de leurs systèmes. Ils souhaitent des informations sur les modes de défaillance de l'intergiciel et les canaux de propagation d'erreur qu'il introduit, ainsi que des informations quantitatives leur permettant de comparer différentes implémentations candidates du point de vue de la sûreté de fonctionnement, afin de sélectionner le candidat le mieux adapté à leurs besoins.

La problématique que nous venons d'énoncer peut se résumer en deux points :

- Obtenir une meilleure connaissance des types de fautes et d'erreurs qui existent dans les systèmes à base d'intergiciel de communication ;
- Développer une méthode permettant de caractériser expérimentalement la robustesse de candidats intergiciels.

Il n'existe actuellement que peu de travaux permettant de répondre à ces interrogations. Notre contribution dans ce domaine est de proposer une méthode de caractérisation applicable à des intergiciels cibles, et de répondre ainsi aux problèmes des intégrateurs de systèmes semi-critiques et des fournisseurs de composants intergiciels.

Nous proposons dans ce document une méthodologie pour l'analyse de la sûreté de fonctionnement d'un intergiciel, basée sur quatre points : une analyse structurelle des intergiciels de communication, l'élaboration d'un modèle de fautes, une classification des modes de défaillance, et le développement d'un ensemble de techniques d'injection de fautes adaptées à l'intergiciel. Nous avons validé notre approche en menant des campagnes d'injection de fautes ciblant plusieurs implémentations de la norme CORBA.

Pour mieux comprendre les aspects techniques de cette problématique, nous proposons l'analogie suivante : l'intergiciel peut être comparé à une « glue » qui permettrait de construire des bâtiments en collant des parties hétérogènes (bois, plastique et ciment). Dans son principe, cette nouvelle technologie de construction de bâtiments paraît attractive : elle réduit les coûts de fabrication, et autorise la mise en œuvre de nouveaux styles architecturaux. Cependant, les constructeurs de bâtiments solides[1] (qui sont par nature relativement conservateurs quant aux technologies qu'ils intègrent dans leurs bâtiments) sont réticents à son utilisation parce qu'ils ont des doutes sur la résistance de la glue lorsqu'elle est soumise à des conditions exceptionnelles. Ainsi une glue destinée au grand public (comme un intergiciel sur étagère) n'est peut-être pas adaptée aux contraintes des constructeurs de bâtiments sûrs.

Nous avons développé une méthode pour évaluer la qualité de la glue (de ces intergiciels), non pas sous l'angle habituel de sa vitesse de séchage, mais vis-à-vis de la robustesse des bâtiments collés. Est-ce que la glue résiste à toutes les situations extrêmes auxquelles le bâtiment pourrait être soumis (orages, tremblements de terre) ? Est-ce que la glue autorise des interactions sournoises entre des éléments du bâtiment, à l'insu des constructeurs ?

[1] Parler de constructeurs de « bâtiments solides » peut surprendre : qui en effet peut envisager passer commande auprès d'un constructeur de bâtiments bancals ? Telle est pourtant la situation dans le secteur informatique, où une grande majorité des logiciels produits sont peu fiables, mais trouvent quand même des clients qui les utilisent dans leur vie quotidienne.

Ce mémoire est structuré en cinq chapitres.

Le chapitre 1 présente la problématique à laquelle nous nous intéressons. Il décrit le contexte de notre travail, et les raisons pour lesquelles la caractérisation des modes de défaillance des candidats intergiciels est nécessaire aujourd'hui. Dans un premier temps, nous décrivons les intergiciels de communication selon plusieurs points de vue, allant de la vision « haut niveau » d'objets répartis communiquant entre eux, à une vision plus détaillée faisant figurer les zones de confinement d'erreur dans le système. Nous présentons les différentes stratégies d'implantation d'un intergiciel de communication.

Nous introduisons ensuite les principes de la caractérisation de la sûreté de fonctionnement de systèmes informatiques, et examinons l'état de l'art relatif à la caractérisation d'intergiciels. Finalement, nous cherchons à identifier les différents points qui rendent délicate la caractérisation d'un intergiciel de communication.

Le chapitre 2 présente la méthode que nous proposons pour caractériser la robustesse d'implantations d'intergiciel. Nous donnons une technique pour construire des systèmes pouvant servir de cible aux expériences d'injection de fautes. Nous définissons les modèles de faute, la classification des modes de défaillance, et les charges applicatives que nous utilisons.

Enfin, nous décrivons un ensemble de techniques d'injection de fautes pour cibler l'intergiciel, et qui permettent d'évaluer l'impact de différents types de fautes sur celui-ci.

Le chapitre 3 donne les résultats de nos expériences d'injection de fautes obtenus par l'application de la démarche du chapitre 2 sur plusieurs implantations de la plate-forme CORBA, selon différents modèles de faute.

Le chapitre 4 analyse les résultats de ces expériences. Nous commençons par comparer les différentes techniques d'injection de fautes, suivant les modes de défaillance qu'ils ont provoqué chez les candidats testés, et vis-à-vis de leur coût de mise en œuvre. Nous examinons l'impact de la version du candidat testé sur les résultats de robustesse. Nous analysons un certain nombre de fautes du logiciel qui ont été révélés par nos expériences, en portant une attention particulière à l'étude des résultats d'un point de vue de la sécurité informatique. Nous apportons quelques éléments d'analyse de l'impact de l'architecture du courtier et de l'exécutif langage utilisé sur le comportement en présence de fautes.

En rassemblant les résultats obtenus des différentes campagnes d'injection de faute, et en pondérant les défauts de robustesse observés en fonction de leur gravité, nous proposons une métrique permettant une classification globale du degré de robustesse des différents candidats intergiciels que nous avons testés. Nous proposons finalement un ensemble de techniques d'encapsulation pour un intergiciel de communication, qui peuvent améliorer la détection d'erreurs et permettre la mise en place de mécanismes basiques de recouvrement d'erreurs.

Le chapitre 5 présente nos conclusions sur ce travail, et les renseignements que vous pouvons apporter aux intégrateurs d'intergiciel de communication et aux fournisseurs de ces logiciels. Nous donnons quelques perspectives sur l'extension de nos travaux à d'autres types d'intergiciel de communication.

Le lecteur intéressé trouvera à la fin de ce manuscrit un glossaire des différents acronymes et termes techniques employés.

Chapitre 1

Problématique

Ce chapitre a pour objectif de présenter le cadre de départ de nos travaux. Dans un premier temps, nous introduisons les concepts de base en sûreté de fonctionnement et systèmes répartis, que nous allons utiliser dans le développement de cette thèse. Nous définissons ce que nous entendons par intergiciel, et décrivons les caractéristiques qui les rendent attrayants pour les concepteurs de systèmes répartis. Nous exposons les craintes des intégrateurs quant à la fiabilité de ces composants logiciels de communication qui prennent placent au cœur de leurs systèmes.

Nous décrivons en quoi consiste la caractérisation de la sûreté de fonctionnement d'un intergiciel candidat. Nous listons quelques domaines d'application pour lesquels cette caractérisation est importante. Nous étudions l'état de l'art dans ce domaine, et expliquons pourquoi les travaux antérieurs sur la caractérisation de logiciels exécutifs comme des noyaux de systèmes d'exploitation doivent être étendus afin de tenir compte des caractéristiques particulières de ces nouvelles cibles.

1.1 Contexte des travaux

Le terme *intergiciel* (ou « middleware » en anglais) est utilisé pour désigner une couche de logiciel qui se situe entre les applications et le système d'exploitation. Le rôle de cette couche de « glue » varie suivant le type d'intergiciel considéré, mais il s'agit généralement de fournir au programmeur d'applications des abstractions de plus haut niveau que celles fournies par le système d'exploitation. Dans cette thèse, nous nous intéressons particulièrement aux *intergiciels orientés communication*, qui ont pour objectif de s'abstraire de la répartition des calculateurs, en présentant un modèle de programmation distribué basé sur l'invocation de méthodes sur des objets.

Le type d'intergiciel de communication le plus répandu en industrie est la plate-forme CORBA. Cette plate-forme est le résultat d'un travail de normalisation mené depuis

1989 par les membres du consortium OMG (*Object Management Group*, `www.omg.org`). CORBA permet de développer et déployer des systèmes répartis, en bénéficiant de facilités pour la communication entre objets. L'infrastructure répartie fournie par cette norme permet l'intégration et l'interopérabilité de composants logiciels et matériels hétérogènes. La plate-forme est basée sur la définition d'un bus à objets, qui permet la communication entre des objets répartis indépendamment :

- de leur localisation, puisque l'invocation d'une méthode sur un objet qui s'exécute localement est syntaxiquement identique et sémantiquement équivalent à l'invocation d'une méthode sur un objet distant ;
- des langages de programmation employés pour implanter les différents objets dans le système (un objet en Java peut interagir avec un objet implanté en COBOL, sans savoir qu'il communique avec un étranger) ;
- de la plate-forme d'exécution déployée sur les différents nœuds composant le système, que ce soit l'architecture des processeurs ou le type de système d'exploitation ;
- de la nature du réseau d'interconnexion entre nœuds du système, que ce soit le type de support physique d'interconnexion (machine multiprocesseurs à mémoire partagée, bus Ethernet ou VME, réseau Myrinet), ou les protocoles de communication employés.

Du point de vue d'un développeur de systèmes distribués sûrs de fonctionnement, les motivations pour l'utilisation d'un intergiciel de communication de type CORBA sont multiples :

- Réduire les coûts de développement et d'intégration des systèmes répartis. En effet, l'intergiciel fournit un modèle de programmation qui permet de s'abstraire de détails liés à la répartition, et de ne pas reprogrammer des primitives de communication ou des services comme la désignation. D'autre part, l'intergiciel facilite l'interopérabilité de composants logiciels, qui peuvent être implantés dans des langages de programmation diversifiés, et sur des plates-formes logicielles et matérielles hétérogènes.
- Bénéficier de souplesse en ce qui concerne le déploiement des applications. En effet, le masquage de la localisation fourni par l'intergiciel permet de modifier la répartition des objets dans le système, par exemple en migrant un service vers une autre machine, sans avoir à reconfigurer les clients de ce service.

Ces technologies ont connu depuis 1995 un succès considérable dans certains secteurs, en particulier pour les logiciels de gestion et de supervision. Devenant plus mûres, elles commencent à être déployées dans des systèmes ayant de plus fortes contraintes de sûreté de fonctionnement. Par exemple, CORBA est utilisé dans des équipements au cœur des réseaux de télécommunication, dans des applications bancaires, et dans des systèmes de commerce électronique. CORBA est utilisé dans les centres de contrôle-commande et de pilotage de satellites, et dans de nombreux systèmes militaires de contrôle-commande. CORBA est l'une des technologies envisagées pour le *Monde*

Ouvert[1] dans le cadre du programme A380 de Airbus, où elle pourrait servir de bus de communication pour intégrer des composants provenant de différents fournisseurs.

Dans ces systèmes, l'intergiciel n'est généralement pas développé en interne par l'équipe de développement, mais est plutôt intégré en tant que composant *COTS* (pour *Commercial-Off-The-Shelf* : composant sur étagère). Si l'utilisation de ces composants intergiciels est attrayante du point de vue économique et technologique, elle n'est pas sans poser problèmes sur le plan de la sûreté de fonctionnement [Arlat et al. 2000]. Les intégrateurs de systèmes s'interrogent sur la confiance qu'ils peuvent accorder à de tels composants, dont le processus de conception et de développement n'est généralement pas compatible avec les exigences de systèmes à fort besoin en sûreté de fonctionnement. Il faut aussi considérer que ces logiciels ont une durée de vie relativement courte, et ne bénéficient donc pas de statistiques de défaillance provenant de leur utilisation en service réel.

Les inquiétudes des intégrateurs quant à la fiabilité de composants COTS sont (clairement) justifiées, comme le démontrent quelques faits divers récents, où des défauts de conception d'un intergiciel de communication ont eu un impact considérable sur les systèmes qui intégraient ce composant :

- en juillet 2003, une vulnérabilité dans le module DCOM de la famille de systèmes d'exploitation Microsoft Windows® a été annoncée par le CERT[2] [CERT 2003]. Cette vulnérabilité permet à un attaquant distant de s'octroyer tous les privilèges sur la machine ciblée, en lui envoyant simplement une requête d'activation d'objet mal formée.

  Cette vulnérabilité a été exploitée en août 2003 par un ver malicieux nommé *W32/Blaster*, qui a sérieusement perturbé le fonctionnement du réseau Internet (ainsi que de nombreux réseaux d'entreprise) pendant plusieurs jours.
- en août 2002, une vulnérabilité dans l'implantation de référence de XDR (une bibliothèque permettant de transformer des données sous un format qui peut être transmis sur le réseau) a été annoncée par le CERT [CERT 2002a]. Le format XDR est utilisé pour la transmission de paramètres dans les appels de procédure à distance (RPC), et est utilisé par plusieurs services système sur les machines Unix. Cette vulnérabilité permettait à un utilisateur malveillant de s'octroyer à distance tous les privilèges sur la machine ciblée.

[1] L'architecture système et réseau embarquée de l'A380 est scindée en deux « mondes » : le monde dit « avionique », qui comporte toutes les fonctions critiques de l'avion comme le pilotage, la navigation, les commandes de vol ; puis le monde dit « ouvert », qui comporte des fonctions non critiques de l'appareil, telles que la maintenance, l'accès aux documentations embarquées, et les services aux passagers.

[2] Le Centre de Coordination du CERT est une organisation qui surveille les tendances en terme d'attaques contre les systèmes informatiques en réseau, et publie des bulletins d'alerte sur des vulnérabilités logicielles connues.

Ces exemples montrent à quel point il est important pour les intégrateurs de systèmes critiques de pouvoir caractériser le comportement en présence de fautes des composants logiciels qu'ils intègrent dans leurs systèmes. Toutefois, si la démarche de caractérisation de la performance d'un système est relativement bien comprise, la caractérisation de sa sûreté de fonctionnement l'est moins. L'un des premiers travaux sur l'analyse du comportement en présence de faute de composants COTS est [Salles 1999], qui propose une méthode de caractérisation des modes de défaillance de micronoyaux du commerce, basée sur des techniques d'injection de faute. Plus récemment, le projet de recherche *DBench* (*Dependability Benchmarking*) a eu pour objectif de développer des étalons de sûreté de fonctionnement pour des systèmes opératoires et des systèmes transactionnels, proches de l'esprit des étalons de performance. Comme nous le verrons au § 1.5, seuls de très rares travaux ont cherché à caractériser les modes de défaillance de systèmes à base d'intergiciel.

Un facteur important dans le processus de validation de composants COTS est leur nature de « boîte noire », puisque les utilisateurs disposent rarement des artefacts nécessaires à la compréhension et la maîtrise de la structure et du fonctionnement interne du composant : code source du logiciel, documents de conception, jeux de test utilisés par les développeurs. Il faut donc proposer des mécanismes de caractérisation qui permettent d'évaluer le composant en tant que boîte noire (c'est-à-dire sans exiger la lecture ou la modification du code source du composant). Toutefois, lorsque le code source de l'intergiciel cible est disponible, ce qui est le cas pour la majorité des candidats que nous avons évalués dans notre travail expérimental, on cherchera à proposer des méthodes d'analyse plus riches qui tirent partie de cette visibilité accrue sur la structure du logiciel.

La problématique de caractérisation de la sûreté de fonctionnement de l'intergiciel, à laquelle nous nous intéressons, peut se résumer en quelques points :

- Quels sont les modes de défaillance d'un intergiciel candidat ? Quelle est la couverture des mécanismes de tolérance aux fautes du système intégrant l'intergiciel ? Respecte-t-il le principe de silence sur défaillance ?
- Quel est le degré de confinement d'erreur assuré par ces composants intergiciels ? Comment analyser et mesurer l'impact d'erreurs se propageant depuis ces composants vers le reste du système ? Comment s'assurer qu'ils ne constituent pas un canal de propagation d'erreur dans le système ?
- Comment sécuriser ou isoler ces composants, afin d'assurer que leur défaillance ne se propage pas vers d'autres éléments du système, en particulier vers des couches plus critiques du système.
- Comment qualifier les différents candidats et apporter des éléments quantitatifs aux instances de certification, concernant le degré de confiance qu'on peut accorder à ces logiciels, lorsqu'ils sont intégrés au sein d'un système critique ?

> *On ne connaît que les choses que l'on apprivoise, dit le renard. Les hommes n'ont plus le temps de rien connaître. Ils achètent des choses toutes faites chez les marchands. Mais comme il n'existe point de marchands d'amis, les hommes n'ont plus d'amis. Si tu veux un ami, apprivoise-moi !*
>
> — *Le Petit Prince*, Antoine de Saint Exupéry

Il existe actuellement très peu de travaux qui permettent de répondre à ces interrogations. L'objectif de cette thèse est de proposer une méthode de caractérisation qui puisse être appliquée à des intergiciels cibles, et de répondre ainsi aux problèmes des intégrateurs de systèmes semi-critiques, et aux développeurs de composants intergiciels. L'intérêt de nos travaux pour un intégrateur de systèmes répartis semi-critiques est multiple :

- Obtenir une meilleure connaissance des types de fautes et d'erreurs qui existent dans les systèmes à base d'intergiciel de communication, et la manière dont ils peuvent se propager dans les systèmes pour provoquer des défaillances.
- Proposer une *méthode d'étalonnage* de la sûreté de fonctionnement d'un composant intergiciel qui permette de comparer la robustesse de différents candidats. Cette information, en complément d'autres critères comme des mesures de performance ou la qualité du dossier technique, et de l'assistance technique disponibles pour le produit, peut être prise en compte par l'intégrateur pour choisir l'implémentation la mieux adaptée à ses contraintes.
- Développer des *techniques d'encapsulation* permettant d'améliorer la robustesse d'une implémentation, en ajoutant des mécanismes de détection d'erreur et de recouvrement au système. Ces mécanismes permettent de renforcer le taux de couverture de l'hypothèse de silence sur défaillance, hypothèse qui est communément employée lors de la conception de systèmes répartis.

Quant aux fournisseurs d'intergiciel, nos travaux permettent d'identifier des points faibles dans leurs produits, tant au niveau de la conception de leur logiciel qu'au niveau de son implantation. Nous verrons par la suite que nos expérimentations ont permis d'identifier plusieurs fautes du logiciel dans différents courtiers CORBA.

Un intégrateur d'intergiciel COTS doit obtenir une confiance justifiée dans l'aptitude du logiciel qu'il place au cœur de son système à satisfaire ses exigences de sûreté de fonctionnement. Pour obtenir une confiance justifiée dans un composant sans pedigree, il faut caractériser son comportement en présence de fautes.

1.2 Notions de Sûreté de Fonctionnement

Avant d'aborder l'analyse de l'architecture des systèmes à base d'intergiciel de communication, nous allons rappeler la terminologie que nous utilisons dans la suite du document pour décrire les notions fondamentales de sûreté de fonctionnement. Cette terminologie et ces concepts sont empruntés à [Laprie 1995].

La *sûreté de fonctionnement* d'un système est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans la qualité du service qu'il leur délivre. L'objectif ultime des recherches dans le domaine est de pouvoir spécifier, concevoir, réaliser et exploiter des systèmes où la faute est naturelle, prévue et tolérable.

Une *défaillance* du système survient lorsque le service délivré dévie de l'accomplissement de la fonction du système, c'est-à-dire de ce à quoi le système est destiné. Une *erreur* est la partie de l'état du système qui est susceptible d'entraîner une défaillance, c'est-à-dire qu'une défaillance se produit lorsque l'erreur atteint l'interface du service fourni, et le modifie. Une *faute* est la cause adjugée ou supposée d'une erreur. On voit donc qu'il existe une chaîne causale entre faute, erreur et défaillance, représentée dans la figure 1.1.

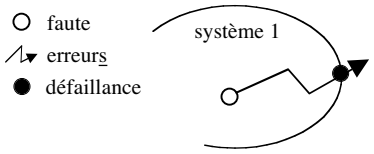

FIG. 1.1 – *Chaîne causale entre faute, erreur et défaillance*

Notons que dans les systèmes auxquels nous nous intéressons, qui sont formés par l'interaction de plusieurs sous-systèmes, les notions de faute, d'erreur et de défaillance sont récursives : la défaillance d'un sous-système devient une faute pour le système global (*c.f.* la figure 1.2 page suivante). Pour que le système soit digne de confiance, et que cette faute ne se propage pas jusqu'au service délivré aux utilisateurs, on cherche à éliminer les *canaux de propagation d'erreur* afin de briser cette chaîne causale. L'un des apports de nos travaux est de proposer des mécanismes permettant d'évaluer la probabilité que la défaillance d'un composant particulier, l'intergiciel, se propage au reste du système.

Un système ne défaille pas toujours de la même manière, ce qui conduit à la notion de *mode de défaillance*, que l'on peut caractériser suivant trois points de vue : le domaine de la défaillance, la perception des défaillances par les utilisateurs du système, et les conséquences des défaillances sur l'environnement du système. Concernant le domaine de défaillance, on distingue entre :

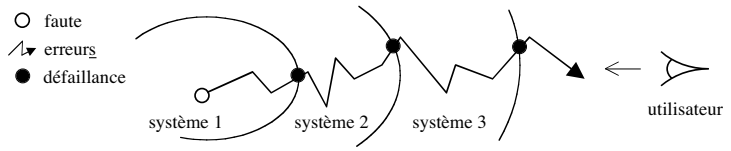

FIG. 1.2 – *Récursivité de la chaîne faute → erreur → défaillance*

- les *défaillances en valeur*, où la valeur du service délivré ne permet plus l'accomplissement de la fonction du système ;
- les *défaillances temporelles*, où les conditions temporelles de délivrance du service ne répondent pas aux besoins des utilisateurs.

Les *modes de défaillance par arrêt* sont relatifs à la fois aux valeurs et aux conditions temporelles : l'activité du système n'est plus perceptible aux utilisateurs. On distingue entre une absence d'activité par figement (où l'état des sorties reste constant), et par silence (dans un système réparti, le nœud concerné n'envoie plus de messages). Une classe importante de systèmes est formée de ceux qui sont *à silence sur défaillance*.

En classant les conséquences des défaillances sur l'environnement du système, on obtient une mesure de la sévérité, ou de la gravité, des défaillances. Le but des travaux de *caractérisation des modes de défaillance* d'un système est de déterminer quels sont ses modes de défaillance (généralement de façon expérimentale, en provoquant des états internes erronés du système, et en analysant quelles pourraient être leurs conséquences pour les utilisateurs du système), et de mesurer la sévérité de ces modes de défaillance.

Ce travail de caractérisation peut être comparé aux expériences d'évaluation de performance, qui constituent souvent le principal critère de sélection lors de l'acquisition d'un nouveau système. Notre travail consiste à évaluer le comportement du système non pas en exécution nominale ou « Panglossienne », où tout est pour le meilleur dans le meilleur des mondes possibles, mais dans un monde plus vraisemblable, qui n'est pas exempt de fautes.

1.3 Systèmes à base d'intergiciel de communication

Dans cette section, nous détaillons les types de systèmes répartis auxquels nous nous intéressons. Nous commençons par décrire le modèle de programmation offert par un système réparti, et le modèle à objets sur lequel nous nous appuyons. Nous analysons ensuite l'architecture d'un système à base d'intergiciel de communication suivant plusieurs points de vue, ce qui nous permet d'identifier les types de faute qui peuvent affecter ces systèmes. Nous regroupons ces fautes en classes, suivant leur nature et la couche du système où elles apparaissent. Nous analysons de quelle façons les erreurs

engendrées par ces fautes peuvent se propager dans le système, et quelles sont les défaillances qui peuvent en résulter.

1.3.1 Modèle de programmation et transparence de la répartition

Un *réseau d'ordinateurs* est un ensemble d'ordinateurs indépendants appelés nœuds, inter-connectés par des lignes de communication. Un *système réparti* à base d'intergiciel de communication offre un modèle de programmation qui permet d'utiliser un réseau d'ordinateurs comme un ordinateur unique, sans se soucier de la répartition des tâches sur les différents nœuds. La localisation des nœuds de calcul est masquée, ou rendue « transparente ». Nous appelons la couche de logiciel qui implémente cette abstraction un *intergiciel de communication*.

On distingue plusieurs types de transparence dans les systèmes répartis :

d'accès lorsque l'accès à une ressource se fait par une interface unique, que cette ressource soit locale ou distante ;

de localisation qui exprime la possibilité d'accéder à une ressource sans en connaître la localisation ;

à la réplication lorsque le service repose sur des exemplaires multiples de ressources, sans que leur multiplicité soit explicitée ;

à la concurrence lorsque plusieurs accès concurrents à la même ressource sont autorisés, sans que ces accès interfèrent entre eux ;

à la mobilité qui permet la migration d'objets sans affecter le déroulement des opérations en cours ;

à la défaillance lorsque la défaillance d'un objet ou d'un nœud peut être masquée aux utilisateurs en attendant le recouvrement de l'erreur correspondant[3].

Notons que la notion de transparence possède un sens particulier en informatique, puisque son dual est l'opacité. On dit qu'une fonctionnalité est « transparente » lorsque le programmeur n'a pas besoin de la prendre explicitement en compte pour en bénéficier. Cette transparence provient du masquage de la complexité sous-jacente, par un mécanisme dont l'implémentation est opaque.

Un intergiciel de communication a donc pour rôle de rendre transparente la localisation des objets ; on peut aussi dire qu'il est chargé de maintenir une *illusion de localité*, ou une abstraction dans laquelle la distance entre les nœuds est nulle.

[3] Citons toutefois la définition de Leslie Lamport : un système est réparti lorsque la défaillance d'un nœud dont on ignorait jusque là l'existence peut empêcher son ordinateur local de fonctionner.

1.3.2 Le modèle à objets

Les organismes internationaux que sont l'ISO (*International Organisation for Standardization*) et l'ITU-T (*International Telecommunication Union*) ont défini un modèle de référence pour le traitement ouvert dans un contexte réparti, appelé RM-ODP (*Reference Model of Open Distributed Processing* [ISO 1995]). RM-ODP peut être considéré comme un méta-standard, qui fournit des concepts architecturaux et une terminologie pour les systèmes répartis ouverts ; la norme CORBA est une instance concrète de ce méta-standard.

Nous nous intéressons à des systèmes répartis *ouverts*, qui sont encapsulés, réactifs, et extensibles dans le temps et dans l'espace. Ces systèmes sont composés d'un ensemble d'*objets*, chacun possédant un état et un comportement. Chaque objet est *encapsulé* par une interface, par laquelle passe toute interaction avec d'autres objets. Cette encapsulation autorise une capacité d'abstraction, qui permet l'indépendance par rapport aux mécanismes et aux technologies utilisées par l'objet, facilitant la construction de systèmes hétérogènes.

Les systèmes ouverts sont réactifs, dans le sens où ils fournissent un service en réponse à des requêtes adressées à des objets dans le système. Ces systèmes sont extensibles dans l'espace : la topologie d'interconnexion entre objets peut évoluer dans le temps. Ils sont aussi extensibles temporellement, puisque de nouveaux objets peuvent s'insérer dans le système au cours de son exécution.

1.3.3 Le modèle d'interaction

Une *interface* est un point d'accès à un objet, qui offre une vue partielle de ses capacités. Elle est constituée d'un ensemble de signatures d'opérations et de définitions de types de données. Toute interaction avec un objet passe par son interface, qui est identifiée par sa *référence*. Notons que le modèle à objets de CORBA est plus restrictif que celui de RM-ODP, puisqu'un objet ne peut posséder qu'une seule interface ; on confond alors les références d'interface et les références d'objet. Une référence d'objet CORBA s'appelle une IOR (ou *Interoperable Object Reference*).

Dans le monde CORBA, les interfaces sont définies en OMG IDL (*Interface Definition Language*). Ce langage définit un ensemble de types de données de base, tels que les nombres entiers, les nombres flottants et les chaînes de caractère ; il permet également de définir des types de données construits, tels que les structures et les séquences. OMG IDL permet la description de la syntaxe des opérations prises en charge par une interface. Une opération comporte un nom, un type de retour, le nom, le type et le mode de transmission de ses arguments (en entrée seule, en sortie ou en entrée-sortie) et la liste des exceptions qui peuvent être levées par cette opération (*c.f.* l'exemple de la figure 1.3 page suivante). Il est important de noter qu'une interface fournit une description purement syntaxique ; elle ne précise pas quelle est la sémantique des opérations.

```
interface NamingContext
{
    exception InvalidName { };

    void bind (in Name n, in Object obj)
       raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
    ...
}
```

FIG. 1.3 – *Extrait d'une interface* CORBA

Une interface décrit un *rôle*, ou un ensemble encapsulé de propriétés et de comportements. Une interface est indépendante du ou des objets qui seront utilisés à l'exécution pour implanter les opérations présentes dans l'interface ; en particulier elle ne dépend pas du langage de programmation qui sera choisi pour son implantation. Un rôle peut être joué par un objet, ou par un ensemble d'objets qui collaborent pour fournir un service.

Une *requête* permet à un objet de communiquer avec un autre objet. Elle est constituée d'une référence d'interface, d'un nom d'opération, et de paramètres (en mode in/out/inout). Les requêtes CORBA comportent également un élément de contexte (le *request context* en anglais), qui permet de transmettre des informations liées au contexte dans lequel s'exécute la requête (par exemple un identifiant de la transaction dans laquelle s'inscrit la requête).

Lors d'une interaction entre deux objets, le *client* est l'objet actif instigateur de la requête. L'objet passif qui répond à la requête du client est le *serveur*. Dans le monde CORBA, un objet peut jouer à la fois un rôle de serveur et un rôle de client ; cette configuration est particulièrement fréquente dans les objets du deuxième tiers d'une architecture à trois tiers.

Une requête CORBA peut être sans retour (*one-way*), auquel cas l'appelant ne reçoit pas de réponse (ni même d'accusé de réception) de la part de l'objet appelé. Plus communément, les requêtes sont avec retour (*two-way*), auquel cas le traitement de l'objet appelant est suspendu en attendant la réponse de l'objet appelé (l'appel est dit *synchrone* ou *bloquant*). Dans le cas d'une exécution nominale, l'objet appelant reçoit en retour le résultat de la requête. Si le traitement de la requête échoue, le client reçoit une *exception* qui lui indique la nature de l'erreur qui s'est produite.

Dans des plates-formes telles que CORBA, qui s'appuient sur les concepts de la programmation orientée objet, il existe un mécanisme d'*héritage* entre interfaces, basé sur une relation de sous-typage. Dans de tels systèmes, où la notion de *polymorphisme* est présente, l'interface désignée par une requête est déterminée dynamiquement, suivant le principe de la liaison dynamique. On peut alors parler d'*invocation de méthode* plutôt que de requête.

1.3.4 Le point de vue traitement

La notion de *point de vue* de la norme RM-ODP fournit une manière de décrire un système en fonction d'une problématique particulière. L'objectif est de décomposer une spécification de système réparti en plusieurs parties, les vues. Chaque vue est une description complète et auto-suffisante du système souhaité, destinée à une audience particulière. Les points de vues qui nous concernent dans cette thèse sont le point de vue traitement, qui permet de partitionner le système en un ensemble d'objets qui interagissent ; le point de vue ingénierie, qui concerne les mécanismes et l'infrastructure nécessaires à la mise en œuvre de la répartition ; et le point de vue technologique, qui décrit les choix de solutions technologiques à partir desquelles le système réparti est construit.

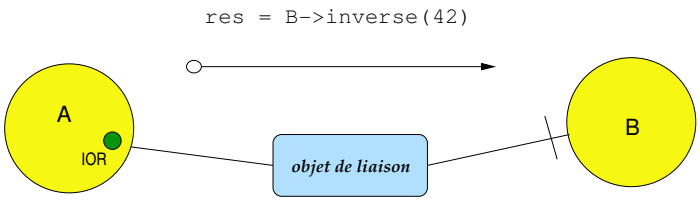

FIG. 1.4 – *Vue traitement d'un système à base d'intergiciel de communication*

La figure 1.4 présente un exemple de système CORBA du point de vue traitement. On voit un objet A invoquer une méthode `inverse` sur un objet B. Dans cette figure, l'intergiciel est représenté par l'objet de liaison, objet qui sert de support à l'interaction entre les objets A et B.

Notons que du point de vue traitement, nous n'avons pas de représentation de la localisation des objets ; on ne se préoccupe pas de savoir si A et B s'exécutent sur le même nœud ou non. Dans le modèle de programmation fourni par l'intergiciel de communication, l'invocation d'une méthode distante est identique à un appel de méthode locale.

Dans le modèle à objets de CORBA, l'objet de liaison est présent de manière implicite lors d'une invocation de méthode (dans d'autres systèmes, la représentation explicite de la liaison permet d'exprimer des contraintes sur des paramètres de qualité de service exigées de la liaison). On voit donc que l'intergiciel de communication est essentiellement invisible à ce niveau d'abstraction (et en définitive c'est exactement la raison pour laquelle l'intergiciel est utilisé).

Les spécifications de la norme CORBA se placent principalement dans le point de vue traitement. Ces normes font abstraction de toute considération sur les mécanismes employés par les fournisseurs d'intergiciel pour implanter la norme : ainsi elles ne précisent pas quelle doit être la représentation d'un objet CORBA en tant qu'entité

d'exécution, et elles ne décrivent pas quel type d'interaction doit avoir lieu avec le système d'exploitation. Ce manque de précision est volontaire : l'OMG souhaite ne pas limiter le choix des développeurs d'intergiciel en leur fixant des restrictions qui pourraient se révéler inappropriées à certains contextes ; tout choix d'implémentation qui n'affecte pas la vision qu'ont les programmeurs applicatifs est autorisé.

Si cette vision de haut niveau est suffisante pour le développement d'une application basée sur la plate-forme CORBA, des informations plus précises sur l'architecture et les choix d'implantation sont nécessaires à l'analyse de la sûreté de fonctionnement d'un système conçu autour de cette architecture. En particulier, il est important d'obtenir des informations sur les régions de confinement d'erreur dans le système. Ces informations deviennent visibles du point de vue ingénierie.

1.3.5 Le point de vue ingénierie

Le point de vue ingénierie de RM-ODP s'applique aux mécanismes nécessaires à la gestion de la répartition. Il permet de représenter de manière explicite la localisation des objets et la gestion des ressources qui leur seront attribuées. On peut considérer le point de vue ingénierie comme étant un méta-modèle des aspects ayant trait à la répartition pour le point de vue traitement.

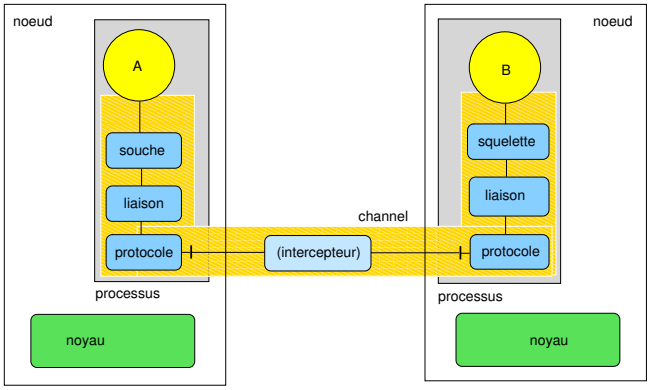

FIG. 1.5 – *Vue ingénierie d'un système à base d'intergiciel de communication*

La figure 1.5 illustre un système à base d'intergiciel de communication du point de vue ingénierie. Par rapport à la représentation de plus haut niveau donnée par le point de vue traitement, on voit apparaître dans cette figure les nœuds physiques qui hébergent les traitements, ainsi qu'une représentation plus détaillée des entités d'exécution qui correspondent aux objets. L'objet de liaison (qui représente l'intergiciel dans le point de vue traitement) a été décomposé en plusieurs parties :

- la *souche* (ou *stub* en anglais) emballe les requêtes sous une forme qui peut être traitée par le protocole de communication sous-jacent. On parle aussi de *mise à plat* de données ou de *marshalling*. La souche doit également déballer les réponses provenant de la couche de protocole pour les présenter à l'objet supérieur.
- le *squelette* (ou *skeleton* en anglais) déballe les requêtes entrantes qui lui sont transmises sous une forme « brute » par l'objet de protocole, avant de les transmettre à l'objet du niveau supérieur, puis emballe les réponses avant de les remettre à l'objet de protocole.
- l'objet protocole gère les communications entre objets. Il est chargé d'établir et de gérer les connexions réseau, et d'échanger des messages concernant la coordination du protocole de communication.

On voit apparaître dans cette figure les régions de confinement d'erreur (dans un nœud ou dans un processus), dont on examinera l'impact au § 1.3.8.

1.3.6 Le point de vue technologique

Le point de vue technologique consiste à étudier les solutions technologiques à partir desquelles le système réparti est construit. Il décrit la nature du support physique d'interconnexion entre nœuds, les protocoles de communication employés dans le système, et les langages de programmation utilisés. Ici nous nous intéressons en particulier aux mécanismes que l'intergiciel utilise pour apporter une illusion de transparence de la répartition.

Les intergiciels de communication diffèrent par le degré de transparence vis-à-vis de la répartition qu'ils offrent au programmeur d'application, c'est-à-dire de la force de l'illusion de localité qu'ils mettent en œuvre.

Les premiers intergiciels de communication étaient les mécanismes RPC (*Remote Procedure Call*, ou appel de procédure à distance) intégrés aux supports d'exécution de certains langages de programmation. Pour effectuer un RPC, le programmeur faisait appel explicitement à une primitive spéciale fournie par le support d'exécution, en précisant l'adresse réseau et le port de communication du service avec lequel il souhaitait communiquer. Ce mécanisme ne fournit ni la transparence d'accès, puisqu'un appel de procédure distant est syntaxiquement différent d'un appel de procédure local, ni la transparence vis-à-vis de la localisation, puisque le programmeur doit connaître l'adresse du nœud où s'exécute le service invoqué. Le degré de transparence fournie est donc limité au langage de programmation utilisé par l'autre processus.

Le mécanisme de RPC du système d'exploitation *Cedar* [Tay & Ananda 1990] a été le premier à ajouter un mécanisme de souches et de squelettes, permettant de rendre l'appel distant syntaxiquement identique à un appel de procédure local. La souche est une représentation de l'ensemble des procédures accessibles à distance sur le

serveur, exprimée dans le langage de programmation du client. Elle fournit la transparence d'accès, puisque son rôle est d'assurer qu'un appel de procédure distante est syntaxiquement équivalent à un appel de procédure locale. Le *squelette* joue un rôle équivalent à la souche, mais du côté serveur.

Les RPC de *Cedar* sont un exemple de la famille des intergiciels de communication de seconde génération. Ces derniers offrent un niveau d'abstraction plus élevé que les RPC basiques, en fournissant un plus grand degré de transparence au programmeur. En contrepartie, le degré d'intrusivité de l'intergiciel augmente, puisqu'il doit mettre en œuvre des mécanismes pour intercepter les interactions avec les objets distants.

Les intergiciels de communication de troisième génération sont représentés par les plates-formes CORBA et DCOM, auxquelles nous nous intéressons dans cette thèse. Ces intergiciels proposent un modèle de programmation dans lequel les interactions se font entre objets, plutôt qu'entre processus. Ils sont basés sur les principes des langages de programmation orientés objet, et enrichissent l'appel de procédure à distance par un mécanisme de liaison dynamique de l'objet invoqué, pour fournir un modèle d'interaction basé sur l'appel de méthode. Ces intergiciels offrent également des moyens pour gérer l'activation des objets.

Le monde CORBA est fortement multi-culturel, tant par la diversité des systèmes d'exploitation, allant de grands systèmes tels que *MVS* aux plates-formes pour organisateurs personnels comme *PalmOS*, que par le nombre de langages de programmation parlés, puisque des projections langage[4] ont été standardisées pour le C++, Java, Common Lisp, C, COBOL, et Python. Cette diversité de plate-forme d'exécution (au dessous de l'intergiciel, le « underware »), et de langage de programmation utilisé (au-dessus de l'intergiciel), conduit à une diversité dans les implantations CORBA. Au vu de cette diversité d'implantations, il est difficile de proposer une méthode de caractérisation qui soit applicable à toutes les implantations possibles. Nous nous focalisons dans cette thèse sur les plates-formes les plus fréquentes dans les systèmes répartis semi-critiques : les systèmes d'exploitation de type POSIX ou Win32, et les langages de programmation de type Java, C++ et C.

Comme le montre la figure 1.6 page suivante, une application répartie basée sur CORBA est constituée de plusieurs éléments :

- un objet client, à l'origine d'une requête. L'objet a obtenu la référence d'une interface (par exemple en faisant appel au service de désignation), et émet une requête vers cette interface.
- un *servant*, qui est la représentation concrète, dans un langage de programmation donné, d'un objet CORBA jouant un rôle serveur. Il implémente les opérations qui sont exposées dans l'interface de l'objet ; on dit qu'il *incarne* l'objet.
  Le cycle de vie du servant est découplé de celui du ou des objets CORBA qu'il incarne par l'intermédiaire de l'adaptateur d'objets.

[4] Dans la norme CORBA, une *projection langage*, ou *language mapping*, définit les aspects de l'interaction entre le niveau applicatif et l'intergiciel qui sont spécifiques à un langage de programmation donné.

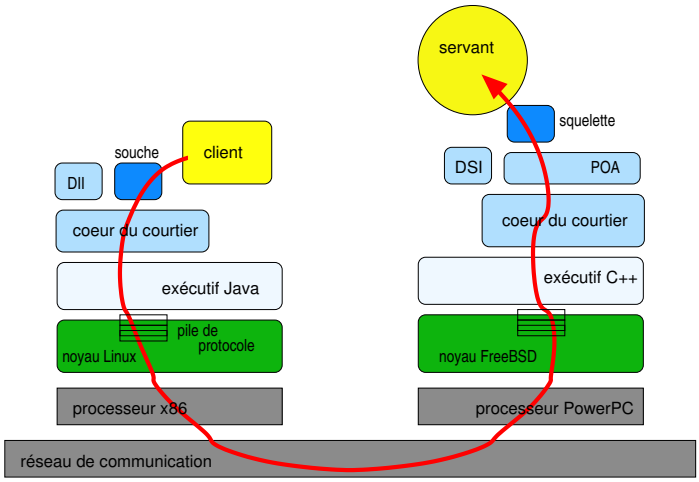

FIG. 1.6 – *Vue technologique d'un système basé sur* CORBA

- un *adaptateur d'objets* (*POA*, ou Portable Object Adapter) qui reçoit les requêtes et les envoie au servant approprié. Il est responsable de la gestion du cycle de vie des servants, et décide quand instancier et quand désallouer les servants. C'est l'adaptateur d'objets qui gère le modèle de concurrence utilisé dans le serveur, en fonction de politiques définies par l'utilisateur.

  L'adaptateur d'objets fournit la transparence vis-à-vis du modèle d'exécution du servant.

- une *souche*, qui a pour rôle de représenter localement une interface distante. Elle sert de mandataire local pour l'interface qu'elle représente ; c'est à elle que s'adresse l'objet client lorsqu'il émet une requête pour cette interface. Elle s'appuie sur les services du courtier côté client pour communiquer avec le servant distant qui incarne l'objet qu'elle représente. La souche est donc dépendante à la fois de l'interface IDL qu'elle représente, du langage de programmation utilisé par le client, de la projection langage et du courtier utilisé côté client.

  La souche fournit la transparence vis-à-vis du langage d'implantation d'un objet client.

- un *squelette*, qui joue le rôle d'intermédiaire entre l'adaptateur d'objets et le servant pour une interface donnée. Le squelette analyse la requête et appelle la méthode correspondante du servant avec les arguments déballés. Le squelette est dépendant à la fois de l'interface IDL qu'il représente, du langage de programmation utilisé pour implanter le servant, de la projection langage et du courtier utilisé côté serveur.

  Le squelette fournit la transparence vis-à-vis du langage d'implantation du servant.

- le *courtier* (ou *ORB*, pour *Object Request Broker* en anglais), qui joue le rôle de bus à requêtes. Un courtier est présent sur chacun des nœuds du système[5], et les implantations de différents fournisseurs doivent pouvoir interopérer. Côté client, le courtier reçoit des requêtes prédigérées par la souche. Il examine la référence de l'interface invoquée afin de déterminer sur quel nœud et quel port de communication s'exécute l'adaptateur d'objets qui héberge cette interface. Le courtier établit une connexion réseau avec l'adaptateur d'objets (ou réutilise une connexion existante), transmet la requête, attend la réponse, qu'il délivre à la souche. Côté serveur, le courtier fournit des services qui sont utilisés par l'adaptateur d'objets pour sa communication avec le squelette.

  Le courtier fournit la transparence vis-à-vis de la localisation des objets, et du protocole de transport employé pour les faire communiquer.
- le *système d'exploitation*, qui comprend les bibliothèques système (le plus la bibliothèque C) et le noyau, avec la pile de protocoles TCP/IP.
- de manière facultative, différents *supports d'exécution* pour les langages de programmation utilisés sur le système. Par exemple, les programmes Java s'exécutent généralement à l'aide d'une *JVM* (*Java Virtual Machine*, ou machine virtuelle Java), qui fournit une indépendance vis-à-vis de la plate-forme sous-jacente.
- les éléments matériels du nœud : processeur, sous-système mémoire, connexion au réseau d'interconnexion.

En complément de ces éléments incontournables, les plates-formes intergicielles fournissent généralement des services supplémentaires, qui facilitent le développement d'applications réparties. Les services les plus fréquemment présents sont les suivants :

- un service de désignation (ou *Naming Service*), qui fournit un service d'annuaire type *Pages Blanches* permettant d'associer un nom à une référence d'interface. Les servants peuvent enregistrer leur référence sous un nom symbolique, et les clients obtenir une référence d'interface en cherchant son nom dans l'annuaire.

  Les services de désignation diffèrent par leur niveau de sophistication : certains ne référencent que des noms simples, d'autres des noms hiérarchiques. Les services les plus sophistiqués, dits de *trading* ou *Pages Jaunes*, permettent des requêtes multicritères avec optimisation, de type « trouver la boulangerie la moins chère dans un rayon d'un kilomètre de la Place du Capitole ».
- un service de diffusion d'événements (ou *Event Service*). Ce service fournit un mécanisme d'interaction de type publication-souscription (*publish-subscribe*), qui permet de découpler les communications entre producteurs et consommateurs d'événements. Ce type de service est particulièrement utile dans des systèmes comme les bourses électroniques ou les systèmes de simulation.

[5] Notons la différence entre le point de vue traitement, où le courtier apparaît comme une seule entité distribuée sur chacun des nœuds, et le point de vue technologique, où l'on décompose le courtier en plusieurs entités qui coopèrent.

- un référentiel d'interfaces, qui fournit des méta-informations sur une description d'interface. Partant d'un nom d'interface, le référentiel permet de connaître la liste de ses opérations et leur signature (le nombre et le type des arguments et la liste des exceptions qui peuvent être levées). Le référentiel d'interfaces est surtout utilisé par le mécanisme d'invocation dynamique de CORBA (le *DII*, ou *Dynamic Invocation Interface*), dont nous ne parlerons pas dans cette thèse.

Les courtiers présents sur les différents nœuds communiquent entre eux à l'aide d'un protocole inter-courtier nommé *GIOP* (pour *Generalized Inter-ORB Protocol*). Ce protocole décrit comment les éléments d'une invocation de méthode doivent être encapsulés dans la couche de transport. Il est basé sur un ensemble de règles qui donnent pour chaque type de données spécifié par le langage OMG IDL, quelle doit être sa représentation lorsqu'il est mis à plat sous forme de séquence d'octets (processus d'« emballage », ou de *marshalling*). Cet ensemble de règles donne le format *Common Data Representation*, ou CDR.

Le protocole GIOP est indépendant du protocole de transport qui le prend en charge ; il présuppose juste que le protocole est orienté connexion, et qu'il est capable de transporter de manière fiable des séquences d'octets. TCP est le protocole de transport le plus communément associé à GIOP, un autre protocole appelé *IIOP* (pour *Internet Inter-ORB Protocol*) faisant le lien entre les deux[6].

On voit sur la figure 1.6 page 19 qu'une requête CORBA, qui paraissait simple du point de vue interaction, traverse en vérité de nombreux éléments ou couches de logiciel et de matériel : souche, noyau de courtier, le système d'exploitation et ses pilotes du matériel, le processeur et les unités d'entrée sortie, et le réseau d'interconnexion. Une faute peut survenir lors de la traversée de chacune de ces couches : soit sur le nœud où réside le client, soit sur le nœud supportant le servant, soit encore dans les matériels de télécommunication qui constituent le réseau.

Afin de caractériser de manière complète le comportement d'un intergiciel en présence de fautes, il sera donc nécessaire d'examiner l'impact de fautes survenant à chacun de ces niveaux.

Lors du développement, une implantation CORBA fournit plusieurs éléments pour permettre de générer l'application :

[6]La norme CORBA permet également aux implantations de négocier l'utilisation d'un protocole de session alternatif à GIOP, dans le cas où les courtiers client et serveur sont en accord. Ainsi, certains courtiers peuvent utiliser un transport par mémoire partagée de type SysV pour la communication entre deux objets qui s'exécutent sur le même nœud. Certains courtiers permettent à l'utilisateur de « plugger » des protocoles de transports spécifiques, comme un bus Myrinet. Ces fonctionnalités ne sont pas considérées dans notre étude.

- le *compilateur IDL*, qui génère le code pour les souches et les squelettes à partir des descriptions OMG IDL. Le compilateur IDL est spécifique au langage de programmation utilisé pour implanter les objets clients et les servants. Pour certains langages de programmation, il est également dépendant du courtier, puisqu'il appelle des fonctions fournies par l'implantation. Pour d'autres langages de programmation (notamment pour Java), la projection langage précise l'interface des méthodes qui peuvent être appelées par les souches et les squelettes ; la compilation IDL est alors indépendante du courtier sous-jacent.
- les fichiers en-têtes et bibliothèques fournis par le courtier. Souvent deux bibliothèques sont proposées, l'une pour le courtier en son rôle de client, et une autre pour le courtier en son rôle de serveur.

Les détails de ces aspects de l'intergiciel diffèrent d'une implantation à l'autre, comme nous le voyons dans la section suivante.

1.3.7 Stratégies d'implantation d'un intergiciel de communication

Les services CORBA tels que la désignation et le référentiel d'interfaces sont généralement implémentés sous la forme de *dæmons*. Un service peut s'exécuter sur un seul nœud, ou peut être fourni par plusieurs nœuds qui collaborent (c'est le cas de la fédération de serveurs de désignation, par exemple).

On a vu qu'un intergiciel de communication est composé de plusieurs éléments : le courtier, l'adaptateur d'objets, les souches et les squelettes. La mise en œuvre des souches et squelettes est fortement dépendante du langage de programmation utilisé pour incarner les objets CORBA. Concernant la mise en œuvre du courtier, trois principaux choix architecturaux sont possibles :

- **stratégie résidente noyau** : le courtier est implanté sous forme de service du système d'exploitation (par exemple, un serveur dans une architecture de type micronoyau). Cette stratégie permet la mise en œuvre d'un certain nombre d'optimisations de performance, puisque le système d'exploitation connaît la localisation de chacun des objets applicatifs. D'autre part, l'authentification des requêtes peut être basée sur les mécanismes d'authentification du noyau.

  Toutefois, le fort degré de dépendance entre l'intergiciel et le noyau imposé par cette approche la rend difficile à porter entre plates-formes, et fait qu'elle est peu utilisée dans la pratique. Le projet *kORBit*[7] est un exemple d'application de cette stratégie.

[7] L'étude *kORBit* a consisté à implémenter le courtier *ORBit* en tant que module du noyau Linux. *c.f.* `kORBit.sourceforge.net`.

- **stratégie orientée dæmon** : le service du courtier est fourni par un ou plusieurs processus dæmon, qui servent d'intermédiaires entre les clients et les servants. Par exemple, chaque nœud peut exécuter un dæmon d'activation, qui est responsable de l'activation des servants et des communications avec les autres nœuds. Le dæmon reçoit toutes les requêtes entrantes vers ce nœud, et les diffuse vers le servant approprié (éventuellement après l'avoir activé). Symétriquement, le dæmon reçoit toutes les requêtes sortantes et les distribue vers le nœud approprié. Cette stratégie d'implantation facilite l'administration centralisée, puisque tous les processus CORBA sont connus du dæmon d'activation. Elle peut également faciliter la mise en œuvre de techniques d'équilibrage de charge ou de tolérance aux fautes par réplication, puisque le dæmon d'activation centralise le routage de toutes les requêtes.
- **stratégie résidente dans l'application** : le code qui implémente les services courtier s'exécute dans le même contexte que le client et que les servants. Le courtier peut être livré sous forme de bibliothèque partagée, a lier aux applications CORBA. Parfois cette bibliothèque est décomposée en deux parties, une étant utilisée par les objets jouant uniquement un rôle de client, et l'autre par les objets qui jouent un rôle de serveur. Cette stratégie est la plus fréquemment utilisée dans les systèmes de type POSIX.

  En réalité, même dans les deux stratégies précédentes, une certaine partie de la fonctionnalité courtier réside dans chaque processus CORBAïfié, afin de gérer la projection langage.

La frontière entre ces choix d'implantation n'est pas rigide ; il est possible de combiner deux approches. Par exemple, l'implantation commerciale *Orbix* utilise une stratégie résidente dans l'application, mais transfère une part du travail de l'adaptateur d'objets vers un dæmon appelé le *orbixd*. Une instance de ce dæmon est présente sur chaque nœud où s'exécutent des servants. Le orbixd est responsable de l'activation des servants, en fonction des demandes des clients et de politiques d'activation choisies par l'administrateur. Il s'appuie sur une base de données appelée *Implementation Repository*, qui associe des noms d'interface aux paramètres nécessaires au démarrage d'un processus qui incarne cette interface (généralement une image exécutable et des paramètres de ligne de commande). Lorsqu'une requête arrive sollicitant une interface pour laquelle il n'existe pas de servant disponible, le *orbixd* démarre un processus pour servir la requête. Avec ce choix d'implantation, une partie de la responsabilité d'administration des processus serveur est centralisée sur chaque nœud (constituant ainsi un point dur).

Ces choix d'implantation ne sont pas sans incidence sur la performance et la facilité d'utilisation du système. Ils ont également un impact sur sa sûreté de fonctionnement, puisque chaque stratégie amène différents canaux de propagation d'erreur.

1.3.8 Propagation d'erreur dans un système à base d'intergiciel

Lorsqu'une erreur qui survient dans un sous-système entraîne une corruption dans un autre sous-système, on parle de *propagation d'erreur*. Le support pour cette propagation est dit *canal de propagation d'erreur*, et peut être un élément physique partagé par les deux sous-systèmes, ou une interaction entre eux. Le domaine de la tolérance aux fautes consiste à trouver des mécanismes de :

- *détection d'erreur*, qui permettent d'identifier le plus vite possible un état erroné. Un checksum intégré à un message est un exemple de mécanisme de détection d'erreur : il permet au récepteur de détecter que le message a été corrompu au cours de la transmission.
- *recouvrement d'erreur*, qui consiste à ramener le système vers un état sûr et sans erreur. Le fait de demander la retransmission d'un message corrompu est une stratégie de recouvrement d'erreur.
- *confinement d'erreur*, qui consistent à empêcher la propagation d'une erreur depuis le sous-système où elle s'est produite.

Considérons les différentes stratégies architecturales, que nous avons présentées dans la section précédente, selon deux points de vue :

- quels sont les canaux de propagation d'erreur induits par l'architecture, pour des erreurs qui se propagent entre l'intergiciel et l'application, ou entre l'intergiciel et le système d'exploitation.
- dans un contexte d'évaluation du COTS, quels sont les problèmes posés par l'architecture en ce qui concerne l'« analysabilité » du composant (détection de zones de code mort, de fonctionnalités cachées). D'autre part, quelle est la facilité pour mettre en œuvre des mécanismes d'encapsulation pour améliorer le confinement et augmenter la robustesse du COTS, ou pour limiter l'accès aux fonctionnalités non utilisées de ce composant.

La stratégie résidante noyau protège le courtier de programmes applicatifs défaillants : le courtier s'exécute dans un contexte privilégié qui n'est pas accessible aux applications. Cependant, le service courtier constitue un canal de propagation d'erreur pour tous les objets localisés sur ce nœud. Dans un contexte d'évaluation de logiciel COTS, le fait de s'exécuter à un niveau de privilège élevé pose des problèmes quant aux possibilités d'analyse de l'intergiciel.

La stratégie orientée dæmon introduit un point de défaillance unique par nœud, puisque la défaillance du dæmon d'activation touche tous les objets CORBA localisés sur ce nœud. Toutefois, le fait de séparer une partie de la fonctionnalité de l'intergiciel dans un processus dédié présente plusieurs intérêts : grâce à la protection des espaces d'adressage, les possibilités de propagation d'erreur du dæmon vers les applications sont réduites, et l'observabilité et la contrôlabilité du processus sont améliorés.

La stratégie résidente dans l'application autorise un niveau de confinement d'erreur variant considérablement en fonction de la correspondance choisie entre objets CORBA et entités fournies par le système d'exploitation sous-jacent (processus, processus léger). La norme CORBA est volontairement muette à ce sujet, préférant laisser au concepteur la liberté de choisir la représentation qui convient le mieux à ses objectifs. Trois grands choix sont envisageables :

- L'utilisation d'un nœud dédié à chaque servant. Dans ce cas, le seul canal de propagation d'erreur entre un objet client et un servant est constitué par le réseau d'interconnexion (ainsi qu'une défaillance de mode commun dans les couches logicielles ou matérielles sous-jacentes). Cependant, ce mode de déploiement est très coûteux en terme de ressources matérielles et peu adapté à un système comprenant un nombre important d'objets.
- Chaque servant se voit affecté à un processus dédié, avec plusieurs servants hébergés sur le même nœud. Cette solution fournit un bon niveau de confinement d'erreur, puisque la défaillance d'un objet n'entraîne pas mécaniquement la défaillance d'autres objets s'exécutant sur le même nœud. Cependant, elle est coûteuse en terme d'utilisation des ressources système, en particulier lorsque le système comprend un grand nombre d'objets relativement légers.
- Plusieurs objets CORBA partagent le même processus système (avec éventuellement une concurrence provenant de l'utilisation de processus légers). Cette technique implique que ces objets partagent le même espace d'adressage (et par conséquent la même zone de confinement d'erreur), la défaillance d'un objet pouvant alors provoquer la défaillance des autres. Ce choix de déploiement fournit donc le plus faible niveau de confinement d'erreur.

  Lorsque l'objet client et le servant partagent le même espace d'adressage, on parle de *colocation*. Le courtier peut alors optimiser les requêtes entre client et serveur en les transformant par une invocation de méthode locale, court-circuitant ainsi la phase d'emballage des arguments et le passage par la pile de protocole réseau.

  Lorsque plusieurs servants sont hébergés dans un même processus serveur, il existe plusieurs modèles d'exécution pour gérer les invocations concurrentes sur les servants. Le courtier peut utiliser un seul contexte d'exécution, en dispatchant les requêtes entrantes vers l'objet appelé via un mécanisme comme `select()` sous POSIX. Il peut aussi créer pour chaque nouvelle requête un processus léger, qu'il détruit à la fin du traitement. Il peut également réutiliser des processus légers provenant d'un pool pré-alloué.

Dans ce qui suit, nous considérons généralement la seconde stratégie, c'est-à-dire un processus par servant. En effet, il s'agit du meilleur compromis entre confinement d'erreur et efficacité, sur les types des plates-formes que nous avons ciblées.

1.3.9 Types d'intergiciels visés

Notre approche de caractérisation peut s'appliquer à différentes catégories d'intergiciels orientés communication :

- des logiciels conformes à la plate-forme CORBA, tels que les implantations *TAO*, *ORBacus*, *omniORB* et les courtiers intégrés aux machines virtuelles Java. Nous pouvons caractériser la fonctionnalité de courtier fournie par une de ces implantations, ainsi que ses services (tels que la désignation et la diffusion d'événements). Dans une moindre mesure, il peut être intéressant d'étudier la robustesse de leurs outils de développement, en particulier le compilateur IDL et le référentiel d'interfaces.
- la plate-forme DCOM de Microsoft. Cet intergiciel de communication supporte la communication à base d'invocation de méthode entre objets répartis hétérogènes. Il s'appuie sur la définition d'un format binaire pour les interfaces des objets, inspiré du mécanisme de *vtables* utilisé par les compilateurs C++ pour la résolution de la liaison dynamique.
- les fonctionnalités orientées communication des exécutifs de certains langages de programmation :
  - le support RMI (*Remote Method Invocation*, mécanisme d'invocation de méthode à distance) fourni par les machines virtuelles Java. Ce mécanisme ne supporte par l'hétérogénéité, puisque seuls des objets implantés en langage Java peuvent interagir. Par contre, il permet l'interopérabilité entre des objets Java s'exécutant sur des machines virtuelles de fournisseurs différents.
  - la *CLR* (*Common Language Runtime*, ou support d'exécution commun) de la plate-forme *.NET* de Microsoft. Le mécanisme *.NET Remoting* autorise les appels de procédure à distance, avec divers formats d'emballage et mécanismes de transport.

L'essentiel du travail expérimental réalisé pendant cette thèse s'est centré sur la plate-forme CORBA, puisqu'il s'agit de la famille d'intergiciels de communication la plus mûre et la plus utilisée dans les domaines d'application qui nous intéressent. Nous discutons dans le chapitre 5 de l'extension de nos travaux à d'autres familles d'intergiciel de communication.

1.4 Techniques de caractérisation de la Sûreté de Fonctionnement

Il existe deux grandes catégories de techniques pour l'évaluation de la sûreté de fonctionnement de systèmes informatiques : les techniques à base de modèles, et les techniques à base de mesures. La modélisation permet aux concepteurs d'obtenir des prévisions sur les attributs de sûreté de fonctionnement d'un système en se basant sur des mesures probabilistes du comportement de ses sous-systèmes. Ces résultats

sont utiles pendant les phases de conception du système, puisqu'elles permettent l'évaluation de mesures pertinentes de la sûreté de fonctionnement avant même que le système ne soit construit.

Cependant, les techniques de modélisation ne peuvent que prédire la sûreté de fonctionnement d'un système. Une fois qu'il a été développé et déployé, les techniques basées sur les mesures peuvent être employées afin d'obtenir des informations plus précises. Il existe deux familles de techniques orientées mesure permettant d'obtenir des informations sur la sûreté de fonctionnement d'un système : l'observation de systèmes opérationnels, et l'injection de faute.

1.4.1 Observation de systèmes opérationnels

Cette approche de caractérisation consiste à observer le comportement d'un grand nombre de systèmes au cours de leur vie opérationnelle. Elle est basée sur l'analyse des fichiers de journalisation maintenus automatiquement par les systèmes, ou sur des informations enregistrées par les administrateurs système. À partir de ces données, il est possible d'obtenir des informations sur la nature et la fréquence des défaillances subies en opération, et sur les types d'utilisation qui ont entraîné la défaillance du système. L'analyse des relevés de défaillance fournit des informations précieuses pour guider l'élaboration d'un modèle de fautes.

L'un des inconvénients de cette approche provient de la rareté des défaillances, qui oblige à recueillir des informations sur une population étendue de systèmes ayant des configurations identiques, pendant une longue période, avant de pouvoir faire des analyses statistiques. Cette approche est donc mal adaptée aux cycles de développement courts, comme ceux des produits intergiciel.

À notre connaissance, il n'y a pas eu de travaux traitant de l'observation sur le terrain des défaillances de systèmes basés sur intergiciel. Cela rend difficile toute validation du degré de représentativité d'un modèle de fautes donné pour ces cibles. Le modèle de fautes que nous utilisons pour les fautes physiques (*c.f.* le chapitre 2) est issu du travail considérable qui a été mené pour caractériser les fautes physiques affectant les processeurs et les mémoires, ainsi que de travaux plus récents sur les fautes dans les réseaux de télécommunication. En ce qui concerne les fautes du logiciel, notre modèle de fautes est inspiré de l'analyse structurelle de l'architecture de systèmes à base d'intergiciel de communication, que nous avons présenté au § 1.3.

1.4.2 L'injection de fautes

L'injection de fautes est une technique de caractérisation de la sûreté de fonctionnement bien connue [Arlat et al. 1993], qui consiste à étudier les réactions d'un système face à des conditions anormales : par l'insertion délibérée de fautes dans le système cible, on cherche à accélérer la caractérisation de son comportement en présence de fautes. Cette approche est complémentaire aux approches analytiques, et permet

d'examiner des états du système qui ne seraient pas atteints par le test fonctionnel conventionnel. L'objectif des expériences d'injection de fautes est de simuler l'effet de fautes réelles qui pourraient affecter le système cible, c'est-à-dire d'étudier l'impact des erreurs qui sont dues à l'activation de la faute.

Dans le contexte général de la sûreté de fonctionnement, l'injection de fautes se positionne comme une technique au service de l'élimination des fautes et de la prévision des fautes. Concernant l'élimination des fautes, l'injection de fautes peut être vue comme un moyen de tester les mécanismes de tolérance aux fautes vis-à-vis d'entrées pour lesquelles ils ont été conçus : les fautes. Sur le plan de la prévision des fautes, l'injection de fautes constitue une forme d'évaluation expérimentale permettant d'estimer des paramètres tels que le taux de couverture des mécanismes de tolérance aux fautes ou la latence de détection d'erreur, et ainsi d'évaluer l'efficacité des mécanismes de tolérance aux fautes.

Les campagnes d'injection de fautes fournissent plusieurs types de résultat :

- elles permettent de comprendre les modes de défaillance du système, et d'observer son comportement en présence de faute ;
- elles fournissent des informations sur les mécanismes de tolérance aux fautes (éventuellement) présents dans le système ciblé, et en particulier permettent de mesurer leur facteur de couverture (la probabilité conditionnelle que, pour une faute donnée dans le système, le système pourra la tolérer).

Les premiers travaux sur l'injection de fautes visaient à simuler l'effet de fautes physiques résultant de la radiation électromagnétique [Karlsson et al. 1998]. Les techniques utilisées consistaient à irradier les circuits électroniques de ions lourds, ou à agir directement sur les broches des microprocesseurs en modifiant les tensions.

Avec la montée en complexité et en vitesse des circuits intégrés modernes, ces techniques sont devenues difficiles à appliquer et à cibler de manière précise. La majorité des travaux récents se sont focalisés sur l'injection de fautes par logiciel [Kanawati et al. 1992]. Cette technique, dite d'*émulation de fautes*, consiste à effectuer la corruption de l'état du système par des mécanismes logiciels. Elle permet de cibler précisément l'adresse mémoire à corrompre et de synchroniser l'instant où la faute est injectée avec l'activité de la charge applicative. L'injection de fautes par logiciel permet donc de cibler différentes couches dans le système (le noyau du système d'exploitation, les services système, l'intergiciel, le code applicatif), et de simuler des fautes dans la mémoire principale, les registres du microprocesseur, ou le bus de communication. Elle offre une grande souplesse concernant le choix du modèle de fautes et le mode de déclenchement de l'injection.

Concernant la représentativité des fautes, des études ont démontré que l'utilisation d'inversions isolées de bit entraîne des erreurs semblables à celles résultant de techniques d'injection physiques (*e.g.* [Rimén et al. 1994, Fuchs 1998]) et permet de simuler les erreurs provoquées par les fautes logicielles avec une bonne fidélité [Madeira et al. 2000].

1.5 État de l'art en caractérisation d'intergiciel

Dans cette section, nous décrivons l'état de l'art en ce qui concerne l'utilisation de techniques d'injection de fautes pour la caractérisation de la sûreté de fonctionnement de systèmes informatiques. Nous nous focalisons sur des recherches qui ont ciblé l'intergiciel, mais considérons également d'autres couches exécutives comme les noyaux de système d'exploitation et les piles de protocole de communication. Nous classifions les techniques d'injection de fautes suivant le type de fautes qu'elles permettent de simuler ; des fautes d'origines internes à un composant, ou des fautes atteignent le composant via ses interactions avec le reste du système.

1.5.1 Fautes d'origine interne

Une faute est dite *interne* lorsque la corruption de l'état du composant qu'elle engendre a son origine dans ce composant. Ceci comprend les erreurs de programmation qui peuvent affecter les données internes du système, et les fautes matérielles qui peuvent corrompre la mémoire du système ou une unité d'entrées/sorties. Le modèle de fautes le plus employé pour simuler ces fautes est l'inversion d'un bit.

Un nombre important d'outils ont été développés afin d'automatiser le déroulement de campagnes d'injection de fautes utilisant ce modèle de fautes. Deux exemples significatifs sont *Xception*, développé à l'Université de Coimbra au Portugal [Carreira et al. 1998], et *MAFALDA*, développé au LAAS-CNRS, Toulouse, France [Arlat et al. 2002]. Ces outils fournissent en général un banc de test dans lequel on exécute son système, un certain nombre d'injecteurs permettant de travailler avec différents modèles de faute, un système pour sauvegarder le résultat des expériences, et des outils d'analyse des résultats. Ces outils ont été appliqués à une variété considérable de systèmes cibles :

- les cartes contrôleurs pour des bus de communication : citons par exemple le projet *FIT* qui a effectué des expériences d'injection de fautes dans des implémentations du protocole TDMA *TTP/C* ;
- des micronoyaux du commerce [Arlat et al. 2002] ;
- des systèmes de gestion de bases de données du commerce [Costa et al. 2000] ;
- des applications embarquées sur des satellites (Mars Pathfinder).

Toutefois, il existe très peu d'exemples d'application de ce type d'outil à des intergiciels de communication. Les deux études les plus proches de notre problématique sont [Chevochot & Puaut 2001], qui cible un support d'exécution distribué pour des applications temps-réel ; et [Chung et al. 1999], qui compare l'impact de fautes telles que l'arrêt de processus et de processus légers sur des applications CORBA et DCOM.

Les travaux de Chevochaut et Puaut concernent des expériences d'injection de fautes par logiciel visant à estimer la couverture de l'hypothèse de silence sur défaillance d'un

support d'exécution distribué à base de micronoyaux COTS temps-réel. La cible des expériences est un intergiciel spécifique pour la tolérance aux fautes, qui s'exécute sur le micronoyau temps-réel Chorus. Le système est constitué d'un ensemble de machines à base de processeur Pentium, communicant sur un bus ATM et sur un bus Ethernet. L'intergiciel implémente des services de communication de groupe, de synchronisation d'horloge et d'ordonnancement réparti. Les fautes injectées sont des inversions de bits dans la mémoire du micronoyau et de l'intergiciel.

Le système cible ne comporte aucun mécanisme matériel spécifique de détection d'erreur, en dehors de ceux implémentés par le processeur. Le système peut être déployé en présence de divers mécanismes logiciels de détection d'erreur, basés sur la redondance (*checksums* sur les messages échangés entre nœuds, par exemple), ou des assertions comportementales (telles que la comparaison du flot de contrôle avec un graphe d'appel calculé statiquement). Les auteurs ont mesuré un facteur de couverture de l'hypothèse de silence sur défaillance variant entre 80,6% (dans une configuration sans aucun mécanisme logiciel de détection d'erreur) et 99,1% (avec tous les mécanismes logiciels de détection d'erreur activés).

Cette étude montre que les fautes affectant l'intergiciel de communication sont bien plus graves – vis-à-vis de la propriété de silence sur défaillance – que les fautes affectant le micronoyau, car toutes les fautes qui ont provoqué une violation de l'hypothèse de silence sur défaillance ont été injectées dans une zone mémoire correspondant à l'intergiciel, et la majorité d'entre elles ont affecté le protocole de multicast fiable. Ce résultat souligne l'importance de caractériser le comportement d'un intergiciel de communication en présence de fautes.

Le mode de défaillance le plus fréquemment observé a été l'arrêt inopiné de nœuds, sans production de valeurs erronées au niveau applicatif ni de propagations d'erreurs vers les autres nœuds. Le mécanisme de détection d'erreur le plus efficace (mesuré par le taux de « première détection », c'est-à-dire le mécanisme qui a été le premier à détecter une erreur lors d'une expérience), est le mécanisme de protection d'espaces d'adressage fourni par le micronoyau, avec plus de 50% de premières détections.

La seconde étude proche de notre problématique est [Chung et al. 1999], qui rapporte des expériences d'injection de fautes ciblant des applications CORBA (plus précisément l'implémentation *Orbix*) et DCOM, sous le système d'exploitation Windows NT. Les fautes simulées sont le gel et l'arrêt inopiné de processus, de processus légers et de nœuds de calcul.

Les auteurs ont constaté que les modes de défaillance des applications DCOM diffèrent suivant que le client et le serveur s'exécutent sur le même nœud, ou sur des nœuds différents : des codes erreurs différents sont levés au niveau applicatif dans les deux cas. Cela implique que la transparence vis-à-vis de la localisation fournie par cet intergiciel de communication disparaît en présence de fautes. Pour les applications CORBA, le mode de signalement des défaillances est le même dans les deux cas, avec toutefois une détection plus rapide lorsque les deux objets s'exécutent sur le même nœud.

Pour les deux intergiciels de communication, les auteurs ont observé que la latence de détection de certaines défaillances (par exemple la perte de la connexion réseau sur un nœud) est très importante, allant jusqu'à 8 minutes dans certains cas. Ce délai est dû à ce que l'intergiciel tente un certain nombre de retransmissions lorsqu'une requête échoue, avant de rapporter l'erreur au niveau applicatif, et que le protocole Ethernet utilise un système de binary-backoff[8] qui peut être lent à détecter les défaillances[9]. Ces observations ont conduit les auteurs à suggérer la mise en place de mécanismes de chien de garde au niveau applicatif, pour détecter les défaillances de manière « hors-bande ».

Nous ne connaissons pas d'autres travaux portant sur des cibles CORBA qui simulent l'effet d'autres types de fautes internes, telles que des corruptions de la mémoire. Cependant, cette technique a été utilisée intensivement pour la caractérisation d'autres classes d'exécutifs, telles que les systèmes d'exploitation et les exécutifs langages.

1.5.2 Fautes d'origine externe

Les fautes d'origine externe sont caractérisées par le fait que leur origine est extérieure au système cible. La simulation de fautes externes permet d'évaluer la *robustesse* du composant cible, c'est-à-dire sa capacité à délivrer un service acceptable en présence d'entrées invalides et de conditions environnementales stressantes.

Test de robustesse

Le test de robustesse permet d'évaluer la probabilité de propagation d'erreur entre les composants du système, ou entre les différents nœuds d'un système réparti. Dans [Miller et al. 1990], la robustesse de différentes implantations des outils standards *Unix* a été mesurée en leur appliquant un flux d'entrée généré aléatoirement. Dans ces expériences, l'interface des outils est constituée des options de la ligne de commande, ainsi que l'entrée standard. Malgré l'utilisation d'une classification extrêmement simple des modes de défaillance (la seule défaillance considérée étant l'arrêt inopiné de l'outil), cette technique dite de *fuzz testing* a montré qu'une proportion élevée des implantations de ces outils avait un taux de défaillance inacceptable.

Un autre exemple connu de test de robustesse est le projet Ballista [Koopman & DeVale 1999], qui a étudié plusieurs systèmes d'exploitation conformes à la norme POSIX. Leurs expériences consistent à utiliser des paramètres

[8] Le mécanisme de retard binaire sur reprise (ou *binary exponential backoff*) est employé sur les canaux de communication à accès multiple. En cas de collision entre deux émetteurs, un temps d'attente « backoff » est calculé par tirage aléatoire dans une plage qui croît exponentiellement avec le nombre de collision.

[9] Notons toutefois que les versions plus récentes de la norme CORBA intègrent un volet *Messaging*, qui permet de contrôler des aspects liés à la qualité de service des interactions entre objets. En particulier, il est possible de configurer un courtier – à condition qu'il implémente cette partie de la norme – pour qu'une exception `TIMEOUT` soit levée si une requête met plus qu'une durée prédéterminée pour être traitée.

invalides dans les appels système, ou à exécuter des séquences d'appels système incorrectes. Un exemple de paramètre invalide est un pointeur nul lorsqu'un appel système attend un pointeur de fichier, ou encore d'utiliser un nombre négatif pour un paramètre qui représente un descripteur de fichier. Cette technique d'injection de fautes nécessite le développement d'un ensemble de classes de paramètres invalides, qui permet de générer pour chaque type de paramètre des jeux de test utilisant différentes variantes invalides.

Les auteurs de ces études ont observé une proportion importante de comportements non robustes sur les systèmes d'exploitation ciblés (par exemple, entre 18 et 34% des tests lancés ont provoqué un mode de défaillance par arrêt inopiné du noyau).

L'approche Ballista a également été appliquée au test de robustesse d'un certain nombre d'implantations CORBA [Pan et al. 2001], en générant des invocations erronées d'une partie de l'interface côté client proposée par un courtier. Par exemple, l'opération `object_to_string`, qui renvoie une représentation textuelle d'une référence CORBA, peut être invoquée avec une référence invalide. On observe alors le comportement du courtier, qui peut lever une exception CORBA, s'arrêter brutalement, ou se geler. Les expériences ont ciblé trois implantations de différents fournisseurs, en testant différentes versions des courtiers sur plusieurs systèmes d'exploitation (*c.f.* la figure 1.7). Les résultats révèlent une proportion élevée de comportements non robustes, tels que l'arrêt inopiné et le gel de processus légers.

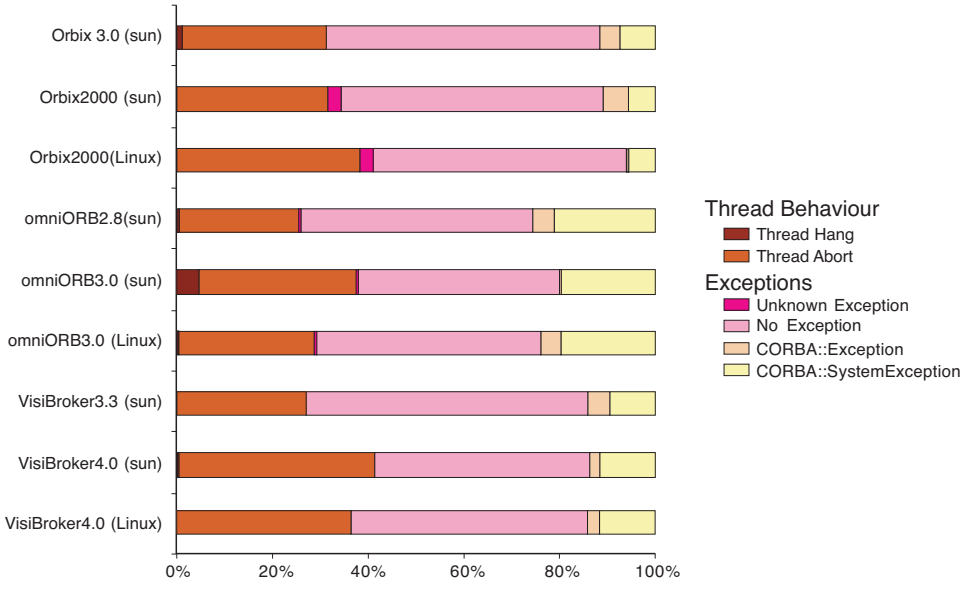

FIG. 1.7 – *Projet Ballista : comparaison de la robustesse de courtiers* CORBA

Il est important de souligner que ces travaux n'ont porté que sur des interactions clientes avec le courtier. En particulier, les activités impliquant une interaction entre un courtier client et un courtier qui joue le rôle serveur ne sont pas concernées par ces tests. Précisons également que la fonctionnalité fournie par l'interface côté client d'un courtier ne joue qu'un rôle mineur dans la vie d'un système basé sur CORBA. En effet, les opérations dans cette interface sont utilisées principalement lors de l'initialisation de l'application, pour des tâches telles que la prise en compte d'options de configuration passées sur la ligne de commande lors du lancement de l'application ou l'initialisation du mode d'ordonnancement employé par le courtier.

En fait, une grande partie de la fonctionnalité proposée par un courtier est fournie de manière implicite, dans le sens où elle est activée sans que l'application y fasse appel explicitement (nous revenons sur ce point au § 1.6). Il est donc difficile de cibler les parties du courtier qui implémentent cette fonctionnalité en utilisant l'approche Ballista.

À notre connaissance, les travaux effectués dans le cadre du projet *Ballista* sont la seule étude d'injection de fautes sur des implantations de la norme CORBA.

Test orienté protocole

L'outil ORCHESTRA [Dawson & Jahanian 1995] développé à l'Université du Michigan permet de valider des propriétés de sûreté de fonctionnement des applications réparties et des protocoles de communication. Le principe de cet outil est de placer, sur chaque nœud du système, une couche d'injection de fautes entre deux couches de la pile de protocole. Des fautes telles que le retardement, la perte, la duplication, et la corruption de messages peuvent ainsi être insérées au cours de l'exécution du système. Cet outil a notamment été appliqué à la caractérisation du comportement en présence de fautes des piles TCP de quatre implantations Unix commerciales.

Le projet *PROTOS* du *Secure Programming Group* de l'Université de Oulu, Finlande, cherche à tester des implémentations de protocoles de communication, vis-à-vis de fautes malicieuses [Kaksonen et al. 2001]. Leur méthodologie de *test de vulnérabilité* s'apparente à du test de robustesse : le système cible est soumis à un ensemble d'entrées invalides, afin de déterminer s'il est vulnérable à ce type d'entrées[10]. La particularité de ce travail est de chercher à identifier des séquences d'entrées qui pourraient être exploitées par un utilisateur malveillant afin d'obtenir des privilèges auxquels il ne devrait pas avoir droit, ou de perturber le système afin de gêner ses autres utilisateurs.

Leur méthode consiste à *(i)* identifier l'ensemble des interfaces par lesquelles le système interagit avec son environnement (en particulier celles qui sont accessibles à un utilisateur distant, par l'intermédiaire du réseau de communication) ; *(ii)* développer

[10]Une *vulnérabilité* est une faute de conception, intentionnelle ou accidentelle, qui favorise la réalisation d'une menace ou la réussite d'une attaque.

une spécification des protocoles utilisés pour interagir avec cette interface (généralement sous la forme d'une grammaire des séquences d'entrée autorisées) ; *(iii)* exécuter des tests extraits de cette grammaire ; *(iv)* analyser les traces du comportement du système afin d'identifier les scénari à risque.

L'extraction de tests à partir de la spécification des interactions autorisées est faite manuellement, par mutation des séquences de messages ou de leurs contenus. Cette approche nécessite donc un travail important (comparé à des approches d'injection de fautes aléatoires), mais les auteurs estiment que des entrées ciblées seront plus à même de révéler des fautes dans le système cible.

Des expériences ont été menées sur plusieurs protocoles dont les implémentations sont largement déployées en industrie : le SNMP (*Simple Network Monitoring Protocol*), utilisé pour le contrôle et la supervision d'équipements réseau ; le LDAP (*Lightweight Directory Access Protocol*), utilisé pour les annuaires d'entreprise ; le WAP (*Wireless Access Protocol*), utilisé pour permettre à des terminaux mobiles, tels que les téléphones portables, d'accéder à des services via le réseau. Les résultats de ces travaux sont probants, puisqu'ils ont permis l'identification de nombreuses vulnérabilités, en particulier un défaut important dans un grand nombre d'implantations SNMP [CERT 2002b].

Test de conformité d'implantations CORBA

Les autres travaux sur la validation de systèmes basés sur CORBA ont employé une approche de test de conformité. Le *test de conformité* consiste à évaluer la correction fonctionnelle d'une implémentation d'un système vis-à-vis d'une spécification de référence. Ce type de test considère généralement le système comme boîte noire. Dans le cas d'implantations de la norme CORBA, la spécification de référence qu'il convient d'utiliser est le *Product Profile* défini par le *Product Standard Definition Subcommittee* de l'OMG pour chaque version de la norme CORBA (les normes elles-mêmes ne sont pas suffisamment précises pour servir de référence à un test de conformité).

Le *Product Profile* pour CORBA 2.3 contraint les points suivants d'une implantation CORBA (seules des implantations C++ ou Java sont traitées par ce document) :

1. la syntaxe des constructions IDL et leur projection dans le langage de programmation cible ;
2. la sémantique des constructions IDL et leur projection dans le langage de programmation cible ;
3. la syntaxe des interfaces IDL spécifiées par la norme CORBA et leur projection dans le langage de programmation cible ;
4. la sémantique des interfaces IDL spécifiées par la norme CORBA et leur projection dans le langage de programmation cible ;
5. la manière d'implémenter les protocoles GIOP et IIOP.

Les implantations qui réussissent ce test de conformité sont autorisées à utiliser la marque CORBA de l'*Open Group*. Il faut reconnaître que ce processus de « labelisation » n'a pas bien fonctionné dans la pratique, puisque seules trois implantations se sont soumises à l'exercice. Constatant cet échec, l'OMG a ensuite soutenu le projet de recherche *CORVAL*, dont l'objectif était de tester les aspects fonctionnels des courtiers, ainsi que leur interopérabilité. Le projet (puis son successeur *CORVAL2*[11]) a développé une suite de tests de conformité pour la norme CORBA 2.3.1, qui couvre les éléments suivants d'un courtier :

- des tests ciblant le traitement du langage OMG IDL, afin de vérifier que le code généré par le compilateur IDL est correct d'un point de vue syntaxique. Le test de validité consiste à vérifier que toutes les opérations attendues sont présentes dans le code généré, avec la bonne signature. Ces tests ont été effectués pour les projections C++ et Java ;
- des tests sur les souches et les squelettes, permettant de vérifier que le comportement à l'exécution du code généré par le compilateur IDL est conforme au comportement attendu. Ces tests invoquent les opérations d'une interface OMG IDL avec des paramètres du type adéquat, et vérifient qu'ils sont bien transmis au récepteur. Notons que ceci revient à du test de compilateur, et que ce type de test se heurte rapidement à des problèmes d'explosion combinatoire.
- des tests de comportement ciblant l'API CORBA, afin de vérifier la conformité des API fournis avec l'implantation ;
- des tests GIOP et IIOP, permettant de vérifier la syntaxe des messages GIOP et leurs échanges par IIOP. Les tests effectués se limitent à un test d'interopérabilité entre courtiers.

Notons que l'approche de validation mise en œuvre dans ces travaux est purement fonctionnelle ; les tests développés ne cherchent pas à caractériser le comportement des intergiciels en présence de fautes. D'autre part, les points de conformité listés ci-dessus ne couvrent que partiellement la surface fonctionnelle d'un courtier, comme nous le montrons au § 1.6.

1.6 Obstacles à la caractérisation d'intergiciel

Comme nous l'avons décrit dans la section 1.5, il existe de nombreux travaux sur la caractérisation de la sûreté de fonctionnement de couches exécutives, en particulier sur le sujet des micronoyaux et les systèmes d'exploitation. Cependant, il existe très peu de travaux concernant l'intergiciel. Cela s'explique en partie par leur apparition relativement récente, en tant que technologie logicielle, mais également par un certain nombre de problèmes qui doivent être résolus avant de pouvoir caractériser un intergiciel.

[11] `http://www.opengroup.org/corval2/`

Les travaux concernant les couches exécutives plus basses ne peuvent pas être transposés directement dans le contexte de l'intergiciel. En effet, cette couche logicielle possède un certain nombre de caractéristiques qui lui sont propres, et qui nécessitent le développement de nouvelles techniques de caractérisation :

- l'intergiciel est implanté de manière répartie : il se base sur des mécanismes de communication à distance afin de fournir des services à la couche applicative. L'utilisation de ces mécanismes expose l'intergiciel à des fautes liées à la communication, qui ont un plus faible impact sur les couches exécutives basses. Par conséquence, ces fautes ont été peu étudiées par les travaux précédents de caractérisation d'exécutifs, qui ont surtout ciblé ces couches exécutives basses. Nous présentons le modèle de fautes que nous avons utilisé dans nos expériences au § 2.1.2.
- le service fourni par un intergiciel orienté communication est réparti ; il n'est pas possible de juger à partir d'un seul nœud si l'intergiciel fonctionne correctement. Dans un contexte de validation, il faut donc un oracle[12] distribué pour détecter les défaillances.

  La détection de défaillance dans un système centralisé est relativement facile, puisque le système d'exploitation connaît l'état et peut tracer les interactions de tous les processus et de tous les éléments matériels. Lorsqu'une ressource devient indisponible, le système d'exploitation peut rapidement informer l'application du problème et arrêter un processus qui se comporte de manière imprévue. Par contre, dans un système réparti, aucun élément n'a une vision globale du système, et il est plus difficile de détecter les erreurs, surtout lorsque le temps est critique ; le développement des détecteurs de défaillance est donc plus délicat dans ce contexte.
- un intergiciel de communication ne fournit pas un service à proprement parler, mais est plutôt chargé de *maintenir une illusion*, permettant au programmeur applicatif de considérer que tous les objets sont locaux, ou que la distance entre les nœuds composant le système est nulle.

  Pour maintenir cette illusion de transparence vis-à-vis de la répartition, l'intergiciel est contraint de s'immiscer dans le code applicatif, afin de pouvoir intercepter les appels vers les objets pour lesquels il joue un rôle de mandataire. Sa technique d'interception, et les mécanismes qu'il met en œuvre pour maintenir l'illusion, ne sont pas normalisés, et varient d'une implémentation à l'autre. Il est donc difficile de délimiter la frontière de l'intergiciel, tâche qui est néanmoins incontournable pour procéder à des expériences d'injection de faute. Nous proposons une méthode contournant ce problème au § 2.1.1.

[12] Dans le contexte du test, un *oracle* est un composant du banc de test qui est chargé de décider si la valeur renvoyée par un composant testé est valide ou non.

- la forme que prend l'intergiciel à l'exécution dépend de nombreux critères, dont les interfaces présentes dans le système, et les langages de programmation utilisés pour implémenter les objets client et les servants. En reprenant notre analogie de « glue », on peut dire que la forme que prend la glue dans le bâtiment final dépend du matériau qu'elle doit coller.

  Toutefois, il est impossible de mener des expériences d'injection de fautes sans avoir une cible concrète. Il faut donc intégrer à la méthodologie de caractérisation une étape de définition d'un système cible, qui soit représentatif des systèmes réels, et qui permette de couvrir le plus possible les configurations que peut prendre l'intergiciel à l'exécution. Nous présentons au § 2.1.1 une méthode pour aborder ce problème.

1.6.1 Transparence, interception et isolation

Une grande partie de la fonctionnalité implantée par un courtier est fournie de manière implicite, et ne résulte pas d'un appel explicite sur une interface publique. Considérons par exemple une invocation CORBA dans un programme utilisant le langage C++ :

```
result = theObject->theMethod("foo", 42);
```

La variable `theObject` est une des instances d'une classe qui hérite de classes fournies par le courtier. Au cours de l'exécution de cette méthode, le courtier identifie la machine sur laquelle s'exécute l'objet `theObject`, et établit (ou réutilise) une connexion réseau vers le port sur lequel écoute cet objet. Il emballe les paramètres de l'appel dans un format standard (le CDR) et envoie les données sur le réseau, puis attend la réponse du serveur et déballe la réponse dans la variable `result`. Si l'invocation a généré une exception CORBA, le courtier signale une exception C++ correspondante.

Toute cette activité est transparente pour le programmeur d'application (c'est-à-dire que son fonctionnement est opaque – il n'en voit pas les détails), et l'appel est identique, d'un point de vue syntaxique, à une invocation normale de méthode sur un objet local, non CORBA. La force de CORBA provient justement de ce que tous les mécanismes sous-jacents ne sont plus gérés par le programmeur applicatif, mais par le courtier.

Toutefois, pour fournir cette illusion de transparence, l'intergiciel est contraint de s'immiscer dans le code applicatif. Dans le code objet d'un programme C++ basé sur CORBA, les liens d'héritage entre classes applicatifs et classes de l'intergiciel entraînent un fort degré de couplage entre le code applicatif et le code provenant du fournisseur. Ce fort degré d'intrusivité de l'intergiciel est contraire au principe de précaution appliqué lors de l'intégration d'un composant COTS au sein d'une application semi-critique, où l'on cherche à isoler le composant pour protéger le reste du système des conséquences de sa défaillance.

Le problème auquel nous nous intéressons dans le chapitre suivant est de développer des mécanismes qui permettent d'activer les fonctionnalités d'un courtier et de tester leur comportement en présence de fautes.

1.7 Récapitulatif

Dans ce premier chapitre, nous avons situé le contexte et la problématique qui sous-tendent nos travaux. Nous avons décrit la tendance de l'industrie informatique à intégrer des composants sur étagère, y compris dans des systèmes semi-critiques. Nous avons exprimé les craintes des intégrateurs de systèmes quant à l'influence de ces composants, dont ils ne maîtrisent pas le processus de développement, sur la sûreté de fonctionnement des systèmes qu'ils construisent. Nous avons donné des exemples récents de fautes dans des intergiciels de communication, qui ont entraîné des conséquences graves pour leurs utilisateurs. Ce contexte motive notre travail sur la caractérisation du comportement en présence de faute d'intergiciels de communication.

Nous avons présenté le modèle à objets sur lequel s'appuient les systèmes CORBA, et analysé l'architecture de ces systèmes des points de vue traitement, ingénierie et technologique. Nous avons analysé les différentes stratégies d'implantation d'un courtier et leur impact sur les mécanismes de confinement d'erreur qui peuvent être mis en œuvre.

Nous avons examiné les différentes techniques de caractérisation de la sûreté de fonctionnement de systèmes informatiques, et brossé l'état de l'art en matière de caractérisation de logiciels exécutifs par injection de faute. Les études récentes indiquent que les mécanismes de communication sont un vecteur privilégié pour la propagation d'erreur dans les systèmes répartis. Les rares travaux qui ont porté sur des implémentations de la plate-forme CORBA ont fait apparaître des doutes quant à la robustesse de ces technologies logicielles.

Enfin, nous avons examiné les difficultés qui doivent être résolues avant de pouvoir appliquer les approches classiques de caractérisation comportementale à l'intergiciel.

Dans le prochain chapitre, nous présentons la méthodologie que nous avons développée qui permet de répondre à cette problématique.

Chapitre 2

Méthodologie

DANS ce chapitre, nous décrivons une méthode pour la caractérisation expérimentale des modes de défaillance d'une implémentation d'intergiciel. Le point de départ de cette méthode est l'analyse structurelle de systèmes à base d'intergiciel de communication, que nous avons présentée dans le chapitre précédent. Nous commençons par identifier les facettes d'un intergiciel de communication que nous souhaitons caractériser, et décrivons une méthode de fabrication de systèmes répartis qui puissent servir de support à notre étude expérimentale. Nous proposons un modèle de fautes adapté à ces cibles, en nous focalisant sur les fautes liées à la répartition. Enfin, nous décrivons un ensemble de techniques d'injection de fautes qui peuvent être utilisées pour évaluer l'impact de ces fautes au niveau de l'intergiciel.

2.1 Préliminaires à la caractérisation

Pour qu'une campagne d'injection de fautes fournisse des résultats qui soient exploitables, il faut que le contexte dans lequel se déroulent les expériences soit comparable au fonctionnement d'un système réel (critère de *représentativité* des expériences), et il faut que l'outillage d'injection de fautes fournisse des observations suffisamment riches pour être intéressantes. Plus précisément, les critères suivants doivent être pris en compte lors de la définition d'une campagne :

- identification des *frontières du système cible* : notre travail consiste à étudier le fonctionnement de l'intergiciel en présence de fautes. Nous devons pour cela commencer par délimiter la partie du système qui nous intéresse, et étudier comment nous pouvons l'isoler du reste du système. Une fois les frontières définies, nous pouvons examiner quelles sont les fautes qui peuvent affecter la partie cible, et étudier les canaux de propagation depuis la cible vers le reste du système.

- le *modèle de fautes* : quels types de fautes insérer dans le système, à quel endroit dans le système les injecter, et à quel instant (synchronisation avec l'activité au niveau applicatif) ?
- la *charge applicative* : quel profil applicatif, ou simulation de l'activité normale du système, employer pendant le déroulement d'une expérience ? La charge est clairement dépendante du système cible. Différentes charges peuvent induire des résultats différents, puisqu'ils activent le système différemment.
- la *classification des modes de défaillance* : comment surveiller le comportement du système et classifier ses modes de défaillance ? Il est important que tous les événements significatifs soient observés, ce qui peut être difficile dans le contexte d'un système distribué.

Nous allons détailler comment nous avons abordé chacun de ces points dans les sections suivantes.

2.1.1 Définition du système cible

Comme nous l'avons vu dans le chapitre 1, l'intergiciel joue le rôle de « glue » entre le niveau applicatif et le système d'exploitation. La forme que prend l'intergiciel à l'exécution dépend de plusieurs paramètres :

- l'interface OMG IDL du service (ou pour un client, de l'ensemble des interfaces qui sont invoquées par le client). En effet, les souches et les squelettes générés – que nous considérons comme faisant partie de l'intergiciel – dépendent de l'interface IDL à laquelle ils correspondent. La signature des méthodes, les types des paramètres, la liste des exceptions levées, et le type d'invocation (avec ou sans retour), ont tous une influence sur la manière dont l'intergiciel est activé.

  Par exemple, les fonctionnalités de mise à plat du type de donnée `double float`, fournies par l'intergiciel, ne seront activées dans le système qu'à condition que le service invoqué comprenne un argument de type `double float`. De plus, le mode d'activation de l'intergiciel peut dépendre du mode de l'argument passé (entrée-seule, entrée-sortie, ou sortie).
- l'API présentée par le système d'exploitation a un impact sur la configuration de la « partie basse » de l'intergiciel, qui interagit avec le système d'exploitation. En effet, le même intergiciel compilé pour Solaris ne se comportera pas de la même manière que s'il est compilé pour GNU/Linux, même si ces deux systèmes d'exploitation sont conformes à la même norme POSIX.

  Certaines implantations d'intergiciel cherchent à encapsuler les différences entre systèmes d'exploitation par l'utilisation de bibliothèques d'adaptation. Par exemple, l'implantation *ORBacus* utilise une bibliothèque appelée JTC (pour *Java Threads for C++*) qui émule l'interface d'interaction avec les processus légers de Java sur différentes plates-formes.

- les choix de configuration de l'intergiciel : le modèle de concurrence employé, la présence éventuelle d'intercepteurs, et le modèle de sécurité. Par exemple, le fait d'activer l'authentification et le chiffrement des connexions par SSL introduit un nouveau mode de défaillance « certificat invalide ».

Cette nature polymorphe de l'intergiciel pose problème lorsqu'on cherche à le caractériser, puisqu'une expérience d'injection de fautes ne peut être menée que sur un système concret, avec une configuration précise de l'intergiciel. Nous ne pouvons pas mener des expériences d'injection de fautes sur le produit tel qu'il est livré par le fournisseur d'intergiciel ; pour reprendre notre analogie d'intergiciel en tant que glue, cela reviendrait à évaluer la qualité de la colle en la maintenant dans son tube. Nous devons donc définir une procédure qui permette de générer un ensemble de systèmes basés sur le candidat intergiciel, qui sont représentatifs (en ce qui concerne l'utilisation résultante de l'intergiciel) de l'ensemble des systèmes possibles.

Nous proposons deux approches pour résoudre ce problème. La première consiste à utiliser comme cible les implantations de services du fournisseur d'intergiciel (par exemple, les services de désignation et de diffusion d'événements pour des cibles CORBA). Ces services sont livrés sous forme immédiatement exploitable ; la forme qu'ils prennent à l'exécution ne dépend pas du reste du système[1]. Ils fournissent d'office une interface explicite par laquelle le reste du système interagit avec eux, et que nous pourrons cibler dans nos expériences.

Une critique possible de cette approche est qu'elle ne permet de caractériser qu'une petite partie des capacités totales de l'intergiciel. Nous verrons dans le chapitre suivant que cette critique est moins fondée qu'elle ne pourrait le paraître, puisque – dans tous les systèmes que nous avons ciblés – les implantations de services d'un fournisseur donné s'appuient largement sur le reste de l'intergiciel de ce fournisseur[2]. Autrement dit, le fait de cibler un service particulier donne une couverture relativement large de la fonctionnalité totale de l'intergiciel. Nous revenons sur ce constat dans le chapitre 4.

Toutefois, ne cibler que les services ne permettrait pas de couvrir de manière complète les fonctionnalités de l'intergiciel. La seconde approche que nous proposons afin de produire des systèmes cibles pour nos expériences d'injection de fautes, consiste à créer des services « synthétiques », avec des interfaces synthétiques. L'objectif de ces interfaces synthétiques est de fournir un moyen d'activer les fonctionnalités « cachées » du courtier.

Par *interface synthétique*, nous entendons une interface qui est créée spécifiquement pour les besoins de nos expériences. Cette interface (et le servant correspondant, qui implémente les opérations offertes par l'interface) a pour unique objectif de nous

[1] Les services peuvent admettre certaines possibilités de personnalisation, par exemple, offrir un choix entre un fonctionnement avec ou sans persistance, mais le degré d'adaptation possible est plus faible que pour un courtier.

[2] En effet, pour implanter un service de désignation, il faut gérer des connexions réseau, emballer et déballer les paramètres de requêtes, gérer des exceptions et des tampons mémoire, traiter des références objet, etc. On couvre ainsi une fraction considérable de la fonction d'un courtier.

permettre d'activer les fonctionnalités de l'intergiciel, en maximisant le degré de couverture.

Plus précisément, les interfaces synthétiques et les servants correspondants devraient satisfaire les besoins suivants :

- couvrir tous les types de données qui peuvent être présents dans le langage de description d'interface (OMG IDL), y compris les types de données composites tels que les unions et structures, et les définitions d'exceptions ;
- inclure des opérations avec les arguments et types de retour qui couvrent toutes les combinaisons possibles de ces types de données, en prenant en compte toutes les conventions possibles de passage de paramètre (`in`, `out` et `inout`), et les modes d'invocation (avec ou sans retour) ;
- étant donné le grand nombre de jeux de test nécessaires pour satisfaire les deux critères précédents, il n'est pas envisageable de conduire ces tests manuellement. Le code d'injection qui cible ces interfaces synthétiques doit donc pouvoir être généré de manière automatique. Pour les mêmes raisons, il devrait être possible de générer de manière automatique une application de charge et des détecteurs de défaillance pour une interface donnée. Il est clair que cette dernière contrainte a une incidence forte sur le niveau sémantique des services que nous pouvons cibler.
- le service synthétique fourni devrait être délivré de manière à ce qu'il soit possible de détecter sa défaillance rapidement, depuis le niveau applicatif. En particulier, le service doit être déterministe.
- le service délivré devrait être dépendant de l'historique des invocations précédentes sur le service (c'est-à-dire qu'il ne devrait pas être sans état). En effet, si le service fourni dépend de l'état interne du servant, il présentera davantage de possibilités pour que les fautes se propagent à l'interface du service que si le servant est sans état.

Nous proposons de satisfaire ces critères en implantant un service d'*écho retardé*, qui comprend un ensemble d'opérations qui acceptent en paramètre un nombre arbitraire de types divers, et renvoient les arguments fournis lors du précédent appel au service. Ce service est déterministe, peut être implémenté pour des signatures d'opération arbitraires, et possède un état.

Une campagne d'injection de fautes visant un courtier comprend alors les étapes suivantes :

1. Générer une interface avec une combinaison arbitraire de types de données, d'exceptions, et des opérations avec des signatures diverses. Puisque l'ensemble des interfaces est infini, le processus de génération de l'interface est aléatoire.
2. Générer les implantations correspondantes pour le service, la charge applicative, l'injecteur de fautes (en fonction du type de faute injectée) et l'observateur de défaillance.

3. Invoquer le service avec des valeurs représentatives (qui sont fonction des types présents dans la signature des opérations), et observer le comportement du service d'écho, en s'assurant qu'il respecte sa spécification.

La génération des paramètres pour les invocations de service est un problème bien connu dans le domaine du test fonctionnel. Il existe plusieurs techniques pour choisir des entrées qui permettent de maximiser le taux de couverture du code, tout en maîtrisant le coût de leur génération ; nous avons choisi d'utiliser les techniques du test statistique [Thévenod-Fosse et al. 1995].

Pour chaque type de donnée OMG IDL, nous définissons une fonction de génération aléatoire. Dans le cas des types de données de base, il suffira généralement de préciser les plages de valeurs minimales et maximales autorisées pour ce type, puis de faire appel à un générateur de nombres aléatoires. Pour un type de données composite, comme une structure ou une séquence, on peut faire appel récursivement aux générateurs des types de base le composant. Par exemple, le type IDL `string` est composé d'une séquence de `char` ; il suffit donc de générer aléatoirement un entier positif qui déterminera la longueur de la chaîne de caractères, puis de faire appel au générateur de caractères aléatoires pour chaque position dans la chaîne.

Idéalement, nous pourrions activer chaque classe de fonctionnalité du courtier de manière indépendante, afin d'obtenir des informations détaillées sur les modes de défaillance par module fonctionnel. Hélas, il n'est pas possible dans notre contexte d'isoler les classes de fonctionnalité les unes des autres – par exemple, presque toutes les interactions avec un courtier depuis l'extérieur impliqueront l'utilisation des fonctionnalités d'emballage des structures de données et de gestion de tampons mémoire. Nous ne pouvons donc pas faire des mesures de robustesse par module fonctionnel, ni analyser la propagation d'erreur entre modules.

2.1.2 Un modèle de fautes pour les systèmes CORBA

Un *modèle de fautes* est une abstraction des différentes classes de fautes qui peuvent affecter un système. Dans le cadre d'une expérience d'injection de fautes, il s'agit de modéliser les perturbations qu'on va introduire artificiellement dans le système.

La représentativité des fautes simulées est une question importante lors de la définition d'une campagne d'injection de fautes. On cherche à injecter des fautes dont l'effet est similaire (au niveau des erreurs qui sont produites [Jarboui 2003]) à celui des fautes réelles, qui sont subies par les systèmes en phase opérationnelle. Un pré-requis à la comparaison des effets des fautes injectées et des fautes réelles est de disposer d'une base de fautes réelles, obtenue à partir de campagnes d'observation de systèmes déployés. Ce type d'étude fournit des statistiques sur les types de défaillances subies par une classe de système, ainsi que leur fréquence d'occurrence, et des hypothèses sur les fautes qui les ont provoquées.

Ce type d'analyse a été mené depuis de nombreuses années pour certaines classes de système, en particulier pour les *mainframes* (les « gros systèmes ») [Sullivan & Chillarege 1991]. En revanche, nous ne connaissons aucune étude de ce type concernant les systèmes à base d'intergiciel. Ceci peut s'expliquer par le fait que – contrairement aux *mainframes*, qui sont déployés en industrie depuis plus de 30 ans – les systèmes basés sur un intergiciel de communication n'ont été déployés à grande échelle que récemment. En conséquence, nous ne pouvons déterminer les classes de fautes qui affectent ces systèmes que par une analyse structurelle, en étudiant l'architecture d'un système type, en identifiant les points où les fautes peuvent survenir, et en étudiant les canaux potentiels de propagation d'erreur. Nous avons présenté notre analyse structurelle au § 1.3.8.

Dans la suite, nous nous intéressons aux fautes qui affectent l'intergiciel même ou dont les effets peuvent être amplifiés par l'intergiciel ; nous ne cherchons pas à lister de manière exhaustive toutes les fautes qui peuvent toucher un système distribué, et encore moins à étudier leur effet. Ces fautes peuvent être classifiées comme suit [Marsden et al. 2002a] :

- les fautes physiques, affectant des éléments matériels du système, tels que la mémoire, les registres du microprocesseur, le fonctionnement des unités d'entrées/sorties. Par exemple, un rayon cosmique peut provoquer l'inversion d'un ou de plusieurs bits en mémoire.
- les fautes de communication, comme la perte, la duplication, le réordonnancement ou la corruption de messages. Cette classe de fautes est généralement supposée ne pouvant pas affecter des intergiciels qui se basent sur un protocole de transport fiable (ce qui est le cas de IIOP dans CORBA). Toutefois, des recherches récentes, présentées au § 2.2.5, indiquent que ces fautes méritent d'être étudiées.
- les fautes du logiciel (qu'elles soient de conception ou de programmation), au niveau de l'application, de l'intergiciel ou du système d'exploitation. Par exemple, une application peut transmettre une référence objet invalide à l'intergiciel en paramètre d'une opération, ou l'intergiciel peut emballer de manière incorrecte certains types de données vers le CDR, ou omettre de vérifier certains codes d'erreur renvoyés par le système d'exploitation.
- les fautes relatives à la gestion des ressources : l'effet de « vieillissement » du logiciel engendre des effets tels que des fuites de mémoire (ce qui est particulièrement fréquent dans les applications CORBA), des effets de fragmentation de la mémoire, de saturation du stockage local, de non-libération de ressources telles que des descripteurs de fichier.

Les fautes physiques et les fautes de communication sont généralement transitoires, et elles sont très rares ; ces fautes ne peuvent que difficilement être révélées par le test fonctionnel du système. Les fautes du logiciel, quant à elles, sont plus aptes à être détectées lors des phases de test (avant la mise en opération du système), mais certaines fautes sont très difficiles à révéler. En effet, le déclenchement de la faute peut

être dû à un événement qui a lieu très rarement, ou bien suite à un enchaînement non déterministe d'événements.

Pour chaque classe de fautes, nous décrivons au § 2.2 des techniques d'injection par logiciel qui permettent de simuler l'effet de ce type de fautes sur des systèmes à base d'intergiciel.

Les techniques d'injection de fautes par logiciel peuvent être classifiées selon l'étape où les fautes sont injectées :

- injection à la compilation : les instructions du logiciel sont modifiées avant l'exécution de l'image binaire. La faute est insérée dans le code source ou dans le code assembleur du programme. Cette méthode d'injection ne nécessite pas de logiciel d'injection supplémentaire pendant l'exécution, et ne cause aucune perturbation dans le système cible pendant l'exécution. Toutefois, elle est difficile à appliquer à une cible dont on ne possède pas le code source[3], et ne peut donc pas être appliquée à tous les scénarios d'utilisation de logiciels COTS.
- injection à l'exécution : au cours de l'exécution du système, un mécanisme dédié est utilisé pour déclencher l'injection de la faute. Ce déclenchement peut être basé sur un temporisateur, qui s'active un certain temps après le début de l'expérience. Alternativement, le déclenchement peut être spatial, attendant que le programme ait exécuté une instruction ou lu une donnée se trouvant à une adresse mémoire prédéterminée. Le déclenchement peut attendre l'occurrence d'un événement significatif, comme la réception d'un message depuis un nœud donné.
- injection hybride : cette approche est utilisée pour injecter lors de la compilation des fautes qui se comportent à l'exécution comme des fautes transitoires. Le principe consiste à ajouter lors de la compilation un injecteur minimal, dont le rôle à l'exécution sera d'autoriser ou non l'injection de la faute.

Le *degré d'intrusivité* du mécanisme employé pour insérer les fautes dans le système au cours d'une expérience doit être réduit au minimum, afin qu'une exécution en présence de faute soit comparable à une exécution nominale.

[3]Il est possible de procéder à la modification de l'image binaire pour émuler certaines fautes simples (par exemple effectuer des inversions de bit, ou modifier le sens d'une instruction de branchement), mais il est difficile d'insérer des fautes plus subtiles (forcer le déroutement d'un message, par exemple).

2.1.3 Modes de défaillance d'un intergiciel de communication

Dans cette section, nous détaillons les types des défaillances qui peuvent affecter un intergiciel, et analysons l'impact de ces défaillances sur le système distribué qui s'appuie sur cet intergiciel. Notre analyse se limite aux défaillances qui sont imputables à l'intergiciel, ou dont les effets peuvent être amplifiés par l'intergiciel.

Rappelons que dans les systèmes que nous considérons, la défaillance d'un composant devient une faute dans le système global. L'impact de ces défaillances dépend donc des mécanismes de détection d'erreur et de recouvrement mis en place dans le système final. Nous classifions les défaillances en fonction de l'élément de l'intergiciel qui les engendre : outils de développement, courtier, et services CORBA.

Défaillances des outils de développement

Citons quelques exemples de défaillances dues aux outils de développement associés à l'intergiciel de communication :

- le compilateur IDL ne compile pas une description d'interface alors qu'elle est correcte ;
- le compilateur IDL compile une description d'interface alors qu'elle est invalide ;
- le code généré par le compilateur IDL (les souches et les squelettes) est invalide. Dans les cas d'erreurs de syntaxe, l'erreur sera détectée par le compilateur du langage de programmation. Des corruptions plus subtiles peuvent provoquer un comportement erroné à l'exécution.
- les fichiers d'en-tête du courtier sont erronés : ils ne correspondent pas à la bibliothèque partagée où réside le coeur du courtier, ou ne correspondent pas au code généré par le compilateur IDL.

L'essentiel de ces défaillances devrait être détectées lors des phases de développement ou d'intégration du système. Nous considérons que les défaillances détectées avant la phase d'intégration sont moins graves qu'une défaillance qui surviendrait plus tard dans le cycle de vie, en phase opérationnelle. Nous avons donc peu travaillé sur la caractérisation de ces outils de développement.

Défaillances du courtier

Énumérons les fonctionnalités d'un courtier CORBA qui peuvent défaillir :

- transmission de la requête : le courtier est responsable de l'acheminement de la requête depuis le client vers le *servant*, ainsi que de la transmission du résultat de la requête (à l'exception des opérations `oneway`). La transmission de la requête peut défaillir soit côté client, soit côté serveur. La défaillance peut être par omission (la requête n'est pas transmise au servant), par duplication (la même requête est transmise plusieurs fois), par fabrication (le servant reçoit des requêtes qui n'ont été émises par aucun client), ou par erreur de routage (la requête ou sa réponse est transmise au mauvais servant).

  Nous distinguerons dans la suite deux types d'erreurs de transmission : le refus de transmission (qui peut être détecté immédiatement par le client), et le gel de la transmission (la réponse à la requête n'arrive pas au bout d'un temps prédéterminé, bien plus long que le temps de transmission normal).
- intégrité de la requête : le courtier corrompt les données qu'il transmet.
- mauvais signalement des exceptions : lors d'une invocation de méthode CORBA, le courtier côté serveur doit informer le courtier côté client qu'une exception s'est produite lors du traitement de la requête, en lui renvoyant un message GIOP avec un type particulier. Le courtier côté client doit signaler à l'application qu'une exception s'est produite, en respectant la manière dont sont définies les exceptions par la projection langage. Si ce signalement est incorrect, soit par omission, soit parce qu'un courtier a signalé une exception qui n'a pas réellement eu lieu, soit parce que le type d'exception signalé ne correspond pas à celui qui a été signalé par le servant, les mécanismes de tolérance aux fautes du système ne seront pas activés correctement.

Défaillances de services CORBA

Examinons maintenant l'impact des défaillances qui touchent les principaux services CORBA : la désignation, la diffusion d'événements, et le référentiel d'interfaces.

Le **service de désignation** constitue un *point dur* (ou « single point of failure ») du système : la défaillance de ce service peut avoir des effets catastrophiques pour un système réparti, puisque les objets ne peuvent plus obtenir les références des services qu'ils souhaitent invoquer.

L'impact de la défaillance de ce service dépend considérablement du rôle qu'il joue dans le système, qui peut y faire très peu appel ou en dépendre beaucoup. Il est possible de se passer d'un service de désignation, en allouant les références objet statiquement ; ce choix est fait dans certaines applications embarquées, où les ressources disponibles sont limitées, et où la configuration du système évolue peu dans le temps. D'autres systèmes plus dynamiques font très largement appel au service de désignation pour accroître la souplesse du système.

Les fournisseurs utilisent des techniques variées pour rendre leur service de désignation tolérant aux fautes. La plupart des implantations prennent en charge la persistance des couples nom–référence, en sauvegardant régulièrement l'état du service sur un support stable. Certaines implantations permettent l'utilisation d'une configuration

primary–backup, où la défaillance du serveur primaire est masquée par un basculement automatique vers le serveur secondaire.

Les modes de défaillance de ce service sont les suivants :

- l'arrêt inopiné du service : le ou les processus hébergeant le service s'arrêtent. Ce mode de défaillance peut être testé localement au nœud, en examinant la table des processus du nœud, ou à distance, en cherchant à établir une connexion réseau au service. Lorsque le service s'est arrêté, la tentative de connexion réseau sera immédiatement refusée par le nœud distant.
- le gel du service : un gel diffère d'un arrêt inopiné en ce que le nœud distant accepte la connexion réseau au service, mais qu'il ne répond pas dans un laps de temps prédéfini qui est largement supérieur au temps de service normal.
- la corruption de données : le service oublie des enregistrements de noms, ou renvoie des réponses incorrectes à des demandes de résolution de nom.

Nous distinguerons dans la suite deux types de gels de service, en fonction du nombre de clients affectés. Un *gel total* implique que tous les clients du service sont bloqués indéfiniment en attendant la réponse du service. Un *gel partiel* implique qu'un seul client est bloqué, alors que d'autres clients perçoivent un service normal pendant la même période.

Le **service de diffusion d'événements** fournit un mécanisme de communication de type producteur-consommateur (ou *publish-subscribe)*, qui complète la communication classique de type requête-réponse. Il fournit une abstraction de canal d'événements, auquel des producteurs et des consommateurs d'événements peuvent s'inscrire ; le canal est responsable de la propagation des événements entre les producteurs et les consommateurs. Ce mécanisme permet de réduire le degré de couplage entre les objets dans le système, puisqu'un producteur n'a pas besoin de se préoccuper du nombre de consommateurs qui sont intéressés par ses événements, et un consommateur n'a pas besoin de savoir qui est à l'origine d'un événement qu'il reçoit.

Les conséquences de la défaillance du service de diffusion dépendent évidemment de la criticité des informations qu'il transporte, mais en général ce service, lorsqu'il est déployé, est un point dur du système. Même une dégradation de sa qualité de service peut nuire à la sûreté de fonctionnement. Les types de défaillances envisageables pour ce service sont :

- l'arrêt inopiné ou le gel du service ;
- la perte ou la duplication d'événements ;
- la corruption d'événements ;
- une dégradation forte de la qualité de service fourni (temps de diffusion qui augmente fortement).

Le **référentiel d'interfaces** fournit des méta-informations sur des interfaces. Il permet à des clients de connaître l'ensemble des opérations offertes par une interface, ainsi que le nombre et le type de ces arguments. Cette information est surtout utile aux clients qui souhaitent utiliser le mécanisme d'invocation dynamique de CORBA (le *Dynamic Invocation Interface*, ou *DII*), qui permet de s'affranchir de l'utilisation de souches.

Les modes de défaillance du référentiel d'interfaces sont semblables à ceux des autres services. L'impact de sa défaillance est considérable pour les clients qui exploitent le DII (puisqu'ils ne pourront plus construire des requêtes dynamiquement), mais ne devrait pas avoir d'impact sur les clients et les serveurs qui utilisent les souches et squelettes.

La dernière classe de défaillances qui peut affecter les systèmes à base d'intergiciel a trait à la gestion des configurations. Dans un système réparti à longue durée de vie, il peut être nécessaire de procéder à des mises à jour de sous-systèmes, et parfois d'ajouter des fonctionnalités à un système en cours d'opération. Cette activité de maintenance introduit un risque d'incohérence dans un système où certains clients utilisent des informations périmées concernant l'interface des serveurs. Ce problème relève davantage de l'administration de systèmes répartis que de la sûreté de fonctionnement des implantations d'intergiciel ; il ne sera donc pas traité dans cette thèse.

2.1.4 Définition du profil applicatif

Dans une campagne d'injection de fautes, la *charge applicative* est un élément du banc de test qui est chargé d'activer les fonctionnalités du composant cible, afin de simuler une activité « normale » du système. Pour que les expériences soient représentatives, la charge applicative doit être représentative de l'activité normale du système.

Il existe plusieurs approches pour atteindre cette représentativité :

- utiliser une application de charge réelle ;
- observer des systèmes réels en opération, et obtenir des traces des interactions du composant cible avec le reste du système. On peut alors générer une application de charge synthétique qui engendre les mêmes interactions avec le composant cible.
- écrire une application de charge qui maximise la couverture fonctionnelle du composant cible (généralement au niveau de son interface, si l'on considère le composant en tant que boîte noire).

En l'absence de systèmes réels sur lesquels nous pourrions appliquer les deux premières approches, nous avons choisi d'utiliser la dernière approche. Nous verrons dans le chapitre suivant que nous avons développé des charges applicatives spécifiques pour chacun des services CORBA que nous avons ciblés. Pour les expériences ciblant les interfaces synthétiques, nous avons des charges applicatives générées automatiquement en même temps que les interfaces.

2.2 Techniques d'injection de fautes pour cibler l'intergiciel

Cette section décrit les différentes techniques que nous proposons afin de caractériser la sûreté de fonctionnement d'un système à base d'intergiciel. Les types de fautes que nous proposons pour un environnement CORBA, et les techniques que nous proposons pour les émuler, peuvent être classifiés comme suit :

- des fautes internes, résultant de fautes matérielles ou logicielles, qui sont simulées en utilisant respectivement des techniques de corruption de la mémoire et des techniques de mutation de code ;
- des fautes qui se propagent à l'intergiciel depuis le niveau applicatif, que nous simulons à l'aide de techniques de test de robustesse et de mesures de tenue en charge ;
- des fautes qui se propagent à l'intergiciel depuis le système d'exploitation, que nous simulons à l'aide de techniques d'interposition d'appel système ;
- des fautes qui se propagent à l'intergiciel depuis des nœuds distants, que nous émulons par des techniques de corruption de messages.

Nous détaillons ces différentes techniques dans les sections suivantes. Le chapitre 3 présentera les résultats issus de campagnes mettant en œuvre certaines de ces techniques.

2.2.1 Corruption de l'espace d'adressage

La technique d'injection de fautes la plus répandue consiste à simuler des fautes physiques qui affectent la mémoire du système, entraînant la corruption de données dans la mémoire principale, dans les registres du microprocesseur, et dans les contrôleurs d'entrées/sorties. Ce type de fautes peut, par exemple, être provoqué par le rayonnement électromagnétique provenant de l'espace. Différentes études ont montré que ce modèle de fautes est relativement représentatif d'autres classes de fautes, y compris les fautes du logiciel [Fuchs 1998, Cheynet et al. 2000].

On peut classer les fautes en fonction de leur période d'activation :

- une faute permanente, qui simule la défaillance d'un élément mémoire. Les fautes de ce type les plus fréquentes sont le collage à zéro ou le collage à un d'un bit. Le bit affecté peut être dans un registre du microprocesseur ou d'une carte contrôleur, un mot mémoire, ou sur un bus de communication.
- une faute transitoire, telle que l'inversion d'un bit dans un mot mémoire, qui simule les effets de rayonnements électromagnétiques. Ce type de fautes est plus difficile à diagnostiquer que les fautes permanentes, et (d'après les études) est plus fréquent en opération.
- une faute intermittente, avec une période de réactivation.

Lors d'une expérience de caractérisation, l'injection de la faute peut être déclenchée un certain temps après le début de l'expérience ; dans ce cas, on parle de déclenchement temporel. Le déclenchement de la faute peut également être spatial, au quel cas elle est injectée à l'instant où le mot mémoire ciblé est accédé en lecture ou en exécution par le système. Enfin, elle peut être statique, c'est-à-dire que la faute est injectée avant même que l'expérience soit démarrée.

Les zones mémoire à cibler dépendent de la stratégie d'implémentation du courtier. La caractérisation d'un courtier intégré au noyau nécessitera des corruptions dans l'espace d'adressage du noyau du système d'exploitation, ainsi que dans les espaces d'adressage des applications CORBA, afin de couvrir les composantes du courtier qui résident dans l'application. La caractérisation d'un courtier orienté-daemon nécessitera la corruption de l'espace d'adressage du daemon d'activation, ainsi que celui des applications CORBA. Concernant un courtier résidant dans les applications, il est possible de cibler différentes zones de l'espace d'adressage de l'application visée :

- la pile et le tas de l'application ;
- le code privé provenant de souches et de squelettes, qui est lié à l'application ;
- la zone de données associée à la bibliothèque partagée implantant le courtier ;
- la zone de code de cette bibliothèque partagée.

Alors que les trois premières zones de l'espace d'adressage sont privées à chaque processus, le code dans la bibliothèque partagée est partagé entre tous les courtiers sur ce nœud[4]. Ceci implique l'existence d'un canal de propagation d'erreur introduit par le courtier, sous la forme de son code partagé. Il faut cependant noter que l'utilisation des bibliothèques système entraîne le même type de problème, puisqu'elles sont partagées par toutes les applications.

L'information sur les plages d'adresses correspondant à chacune de ces zones peut être obtenue en interrogeant les fonctionnalités réflexives du système d'exploitation, par exemple en parcourant le système de fichiers `/proc` sur Linux, ou à l'aide de l'outil `pmap` sous Solaris. Il existe plusieurs techniques pour effectuer par logiciel l'inversion d'un bit en mémoire ; on peut, par exemple, utiliser les fonctionnalités de mise au point (les *hardware watchpoints*) de certains microprocesseurs modernes pour passer en mode superviseur lorsque l'adresse mémoire cible est lue.

Contrairement au cas des fautes externes, où il est difficile d'activer un module fonctionnel de l'intergiciel indépendamment des autres, l'injection de fautes de la mémoire permet de cibler des modules particuliers de l'intergiciel. En effet, il est possible de décomposer l'espace d'adressage du courtier par catégorie fonctionnelle et de choisir une adresse à corrompre dans un module cible.

[4]Dans les systèmes d'exploitation modernes, la zone mémoire contenant le code d'une application n'est pas accessible en écriture, et peut donc être partagée entre plusieurs processus sans entraîner de complications liées à la cohérence.

Déroulement d'une campagne

Avant de pouvoir lancer une expérience, il faut générer une application de charge, un injecteur et des outils d'observation. Au cours d'une expérience, le banc de test simule une faute et observe son impact sur le système, l'intergiciel et les applications. Une *séquence* regroupe des expériences pour lesquelles les fautes sont de même nature. Un nombre suffisant d'expériences permet d'obtenir des résultats statistiques réalistes. Une *campagne* d'expérimentation regroupe des séquences d'expériences visant la même cible, sous un même profil d'activation.

Les mesures qui peuvent être obtenues par des campagnes utilisant cette technique d'injection de fautes sont : la caractérisation des modes de défaillance, des informations sur la propagation d'erreur entre modules fonctionnels du courtier, et des mesures de la latence de détection d'erreur.

2.2.2 Techniques de mutation de code

Cette technique de caractérisation cherche à évaluer l'impact de fautes du logiciel. Elle consiste à introduire artificiellement des erreurs dans le code source du programme, et à observer le comportement du candidat modifié (qu'on appelle un *mutant*). Certains travaux [Daran & Thévenod-Fosse 1996] ont démontré que la mutation de programmes induit des erreurs qui sont de nature similaire aux erreurs provoquées par des fautes de programmation réelles.

Les premiers travaux dans ce domaine ont ciblé le test, en utilisant la mutation afin de mesurer la pertinence d'un ensemble de jeux de tests. Des travaux plus récents [Voas & McGraw 1997] ont employé la mutation de programme comme technique de caractérisation de la robustesse de logiciels. Ces travaux sont plus proches de notre objectif, qui est d'identifier les types d'erreurs et de défaillances qui peuvent être provoquées par des fautes de logiciel, et d'évaluer la capacité des mécanismes de détection d'erreur du système à détecter et gérer ces erreurs.

Les travaux rapportés dans la littérature consistent essentiellement à effectuer des mutations qui changent la valeur d'une constante dans le programme, ou le type d'un opérateur (par exemple, changer un opérateur + en opérateur −, ou changer le sens d'un opérateur de comparaison dans une instruction conditionnelle) [Daran & Thévenod-Fosse 1996]. D'autres exemples de mutations sont le remplacement d'un nom de variable ou de fonction par une autre variable ou fonction. Des travaux plus récents ont ciblé le développement d'opérateurs de mutation orientés objet [Chevalley & Thévenod-Fosse 2002], qui visent à simuler des fautes de logiciel spécifiques à l'utilisation de techniques de conception et de programmation orientées objet. Un exemple de ces opérateurs de mutation est d'échanger un opérateur de test d'égalité de référence par un test d'égalité structurelle.

Si l'on considère l'application de ces techniques à des systèmes basés sur l'intergiciel, on peut identifier plusieurs cibles de mutation :

- Les interfaces IDL. Par exemple, un paramètre passé avec des conventions d'appel en entrée (le qualificateur `in` dans le langage OMG IDL), pourrait être changé afin d'être passé en entrée-sortie (`inout`). La mutation pourrait également affecter des définitions de types de données, par exemple, en remplaçant une séquence de longueur non bornée par une séquence de longueur fixe.
- Les souches et squelettes générés par le compilateur IDL. Ces mutations ciblent des fautes dans la chaîne de compilation CORBA. Par exemple, deux paramètres d'une méthode pourraient être intervertis dans le squelette avant d'être transmis vers l'objet distant. Si les types des paramètres sont incompatibles, ce type de mutation devrait être détecté lors de la phase de compilation de l'application.
- Le code source de la partie du courtier qui est résidente dans l'application (souvent sous la forme d'une bibliothèque partagée, comme nous l'avons vu au § 1.3). Cibler cette partie de code vise à évaluer l'effet de fautes du logiciel résiduelles dans l'intergiciel lui-même. Une manière de construire un modèle représentatif de ce type de fautes serait d'examiner les correctifs d'erreurs qui sont appliqués dans les versions successives de l'intergiciel, afin d'analyser les types de fautes qui ont été corrigés par les développeurs.
- Au niveau de l'application. Cibler cette partie du code permet d'évaluer l'impact de fautes qui seraient introduites par le programmeur d'application. Dans ce contexte, des modèles de fautes usuels tels que le *Orthogonal Defect Classification* [Sullivan & Chillarege 1991] pourraient être employés. Les fautes d'initialisation et la corruption de pointeurs sont des exemples de mutation.

De la même manière que l'utilisation d'un langage de programmation orienté-objet introduit de nouvelles classes de fautes du logiciel, qu'il convient de simuler à l'aide d'opérateurs de mutation spécifiques, il serait intéressant d'identifier un certain nombre d'**opérateurs de mutation orientés CORBA**, qui auraient comme objectif de simuler des fautes du logiciel qui proviennent de l'utilisation d'un courtier CORBA. Par exemple, la gestion de la mémoire est notoirement délicate dans un contexte CORBA, lorsqu'on utilise des langages de programmation primitifs (tels que C et C++) qui ne fournissent pas de gestion automatique de l'allocation mémoire. Il pourrait donc être intéressant de chercher à développer des opérateurs de mutation qui simulent une fuite mémoire à l'interface entre le courtier et l'application. D'autres opérateurs de mutation pourraient cibler l'utilisation de références CORBA.

Malheureusement, ce travail de définition d'opérateurs de mutation est (sauf en ce qui concerne les interfaces IDL) nécessairement spécifique à un langage de programmation donné. Les implémentations de courtiers CORBA les plus communément utilisées sont écrites en C++, mais d'autres sont écrites en Java ou en C par exemple. Appliquer le même type d'analyse à ces autres courtiers nécessiterait le portage de l'ensemble des opérateurs de mutation, ainsi que de l'outillage associé qui permet de les introduire automatiquement dans un programme. L'effort nécessaire à ce type de portage dépendrait fortement du degré de ressemblance entre les langages de programmation visés.

Le déroulement d'une expérience utilisant cette technique d'injection de fautes est particulier dans le sens où il faut, avant chaque expérience, recompiler la partie du système qui a été affectée par la faute. Certaines mutations auront des effets qui peuvent être détectés à la compilation, auquel cas il est inutile de procéder à une exécution du système. Si la faute n'est pas détectée à la compilation, on exécute le système pendant un laps de temps prédéfini, et on observe le comportement des différents éléments, pour savoir si le mutant est *tué* (si l'erreur engendrée par la faute est détectée) ou non.

2.2.3 Techniques de corruption à l'API

La *robustesse* d'un système est une mesure de sa capacité à délivrer un service correct en présence d'entrées invalides et de conditions environnementales défavorables, ou « stressantes ». Le test de robustesse consiste à appliquer des données invalides à l'interface externe du système, puis à observer son comportement. En appliquant des tests de robustesse, on considère chaque interface d'un sous-système comme étant un canal de propagation d'erreur potentiel qui a été mis en place par les développeurs du système, et on cherche à évaluer le degré d'étanchéité de ces interfaces vis-à-vis des fautes qui peuvent se propager depuis d'autres sous-systèmes.

Le test de robustesse nécessite, pour sa mise en œuvre, que le système cible possède une interface de service explicite, qui peut alors être ciblée par l'injecteur de fautes. Ceci est difficile dans le cas d'un courtier CORBA, dans la mesure où, comme nous l'avons vu au § 1.6.1, une grande partie de la fonctionnalité fournie par un courtier est implicite. En effet, la fonctionnalité qui est disponible via une interface explicite se résume aux points suivants :

- l'initialisation du courtier : certaines opérations permettent la prise en compte d'informations provenant de l'environnement d'exécution afin d'obtenir des références initiales vers certains services (processus d'amorce ou de *bootstrap*). C'est le cas en particulier de l'opération `resolve_initial_references`, qui peut par exemple analyser les arguments fournis sur la ligne de commande lors du lancement de l'application.
- gestion de l'adaptateur d'objets (POA), pour un objet qui joue un rôle de serveur. Les opérations explicites de l'interface du POA permettent à un servant de s'inscrire auprès de son adaptateur d'objets, et de contrôler son cycle de vie.
- gestion de politiques : certaines opérations permettent le contrôle de changements dynamiques dans les politiques utilisées par le courtier, telles que le modèle de concurrence, la politique de contrôle d'accès.
- la conversion de références objet vers une représentation textuelle, et l'opération inverse d'extraction d'une référence à partir d'une chaîne de caractères.
- des opérations utilitaires, permettant la création de certains types de données.

Les travaux rapportés dans [Pan et al. 2001] sur le test de robustesse de courtiers CORBA ont ciblé environ 20 opérations dans cette interface publique. Ces opérations constituent une proportion relativement réduite de la fonctionnalité fournie par un courtier, et elles sont pour la plupart utilisées uniquement lors de l'initialisation du courtier. En effet, une grande partie de la fonctionnalité implémentée par un courtier est fournie de manière implicite, et ne résulte pas d'un appel explicite sur une interface publique. Considérons par exemple une invocation CORBA dans un programme écrit en langage C++ :

```
result = theObject->theMethod("foo", 42);
```

La variable `theObject` est une instance d'une classe qui hérite de classes fournies par le courtier. Au cours de l'exécution de cette méthode, le courtier identifie le nœud sur lequel s'exécute l'objet `theObject`, et établit (ou réutilise lorsqu'elle existe déjà) une connexion réseau vers ce nœud. Il emballe les paramètres de l'appel dans un format standard (le CDR), et envoie les données sur le réseau. Il attend la réponse du serveur, et déballe la réponse dans la variable `result`. Si l'invocation a généré une exception CORBA, le courtier lève une exception C++ correspondante.

Toute cette activité est transparente pour le programmeur d'application (ou opaque – il ne voit pas les détails), et l'appel est identique d'un point de vue syntaxique à une invocation normale de méthode sur un objet local, non CORBA. La force de CORBA provient justement de ce que tous les mécanismes sous-jacents ne soient plus gérés par le programmeur applicatif, mais par le courtier.

Cependant, cette transparence implique que la fonctionnalité fournie par le courtier n'est pas présentée aux clients par l'intermédiaire d'une interface explicite. On ne peut pas, par conséquent, appliquer la technique de test de robustesse à ces fonctionnalités de transparence.

Ces fonctionnalités implicites fournies par le courtier peuvent être catégorisées comme suit :

- l'interaction avec le langage de programmation du niveau applicatif : il faut exposer les interfaces spécifiées par la norme CORBA de manière compatible avec la projection langage. Il faut également fournir des méthodes d'emballage et de déballage pour tous les types CORBA, gérer la création et la destruction des objets, et implémenter les exceptions de manière compatible avec la projection langage.
- fonctionnalités orientées réseau : résolution des adresses des nœuds, établissement de connexions réseau, envoi et réception d'informations depuis des machines distantes ;
- gérer la concurrence en fonction de la politique demandée par l'application, en s'appuyant sur les primitives fournies par le système d'exploitation ;
- gestion de ressources : allouer et libérer des tampons mémoire, des pools de processus légers.

Déroulement d'une campagne

Une expérience d'injection de fautes utilisant cette approche de test de robustesse consiste à exécuter les étapes suivantes :

1. Générer une interface avec une combinaison de définitions de types et des signatures d'opération.
2. Générer une implantation correspondante pour le service, la charge applicative, et les composants d'observation.
3. Déclencher l'injecteur de fautes, qui invoque alors le service avec des paramètres corrompus.
4. Observer la réaction du système face à cette situation « stressante ».

Il convient de sélectionner les paramètres employés pour l'invocation du service afin de maximiser la couverture des chemins dans le code ciblé, c'est-à-dire dans le code du courtier, des souches et des squelettes. Cette même problématique de génération de paramètres afin de maximiser des critères de couverture se retrouve dans le domaine du test fonctionnel. Plusieurs techniques sont connues, parmi elles la génération statistique d'entrées de test [Thévenod-Fosse et al. 1995].

En l'absence d'informations sémantiques sur les domaines d'entrée admis par les opérations dans l'interface, les types de corruption qui peuvent être appliqués aux paramètres dépendent uniquement de leur type IDL, ainsi que de la projection langage. Il faut noter que la majorité des types OMG IDL sont « incorruptibles », dans le sens où toutes les formes binaires qui peuvent être stockées en mémoire sont une représentation valide d'un élément du type. D'autres types, par contre, ont un domaine d'entrée plus restreint, et peuvent donc faire l'objet de corruptions. C'est le cas par exemple des références objets.

2.2.4 Technique d'injection dans l'exécutif

La couche d'intergiciel s'appuie sur des services fournis par le système d'exploitation, tels que la communication via le réseau d'interconnexion, l'ordonnancement de processus et de processus légers, et la sauvegarde d'état au travers du système de fichiers. Les détails de ces interactions entre l'intergiciel et le système d'exploitation dépendent de la nature du système d'exploitation, mais dans la plupart des cas elles consistent en :

- des demandes de service provenant de l'intergiciel, par le biais d'*appels système*. Ces appels système prennent en paramètre un certain nombre d'arguments, et l'appelant reçoit une valeur en retour ;
- des notifications asynchrones, permettant au système d'exploitation d'informer l'intergiciel qu'un événement a eu lieu.

La technique d'injection dans l'exécutif cherche à évaluer les conséquences d'erreurs qui pourraient se propager à l'intergiciel par le biais de cette interaction. Cette propagation peut se faire suivant plusieurs schémas :

- une faute matérielle sur le nœud local entraîne un dysfonctionnement du système d'exploitation, qui se propage à l'intergiciel via une demande de service ;
- une faute logicielle dans le noyau ou dans les bibliothèques système entraîne un dysfonctionnement du système d'exploitation, qui se propage alors jusqu'à l'intergiciel ;
- le noyau prend trop de temps pour exécuter une demande de service, entraînant la défaillance temporelle de l'intergiciel ;
- l'intergiciel fait une demande de service erronée, qui entraîne la défaillance du système d'exploitation [Koopman & DeVale 1999]. Dans ce cas, c'est la robustesse du système d'exploitation qui est en jeu, ce qui sort du champ de notre thèse ;
- l'intergiciel interprète mal la valeur de retour d'un appel système, entraînant la défaillance de l'intergiciel ;
- l'intergiciel interprète mal une notification asynchrone, menant à sa défaillance.

Ces interactions se passent à travers l'API du noyau de système d'exploitation ; dans la suite nous utilisons la terminologie du standard POSIX, mais des concepts semblables existent dans la majorité des systèmes modernes.

Propagation via un code retour

Considérons plus en détail la propagation d'une erreur depuis le système d'exploitation via le code retour d'un appel système. La majorité des appels système POSIX comprennent un code retour qui doit être vérifié par l'appelant. Typiquement, la valeur 0 indique que l'opération s'est effectuée sans problème, et une valeur négative indique une déviation par rapport au déroulement nominal de l'appel système. Parfois la valeur de retour de l'appel système joue un double rôle, servant à la fois pour communiquer une valeur (lorsque la valeur est positive), et pour signaler une erreur (lorsqu'elle est négative).

Lorsque le code retour d'un appel système indique un déroulement anormal, l'application peut consulter une variable de type entier nommée `errno`, qui fournit des informations supplémentaires sur la nature du problème. La valeur de cette variable indique quel type d'erreur a eu lieu (*c.f.* quelques exemples dans la figure 2.1 page suivante).

Pour chaque appel système, la spécification POSIX précise la liste des codes d'erreur qui peuvent résulter de l'appel, avec une description de la sémantique du code d'erreur. Parfois une grande variété de situations peut résulter d'un seul appel système, d'où la difficulté à programmer de manière robuste. La figure 2.2 page suivante illustre cette

EINVAL	l'un des paramètres de l'appel système était invalide
EPERM	l'appelant ne possède pas les autorisations nécessaires pour effectuer l'opération demandée
ENOMEM	le système n'a pas suffisamment de ressources mémoire pour effectuer l'opération
ENOSPC	le disque local est saturé
ECONNREFUSED	la connexion réseau vers le système distant n'a pas pu être établie
EADDRINUSE	l'adresse réseau est déjà utilisée par un autre processus
EINTR	l'exécution de l'appel système a été interrompue par l'arrivée d'un signal

FIG. 2.1 – *Exemples de code retour d'appel système*

difficulté par un exemple d'appel système utilisé par l'intergiciel pour la communication réseau.

Ce code retour peut servir de vecteur de propagation d'erreur de plusieurs manières :

- le code indique une situation d'erreur, mais le programmeur de l'intergiciel n'a pas pris en compte cette possibilité. Le résultat peut être un comportement par défaut (par exemple l'arrêt immédiat de l'application), ou une corruption non détectée de l'état de l'application.
- le code indique qu'un appel système a été interrompu (et qu'il devrait être rejoué par l'intergiciel), mais le programmeur n'a pas prévu ce cas de figure.
- le système d'exploitation est défaillant, et renvoie un code retour qui n'est pas valide pour cet appel système. L'intergiciel doit décider comment traiter ce cas.

Propagation via une notification asynchrone

Le second mode d'interaction entre l'intergiciel et le système d'exploitation consiste en l'utilisation de signaux envoyés par le noyau aux processus composant l'intergiciel. Les signaux servent au contrôle de processus (suspension, arrêt, reprise) et à informer un processus de manière asynchrone qu'un événement est survenu. Ces événements peuvent être des diagnostics d'erreur (tentative d'accès à une plage mémoire non autorisée, le processus a exécuté une instruction invalide) ou la notification d'un événement ordinaire (un temporisateur a expiré, ou un processus fils est mort).

Si un processus s'est inscrit pour recevoir ce signal, il a la possibilité de réagir de manière appropriée. Le cas échéant, le processus est arrêté par le système d'exploitation. Des erreurs peuvent se propager via ce mécanisme lorsque le système d'exploitation

Lors de l'invocation d'une méthode sur un objet distant, l'intergiciel emballe les arguments de l'opération au format CDR dans un tampon `buffer`, puis envoie les données à la machine où s'exécute l'objet.
L'intergiciel fait donc un appel au système d'exploitation :

```
ret = write (fd, buffer, buffer_length);
```

La variable `fd` est un descripteur qui représente un canal de communication avec la machine distante. Dans la plupart des cas, `ret` sera positif et égal à `buffer_length`, le nombre d'octets dans `buffer`. Si `ret` est positif mais différent de `buffer_length`, il représente le nombre d'octets qui ont été effectivement écrits, et la demande d'écriture n'a pu être que partiellement accomplie. Dans ce cas, l'intergiciel doit répéter la demande de service pour les données qui n'ont pas été transmises. Ce comportement du système d'exploitation est autorisé par la spécification POSIX, qui ne garantit l'atomicité d'une opération d'entrée/sortie que pour des messages de taille inférieure à 512 octets.
La valeur de `ret` peut également être négative, ce qui permet (avec `errno`) de signaler plusieurs types d'erreur :

- `EBADF` : le descripteur de fichier `fd` est invalide ;
- `EFAULT` : l'adresse du tampon `buffer` n'est pas valide ;
- `EPIPE` : le canal de communication avec la machine distante a été fermé inopinément.

Enfin, la valeur `EINTR` de `errno` indique que l'appel système a été interrompu (par exemple par l'arrivée d'un signal), et qu'il doit être rejoué par l'intergiciel.
En conséquence, écrire des données vers un objet distant n'est pas une opération triviale. Le code ci-dessous est un exemple d'interaction robuste avec un système d'exploitation conforme à la norme POSIX :

```
int safe_write (int fd, char *buf, int len) {
  int written;

  while (len > 0) {
    written = write (fd, buf, len);
    if (written < 0) {
       if (EINTR == errno || EAGAIN == errno) continue;
       if (EFAULT == errno || EBADF == errno)
          throw INTERNAL (errno, MinorCodeWrite);
       throw COMM_FAILURE (errno, MinorCodeWrite);
      }
    buf += written;
    len -= written;
  }
  return 0;
}
```

On voit sur cet exemple qu'il est facile d'oublier certains chemins de traitement d'erreur. Cette observation fournit un argument supplémentaire en faveur de l'utilisation d'un intergiciel de communication, puisqu'il permet de réutiliser du code qui a été écrit par des spécialistes, et qui a pu être testé.

FIG. 2.2 – *Sur la difficulté à programmer de manière robuste*

lève des exceptions invalides, ou lorsqu'il les lève au mauvais instant, ou lorsque l'intergiciel ne traite pas correctement toutes les exceptions qui peuvent survenir.

Un dernier type d'interaction avec le système d'exploitation concerne la gestion des événements. L'intergiciel peut utiliser l'appel système `select` pour rester en sommeil jusqu'à ce que de l'activité soit détectée sur un ensemble de descripteurs de fichier. Si le système d'exploitation n'informe pas l'intergiciel lorsque l'activité se produit, l'intergiciel ne pourra pas traiter les requêtes entrantes.

Déroulement d'une campagne

Une campagne utilisant cette technique d'injection de fautes consiste à simuler différents types de comportements non-nominaux du système d'exploitation, et à évaluer leur impact sur le comportement de l'intergiciel. Nous verrons au § 3.5 comment nous avons implémenté l'interception d'appels système, et la liste de fautes que nous avons injectées.

Les types d'expériences qui peuvent être menées prennent alors trois formes :

- simuler des défaillances arbitraires du système d'exploitation, afin d'évaluer les mécanismes de détection d'erreur de l'intergiciel, et de caractériser ses modes de défaillance.

  Une défaillance arbitraire peut être par exemple un signalement incorrect d'erreur, qualifié de *hindering* (gêne ou entrave au bon fonctionnement des mécanismes de recouvrement d'erreur) par le projet Ballista. Par exemple, lorsque l'intergiciel demande la création d'un nouveau processus léger, on peut simuler un code retour « impossible d'accéder au fichier demandé ».
- simuler des comportements corrects mais rares du système d'exploitation, afin de vérifier que l'intergiciel les traite correctement. On parle parfois de test *off-nominal*, pour insister sur le fait que ces tests ciblent des chemins peu empruntés du système. Ces cas sont difficiles à identifier par des tests fonctionnels classiques, puisqu'ils se produisent très rarement (*c.f.* l'exemple de la figure 2.2 page précédente). D'autre part, il existe peu d'outils d'analyse du code source qui permettent d'identifier des chemins possibles, mais non traités par le programmeur.
- simuler des erreurs qui sont signalées à l'intergiciel par le système d'exploitation, et vérifier que l'intergiciel traite bien ces situations d'erreur, par exemple en levant une exception, ou en basculant vers un mode dégradé.

Ces types d'expériences permettent d'évaluer la robustesse de l'intergiciel. Cependant, les deux derniers sont les plus réalistes en pratique.

Basculement vers des modes dégradés

Nous n'attendons pas de l'intergiciel un comportement qui puisse résister à toutes les défaillances arbitraires du système d'exploitation. Toutefois, un intergiciel devrait pouvoir gérer **certains** types de défaillance du système d'exploitation de manière contrôlée et robuste. Celà implique que l'intergiciel ne va pas propager l'erreur jusqu'au niveau applicatif, mais va chercher au minimum à signaler le problème à l'application, afin que cette dernière puisse mettre en œuvre d'éventuels mécanismes de recouvrement d'erreur. Dans le cas d'un intergiciel CORBA, le mécanisme employé pour ce signalement est la levée d'une exception. L'effet au niveau applicatif d'une exception CORBA dépend de la projection langage, mais le plus souvent se traduit par une exception native du langage de programmation.

La norme CORBA précise quelles exceptions devraient être levées face à certaines conditions exceptionnelles. Par exemple, une exception CORBA de type `NO_MEMORY` doit être levée pour signaler un problème concernant l'allocation de mémoire. Une exception de type `PERSISTENT_STORE` doit être utilisée pour signaler un problème concernant la sauvegarde sur disque. L'exception `NO_RESOURCES` peut être utilisée pour signaler que le serveur ne dispose pas de suffisamment de ressources pour accomplir la requête.

L'intergiciel peut se prémunir contre certaines erreurs de ressource en pré-allouant certaines ressources à l'initialisation. Cette technique est utilisée par le serveur web *Apache*, par exemple. Lorsqu'une erreur de ressources survient malgré ces précautions, l'intergiciel devrait la traiter de manière contrôlée. En particulier, dans un objet serveur, une erreur de ressources transitoire qui affecte une interaction avec un client ne devrait pas affecter les interactions avec d'autres clients.

Travaux apparentés

Pour des applications sur le système d'exploitation Microsoft Windows, l'outil *Holodeck*[5] intercepte les appels système effectués par une application cible, et permet de simuler différents types d'erreur, tels que des allocations mémoire qui échouent, des demandes d'accès à la base de registre, ou l'échec d'opérations d'accès au réseau de communication. L'outil permet de déclencher l'injection de la faute en fonction de plusieurs critères.

Pour des systèmes Unix, le projet *FIG* (*Fault Injection in glibc*) à l'Université de Berkeley consiste à simuler la défaillance d'appels de service à la glibc (la bibliothèque système utilisée sur les machines GNU/Linux). Le projet a permis d'identifier des défauts de robustesse dans des applications largement déployées, tel que le serveur web *Apache* et l'éditeur de texte *GNU Emacs*.

[5] L'outil *Holodeck*, initialement développé au *Florida Institute of Technology*, est désormais distribué par la société *SecurityInnovation* (`www.sisecure.com`).

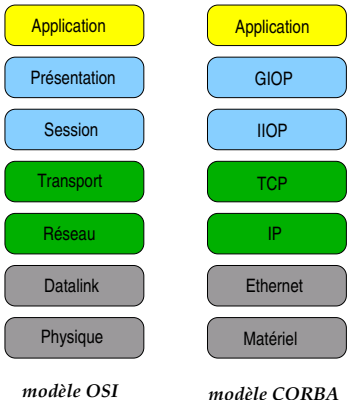

FIG. 2.3 – *Couches de protocole réseau dans* CORBA

2.2.5 Techniques de corruption de messages

Ce modèle de fautes permet d'évaluer l'effet de fautes qui touchent les services de communication du système. Il s'agit sans doute du modèle de fautes le plus spécifique aux intergiciels orientés communication, puisqu'il permet d'évaluer à quel point de nouvelles classes de fautes, provenant de la répartition, peuvent affecter la sûreté de fonctionnement du système réparti.

Les expériences utilisant ce modèle de fautes consistent à faire en sorte que la cible soit affectée par une faute réseau. Nous avons provoqué des erreurs qui affectent trois niveaux distincts de la pile de protocole OSI [6] (*c.f.* la figure 2.3) :

- couche réseau : simulation de l'interruption de connexions TCP vers la cible ;
- couche session : faire en sorte que la cible reçoive des requêtes qui ont été corrompues au niveau IIOP ;
- couche présentation : la cible reçoit une requête qui contient une corruption au niveau GIOP.

La première classe d'erreurs simule l'effet de défaillances plutôt grossières d'éléments du réseau de communication : défaillance d'un routeur, débranchement d'un câble réseau. Dans la mesure où les systèmes que nous ciblons n'intègrent pas de mécanisme spécifique de tolérance aux fautes, ce type d'erreur a une très forte probabilité d'entraîner la défaillance du système. Pour ce type de faute, il est surtout intéressant

[6] La norme CORBA définit un ensemble de couches de communication, qui ont une correspondance naturelle avec les couches du modèle OSI. Le modèle OSI (pour *Open Systems Interconnect Reference Model*, ou interconnexion de systèmes ouverts) a été proposé par l'ISO pour normaliser la communication entre systèmes informatiques. Ce modèle définit sept couches, allant du niveau physique au niveau de l'application.

de mesurer la latence de détection d'erreur, autant du côté de la cible, que du côté des clients de la cible. Les travaux de Chung [Chung et al. 1999] que nous avons analysés à la page 29 ont évalué l'impact de ce type de faute.

Les deux classes suivantes simulent des erreurs plus pernicieuses, puisqu'il s'agit d'erreurs qui devraient être détectées par le sous-système de communication. En effet, la norme CORBA fait l'hypothèse que le protocole de transport servant à la communication entre objets est fiable [OMG 2002b].

La corruption de messages réseau, même sur des protocoles de transports fiables tels que TCP, est plus fréquente qu'on ne le croit. En s'appuyant sur l'analyse de trafic réseau sur plusieurs types de machines connectées à un réseau local et sur Internet, l'étude [Stone & Partridge 2000] rapporte que environ un paquet sur 32000 présente une checksum TCP invalide. Les auteurs estiment que le nombre de paquets délivrés à l'application par la pile TCP/IP comme étant valides alors qu'en vérité ils ont été corrompus, atteint des taux variant entre un sur quelques millions et un sur 10 milliards (ce taux varie considérablement en fonction de la qualité des cartes réseau utilisées). Ceci est dû au fait que la checksum TCP, qui ne fait que 16 bits, ne peut pas détecter certaines erreurs.

✠$\mathcal{A}$ — Checksums, CRCs et erreurs

La *checksum* d'un bloc de données est un code qui permet de détecter une erreur qui serait survenue au cours de la transmission des données. La checksum est calculée en prenant une somme des éléments du bloc. De par sa construction, une checksum ne peut pas détecter certains types d'erreur : le réordonnancement de plusieurs octets dans un message, l'insertion ou la suppression d'éléments nuls, et les erreurs multiples dont l'effet sur la checksum s'annule. Un code *CRC* ou *Cyclic Redundancy Check* est une valeur entière calculée par une opération de division binaire des données par un polynôme générateur. Un CRC a un pouvoir de détection d'erreur meilleur qu'un checksum de la même taille.

Dans la pile de protocoles qui nous intéresse, le support physique Ethernet utilise un CRC de 32 bits ; la couche IPv4 comporte une checksum de 16 bits sur les données de l'en-tête IP ; la couche TCP ajoute une checksum de 16 bits couvrant l'en-tête TCP et les données. Malgré cet empilement de mécanismes de détection, quelques rares erreurs peuvent atteindre le niveau applicatif.

Sur un réseau local à 100 Mb/s avec un taux d'utilisation moyen de 10%, et en supposant que la taille moyenne des paquets TCP est de 400 octets, un taux de 1 sur 100 millions donne un MTBF d'environ neuf heures ; il s'agirait donc d'une source non négligeable d'erreurs non intentionnelles.

Les corruptions au niveau de la couche session que nous simulons peuvent être provoquées par trois classes de faute sous-jacentes :

- des *fautes physiques transitoires* dans les éléments matériels servant à la communication entre les nœuds. On peut citer par exemple la défaillance de composants mémoire dans les routeurs, ou des fautes lors de transferts DMA avec la carte réseau.
- la propagation vers le système cible d'une faute qui a affecté un nœud distant qui interagit avec la cible. La faute sur le système distant a pu affecter son système d'exploitation, sa pile de protocole ou la couche intergiciel, et a conduit à l'émission, par le système distant, d'une requête corrompue vers la cible.
- les *fautes malicieuses*, telles que des attaques en déni de service visant le système cible. Étant donné le rôle pivot d'éléments de l'intergiciel tels que le service de désignation (*c.f.* § 2.1.3), un attaquant qui réussit à provoquer l'arrêt du service peut entraîner la défaillance du système complet. Ce type d'attaque est fréquent sur des réseaux publics comme Internet. Toutefois, la plupart des systèmes semi-critiques basés sur l'intergiciel sont déployés sur des réseaux privés, où tous les intervenants sont supposés dignes de confiance.

  Notons également que les erreurs que nous injectons dans nos expériences (des inversions de bits et la mise à zéro de séquences d'octets) ont peu de chance d'être représentatives des activités d'un utilisateur malicieux, qui fera subir au système des séquences d'entrées invalides plus complexes. Toutefois, nos expériences donnent une limite inférieure au degré de perturbation que peut provoquer un utilisateur malicieux du système.

Par ces expériences de corruption de messages, nous évaluons la robustesse de l'intergiciel vis-à-vis de ce type d'entrées invalides. Ces expériences nous permettent également d'évaluer la probabilité de propagation d'erreur entre un client et un serveur, c'est-à-dire de savoir à quel point l'intergiciel constitue un canal de propagation d'erreur.

Réception de messages corrompus

Ces expériences peuvent étudier l'impact de différents types d'erreurs. La corruption la plus utilisée historiquement dans le contexte d'injection de fautes ciblant la mémoire est l'inversion de bits. Deux types d'erreurs sont particulièrement intéressants à étudier : le cas d'inversions isolées de bits, et la mise à zéro de deux octets successifs dans un message. Ces deux formes de corruption sont parmi les plus fréquemment observées dans l'étude [Stone & Partridge 2000] citée précédemment, et nous faisons l'hypothèse qu'ils sont représentatifs de la propagation d'erreur depuis des nœuds distants.

Ces fautes peuvent être injectées de différentes manières. Une possibilité serait d'utiliser du matériel réseau conçu spécialement pour cette tâche, comme le font [Floering et al. 2002], mais ceci est coûteux à mettre en œuvre, et

offre peu de souplesse à l'utilisation. En utilisant des techniques d'injection de fautes par logiciel, les fautes pourraient être injectées au niveau de la couche de transport (par exemple en instrumentant la pile de protocole du système d'exploitation [Dawson & Jahanian 1995]). Cependant, en injectant à ce niveau il y a une forte probabilité que la corruption des messages soit détectée par la pile TCP du nœud distant, et donc que le message ne soit pas délivré à la couche intergiciel. Il est donc plus efficace d'injecter la faute au niveau applicatif, avant que les données ne soient encapsulées par la couche de transport.

En injectant à ce niveau, on simule des fautes du logiciel qui surviennent chez le client au niveau de l'application, de l'intergiciel ou du système d'exploitation distant. On simule également la très faible proportion de paquets qui subissent une corruption que le protocole TCP n'est pas en mesure de détecter, et qu'il délivre comme s'ils étaient valides. [Dans ce dernier cas, nous faisons l'hypothèse simplificatrice que les types de corruption qui ne sont pas détectés par le checksum TCP sont distribués uniformément.]

Un point positif de cette technique d'injection de fautes est qu'elle est très portable, puisque le seul élément dans notre banc de test qui soit spécifique à une implantation CORBA donnée est le code qui est responsable du démarrage de la cible. La technique peut même être appliquée, sans grand effort d'adaptation, à des cibles non-CORBA, puisque la nature de la faute que nous injectons ne dépend pas de particularités du protocole de transport utilisé dans CORBA. D'autre part, cette technique est non intrusive, puisqu'elle ne nécessite aucune instrumentation du service cible.

Fautes de protocole réseau

Outre l'intérêt de considérer des fautes qui simulent la corruption des messages, il peut être intéressant d'étudier l'impact de fautes de plus haut niveau, qui affectent la sémantique des messages plutôt que leur contenu syntaxique. Dans un intergiciel de type CORBA, ces fautes concernent le protocole GIOP.

L'objectif de ce type d'expériences est d'évaluer la robustesse d'une implantation vis-à-vis de requêtes invalides ou incohérentes provenant de nœuds distants. Citons quelques exemples d'incohérence que nous pourrions simuler expérimentalement :

- la réception de messages GIOP inattendus. Par exemple, il existe un message de type `LocateReply` dans le protocole GIOP, qui est envoyé en réponse à un message de type `LocateRequest` (ce message sert aux courtiers à localiser les objets). Un exemple de faute injectée consisterait à envoyer un message de type `LocateReply` à un objet, sans qu'il ait émis le message `LocateRequest` correspondant.

- la réception de messages GIOP qui contiennent des identifiants de requête inattendus. Chaque message GIOP contient un identifiant numérique de requête, qui est alloué par le courtier client. Cet identifiant permet d'associer une réponse à la requête à laquelle il répond (ceci est nécessaire lorsque plusieurs connexions au niveau GIOP sont projetées sur une seule connexion gérée par la couche de transport). Une injection de faute consisterait à envoyer à la cible un message de type `RequestReply`, contenant un identifiant de requête qui ne correspond pas à une requête qui a été émise par ce client. Un autre type d'injection consisterait à évaluer le comportement du courtier lorsqu'il reçoit plusieurs messages portant des identifiants de requête identiques (il devrait ignorer les messages contenant des identifiants qu'il a déjà traités).
- la réception de messages GIOP dont l'identifiant d'objet contient un contexte de requête qui est anormal. Le contexte de la requête est un champ qui peut être utilisé par le courtier afin de propager des informations liées spécifiquement à la requête (par exemple, une clé de session, ou l'identifiant du jeu de caractères qui a été négocié lors de l'établissement de la connexion réseau). Le courtier devrait ignorer tous les éléments du contexte de requête qu'il ne sait pas reconnaître.
- l'utilisation d'options de négociation inhabituelles, lors de l'établissement d'une connexion GIOP. Par exemple, des jeux de caractères inconnus peuvent être proposés par un courtier, pour vérifier que le courtier cible les ignore.
- l'utilisation d'adresses réseau inhabituelles. Citons comme exemples l'utilisation de noms DNS très longs, l'utilisation de noms IPv6, de ports TCP en dehors de la plage généralement autorisée.
- des messages fragmentés : la version 1.2 du protocole GIOP permet aux clients d'émettre des messages *fragmentés*, c'est-à-dire décomposés en plusieurs messages de la couche transport. Cette possibilité est avantageuse lorsqu'il s'agit de transférer de gros volumes de données, puisque le courtier peut éviter de stocker de gros tampons en mémoire. Cependant, le traitement de ces requêtes côté récepteur est plus compliqué, puisque le courtier est responsable du réassemblage des messages provenant de la couche transport.

Le comportement attendu de la cible est de répondre par un message GIOP de type `MessageError`, ou simplement de fermer la connexion réseau par laquelle le message corrompu lui est parvenu. L'erreur ne devrait pas affecter les interactions de la cible avec d'autres clients.

2.3 Récapitulatif

Dans ce second chapitre, nous avons présenté la démarche que nous proposons pour la caractérisation du comportement d'intergiciels de communication en présence de fautes. Notre approche est expérimentale, et basée sur l'utilisation de plusieurs techniques d'injection de fautes.

La méthodologie que nous avons développée s'articule de la manière suivante :

Identifier les frontières du système cible et identifier des interfaces permettant l'activation de ses fonctionnalités. Le service rendu par un intergiciel de communication consiste à mettre en œuvre un modèle de programmation dans lequel un objet peut communiquer avec ses pairs sans se préoccuper de leur localisation, du langage de programmation employé ni de l'environnement d'exécution. Afin de maintenir cette illusion de localité, l'intergiciel s'immisce dans le code applicatif pour intercepter les événements qu'il va dérouter vers des nœuds distants. Ce fort degré d'intrusivité, et la variation dans les techniques d'implantation utilisées par différents fournisseurs, fait qu'il est difficile de démarquer une frontière claire autour de l'intergiciel, et rend délicate la maîtrise de l'activation de ses fonctionnalités.

Nous avons contourné cet aspect « difforme » de l'intergiciel en prenant pour cible les services CORBA tels que la désignation et la diffusion d'événements. Ces services présentent une interface bien définie que nous pouvons cibler dans nos expériences d'injection de fautes. Dans la mesure où les services s'appuient sur des courtiers, cibler les services permet de couvrir de manière satisfaisante les fonctionnalités de courtier d'un intergiciel.

Il est également nécessaire d'inclure les outils de développement d'un fournisseur dans une campagne expérimentale, puisque les souches et les squelettes qu'ils génèrent sont intégrés aux objets applicatifs du système.

Nous avons proposé une technique permettant de construire un ensemble de systèmes « synthétiques » basés sur un intergiciel, qui permettent d'activer les fonctionnalités des outils de développement et de courtier d'un fournisseur donné.

Établir un modèle de fautes qui soit représentatif des fautes qui peuvent affecter un système réparti s'appuyant sur un intergiciel. En l'absence d'études approfondies sur la pathologie des fautes dans les systèmes CORBA, nous avons identifié des classes de fautes à partir d'une analyse structurelle des architectures de systèmes à base d'intergiciel de communication. Les fautes identifiées comprennent des fautes physiques transitoires entraînant une corruption de la mémoire, des erreurs de communication, des fautes du logiciel dans les courtiers et dans les outils de développement des fournisseurs, et des perturbations dans les interactions exécutif $\leftrightarrow$ intergiciel et intergiciel $\leftrightarrow$ application.

Classifier les modes de défaillance et leur associer des techniques de détection expérimentaux. Nous avons identifié les différentes classes de défaillances qui peuvent affecter le courtier, les différents services CORBA, et les outils de développement fournis avec l'intergiciel. Dans le chapitre suivant, nous verrons quels sont les mécanismes mis en œvre dans le banc de test pour permettre toutes les observations nécessaires à l'identification et la classification des manifestations de fautes.

Proposer des techniques de caractérisation permettant d'émuler chaque classe de fautes précédemment identifiée, et évaluer son impact sur le système. Les fautes physiques transitoires peuvent être émulées à l'aide de techniques de corruption par logiciel de plages mémoire utilisées par différents composants de l'intergiciel. Certaines fautes logicielles peuvent être étudiées en utilisant des techniques de mutation de code, qui introduisent artificiellement des erreurs dans le code source du composant cible afin d'étudier le comportement du mutant résultat.

Quant aux fautes externes, l'impact de fautes qui atteignent l'intergiciel via son interaction avec le niveau applicatif peut être étudié à l'aide de techniques de test de robustesse, consistant à effectuer des appels invalides à l'intergiciel. Les fautes qui atteignent l'intergiciel via son interaction avec le système d'exploitation peuvent être simulées à l'aide de techniques d'interception d'appels système. Nous proposons plusieurs expériences où le comportement du système d'exploitation ne correspond pas à celui attendu par l'intergiciel. Enfin, les fautes qui atteignent l'intergiciel dans des requêtes provenant d'objets situés sur d'autres nœuds sont étudiées à l'aide de techniques de corruption de messages.

Au chapitre suivant, nous présentons les résultats expérimentaux obtenus en appliquant cette méthode de caractérisation à un certain nombre d'implémentations CORBA sur étagère.

Chapitre 3

Résultats Expérimentaux

Les grandes personnes aiment les chiffres. Quand vous leur parlez d'un nouvel ami, elles ne vous questionnent jamais sur l'essentiel. Elles ne vous disent jamais : « Quel est le son de sa voix ? Quels sont les jeux qu'il préfère ? Est-ce qu'il collectionne les papillons ? » Elles vous demandent : « Quel âge a-t-il ? Combien a-t-il de frères ? Combien pèse-t'il ? Combien gagne son père ? » Alors seulement elles croient le connaître.

— *Le Petit Prince*, Antoine de Saint Exupéry

DANS ce chapitre, nous présentons les résultats d'expériences que nous avons menées afin de valider la méthode de caractérisation présentée dans le chapitre précédent. Nous commençons par décrire les cibles sur lesquelles nous avons travaillé ; il s'agit principalement d'implantations de la plate-forme CORBA. Nous décrivons ensuite notre banc de test expérimental, les différents événements significatifs que nous avons mesurés et notre classification des manifestations de fautes. Dans nos expériences, nous nous sommes particulièrement intéressés à trois aspects importants d'une implémentation CORBA : son service de désignation, son service de diffusion d'événements et ses fonctionnalités de courtage.

Nous poursuivons en présentant les résultats d'expériences d'injection de faute obtenus en appliquant les différentes méthodes introduites dans le chapitre précédent. Nous avons focalisé notre travail sur deux méthodes d'injection de faute : les corruptions réseau, qui nous paraissaient particulièrement pertinentes pour des intergiciels de communication, et la propagation d'erreurs depuis l'exécutif, méthode de caractérisation qui a été peu traitée dans la littérature. Nous présentons également quelques résultats d'expériences de test de robustesse de compilateurs IDL, basées sur la mutation d'interfaces OMG IDL.

Dans le chapitre 4 nous analyserons ces résultats de manière comparative, en présentant des commentaires plus généraux.

3.1 Cibles et contexte expérimental

La plate-forme CORBA est l'intergiciel le plus utilisé par les intégrateurs de système à fort besoin en sûreté de fonctionnement. La norme existe en effet depuis plus d'une décennie, et les implantations disponibles sur le marché sont de qualité satisfaisante. Nous avons donc choisi de concentrer notre travail expérimental sur ce type de plate-forme standardisé. Toutefois, nous verrons dans la § 4.5.2 que notre approche devrait facilement s'adapter à d'autres types d'intergiciel.

Nos expériences ont ciblé les implémentations suivantes de la norme CORBA :

- *ORBacus* pour C++, une implantation commerciale qui a été développée par la société *OOC*. Bien qu'elle soit commerciale, ses codes sources sont disponibles. Cette société a été rachetée par la société *IONA*, qui commercialise désormais le produit. Nous avons testé la version 4.1.1.
- *TAO* pour C++, une implantation sous licence libre, développée initialement à l'Université de Washington, et désormais conjointement par l'Université de Californie à Irvine et par la société *Object Computing, Inc.* (qui propose également un support commercial pour le produit). Cette implantation a pour particularité de fournir un certain nombre d'extensions concernant le fonctionnement temps réel, telles que des aspects de la norme *RT-CORBA*. Nous avons testé la version 1.2.1.
- *omniORB* pour C++, une implantation sous licence libre, initialement développée par les laboratoires de Cambridge de *AT&T*. Nous avons testé la version 4.0.
- *MICO* pour C++, une implantation sous licence GNU GPL. Nous avons testé la version 2.3.6.
- *ORBit* pour C, une implantation sous licence libre. Cette implantation a pour particularité d'avoir été destinée à fonctionner au cœur du desktop *GNOME*. Elle privilégie donc une surface de code réduite et la rapidité. Nous avons testé la version 0.5.17.
- les courtiers inclus avec les machines virtuelles Java suivantes :
  - JDK 1.3.1 de *Sun Microsystems*,
  - JDK 1.4.1 de *Sun Microsystems*,
  - JDK 1.4 de *IBM*.

  Depuis la version 1.3 de la norme Java, les environnements d'exécution Java doivent fournir une implémentation CORBA. Ces environnements fournissent par conséquent un compilateur IDL qui implémente la projection langage pour Java, une fonctionnalité de courtier, et une implantation du service de désignation.
- *OpenORB* pour Java. Ce courtier, distribué sous forme d'empaquetage de classes Java (un fichier au format jar), est destiné à remplacer celui fourni avec les machines virtuelles Java. Cette implantation offre également une suite très complète de services CORBA. Nous avons testé la version 1.3.0.

- *JORBacus* pour Java, une implantation commerciale développée par la société *OOC* et désormais distribuée par *IONA*. Ce produit comporte simplement l'aspect courtier et projection langage. Nous avons testé la version 4.1.2.
- *Fnorb* pour le langage Python, une implantation sous licence GNU GPL. Nous avons testé la version 1.3.
- *Visibroker*, une implantation commerciale pour C++ de la société Inprise.
- *OrbixWeb*, une implantation commerciale pour C++ et Java de la société *IONA*.

Les candidats implémentés en C et C++ et distribués sous forme de code source ont été compilés avec la version 2.95.4 du compilateur GCC.

Nos expériences ont ciblé différents composants fournis par ces implémentations :

- certains outils de développement d'un fournisseur : le compilateur OMG IDL, et le référentiel d'interfaces CORBA ;
- le service de désignation ;
- le service de diffusion d'événements ;
- le courtier, instancié à l'aide du service d'écho (service synthétique).

Dans la suite du chapitre, nous désignons le composant cible par le terme SUT, pour *System Under Test*.

3.1.1 Configuration du banc de test

Le banc de test que nous avons mis en place pour nos campagnes comprend les éléments suivants :

- des processus constituant la *charge applicative*, dont le rôle est d'activer les fonctionnalités du SUT. Dans certains cas, nos processus de charge jouent également le rôle de l'oracle, en détectant en ligne d'éventuelles déviations du comportement du SUT vis-à-vis de son comportement attendu. Nous donnerons davantage de détails sur les processus de charge lorsque nous décrirons en détail chaque campagne d'injection.
- l'*injecteur*, qui est responsable de l'injection de la faute. Dans la majorité de nos expériences, le déclenchement de l'injection est temporel.
- des *composants d'observation*, qui enregistrent toutes les informations utiles sur le déroulement d'une expérience (les exceptions CORBA qui se produisent, l'arrêt inopiné de processus). Ces informations sont enregistrées dans une base de données SQL.
- un *lanceur* sur chaque nœud, qui reste à l'écoute des demandes de lancement de processus.

- des scripts de contrôle d'expérience, qui prennent en paramètre un identifiant de faute à injecter (la position dans le message, par exemple). Le script lance la cible puis les processus de charge, attend un certain temps, déclenche l'injecteur, attend encore 30 secondes, puis arrête l'expérience.
- des scripts de contrôle de campagne, qui choisissent le mode d'injection et la cible, et exécutent séquentiellement le script de contrôle d'expérience pour chaque position possible de faute.
- les outils d'analyse hors ligne, qui traitent les données produites par les composants d'observation. Ces outils produisent alors des statistiques qui portent sur une campagne entière.

Nos expériences ont été menées sur un rack de machines à base de processeurs i686, dotées de 512 mégaoctets de mémoire principale. Les machines sont inter-connectées via un réseau Ethernet à 100 mégabits/seconde. Les machines utilisent le noyau Linux version 2.4.21, et leurs horloges sont synchronisées à l'aide du protocole NTP.

3.1.2 Événements significatifs lors d'une expérience

Les événements et les manifestations de faute que nous pouvons observer au cours d'une expérience sont les suivants :

- Arrêt inopiné d'un nœud : un nœud sur lequel s'exécute un ou plusieurs objets CORBA devient inaccessible depuis les autres nœuds. On teste cette condition en essayant d'exécuter une commande système depuis un nœud distant.
- Arrêt inopiné d'un processus ou d'un processus léger. Dans nos expériences, cet événement est observé localement sur chaque nœud par le lanceur. Le lanceur est averti immédiatement lorsque l'un de ses processus fils meurt ; nous pouvons donc enregistrer la date précise de la mort du processus.
- Gel d'un processus ou d'un processus léger : le SUT accepte une demande de connexion, mais ne renvoie pas de réponse avant un temps prédéfini, qui est largement supérieur au temps de réponse nominal.
- Propagation de l'erreur au niveau applicatif : l'erreur insérée dans l'intergiciel a entraîné une erreur au niveau applicatif. Dans nos expériences, cette manifestation de faute est généralement observée par les processus de charge, qui sont conçus de façon à pouvoir détecter un comportement erroné du SUT.
- Levée d'une exception CORBA : lors d'une invocation de méthode, le client reçoit une exception plutôt qu'une valeur de retour. Nous distinguons les exceptions de type système (les `SystemException`, qui sont levées par l'intergiciel) des exceptions de type utilisateur (les `UserException`, qui sont levées au niveau applicatif).

Ces manifestations de fautes ne sont pas mutuellement exclusives. Lors d'un arrêt inopiné du SUT, par exemple, les clients de ce service reçoivent généralement une exception leur indiquant qu'un problème de communication a été rencontré. On tiendra compte de ce fait lors des analyses des résultats des campagnes d'injection de faute : dans certains cas, on ne retiendra que l'événement le plus grave observé pour chaque expérience, et dans d'autres nous nous concentrons sur le premier événement observé.

D'autre part, nous enregistrons des informations complémentaires concernant chaque expérience : le temps CPU consommé par les processus constituant l'application de charge, le temps CPU consommé par le SUT, et la quantité de travail effectuée par l'application de charge. Ces informations permettent d'analyser l'impact de la faute sur la performance du système, et de détecter un gel applicatif.

Notre banc de test nous permet également d'enregistrer d'autres informations sur le déroulement des expériences, comme la trace de la pile d'exécution en cas d'arrêt inopiné du SUT, la liste des messages d'avertissement imprimés par le SUT, la trace des appels système effectués par le SUT. Nous pouvons également capturer les messages réseau reçus et transmis par le SUT. Ces informations sont utiles pour le diagnostic des défaillances observées, comme nous le verrons au chapitre suivant.

À l'issue de chaque expérience, nous analysons les manifestations de faute observées, et établissons une catégorisation des expériences en fonction des types de ces manifestations. Cette classification (présentée dans la figure 3.1 page suivante) n'est pas disjointe, et toutes les catégories ne s'appliquent pas à toutes les expériences que nous avons menées. Nous verrons les adaptations et la classification disjointe que nous avons utilisées dans les sections suivantes.

3.2 Impact de corruptions IIOP sur le service de désignation

Dans cette section, nous présentons les résultats d'expériences qui ont cherché à évaluer l'impact de fautes qui atteignent l'intergiciel par le réseau de communication. Plus précisément, nous simulons l'impact de messages inter-courtier qui ont subi une corruption au niveau de la couche IIOP (*c.f.* § 2.2.5). Ces résultats sont une extension des expériences rapportées dans [Marsden et al. 2002c].

Notre cible dans cette campagne est le service de désignation CORBA, qui fournit un répertoire hiérarchique pour les références objet, permettant à des applications serveur d'annoncer la disponibilité d'un objet sous un nom symbolique, et aux clients d'obtenir une référence objet en résolvant ce nom. Ce service joue généralement un rôle central dans un système réparti, constituant de fait un point dur ; il est donc important pour un intégrateur de système d'être sûr que son implantation fonctionnera bien en présence de fautes.

NonActivation	la faute injectée n'a pas été activée
ApplicationFailure	l'erreur qui a affecté l'interaction entre le SUT et un client n'a pas été détectée au niveau de l'intergiciel, et s'est propagée au niveau applicatif
RemotePropagation	l'erreur qui a affectée l'interaction entre le SUT et un client s'est propagée à d'autres clients du SUT. L'intergiciel a donc servi de canal de propagation d'erreur entre deux clients.
KernelCrash	défaillance par arrêt du système d'exploitation sur un nœud
KernelHang	défaillance par gel du système d'exploitation sur un nœud
ServiceCrash	défaillance du SUT par arrêt inopiné. Dans nos expériences, l'arrêt du SUT a toujours été accompagné d'une détection d'erreur dans les processus de charge
ServiceHang	défaillance du SUT par gel, qui affecte tous les clients
ConnectionHang	gel d'une seule connexion avec le SUT, qui n'affecte qu'un seul client
DetectionException	l'erreur est détectée par la levée d'une exception CORBA dans l'injecteur. Elle ne s'est pas propagée à l'application de charge
ActionRecouvrement	l'erreur a été détectée par le SUT, qui a déclenché une action de recouvrement d'erreur
NonObservation	tous les autres cas

FIG. 3.1 – *Classification des manifestations de faute*

3.2.1 Configuration expérimentale

L'application de charge que nous avons développée pour le service de désignation a un comportement cyclique qui consiste à construire un arbre de nommage de profondeur 100, à résoudre des noms à différentes hauteurs dans l'arbre, puis à détruire l'arbre. Puisque l'arbre est créé de manière déterministe, l'application de charge est capable de vérifier en ligne la validité des résultats fournis par le SUT. Lorsqu'un élément de la charge détecte une anomalie dans le fonctionnement du SUT, par exemple une résolution de nom qui ne renvoie pas la bonne référence[1], elle signale aux compo-

[1]Le modèle à objets de CORBA n'autorise pas le test d'égalité entre références objets. Nous ne pouvons donc pas détecter des réponses invalides du service de désignation en effectuant une simple comparaison entre la référence liée et celle renvoyée par le SUT. Afin de contourner ce problème, nous utilisons des objets contenant des identifiants uniques, et effectuons la vérification d'égalité sur cet identifiant.

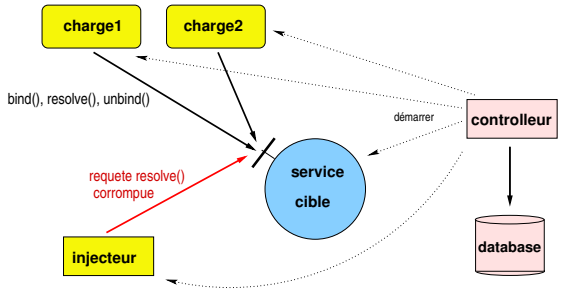

FIG. 3.2 – *Configuration des expériences ciblant le service de désignation*

sants d'observation qu'une défaillance applicative s'est produite. De la même façon, lorsqu'elle reçoit une exception provenant du SUT, elle en informe les composants d'observation afin que cet événement soit enregistré, avec sa date d'occurrence.

La configuration expérimentale pour nos expériences ciblant le service de désignation est illustrée dans la figure 3.2.

Dans nos campagnes d'injection de faute, chaque expérience correspond à l'injection d'une seule faute. Un processus de contrôle démarre le SUT et obtient sa référence objet. Il démarre alors les processus de charge, en leur fournissant la référence du SUT. Après un certain laps de temps (distribution aléatoire autour de 20 secondes), le composant injecteur envoie une requête corrompue au service, en lui demandant de résoudre un nom qui a été précédemment lié. L'injecteur attend alors la réponse du service, et vérifie que la réponse est correcte. Si aucune réponse n'arrive dans un délai de 30 secondes, un mode *ConnectionHang* est signalé aux processus d'observation. À la fin de l'expérience, les processus d'observation vérifient que la machine cible fonctionne correctement (en cherchant à exécuter une commande sur la machine), que le SUT répond encore aux requêtes des clients, et que les processus de charge fonctionnent encore. Chaque expérience dure ainsi environ deux minutes.

L'injecteur de faute dans ces expériences est un programme qui prend en paramètre la référence objet du SUT et un entier identifiant la position de la faute à injecter. Le comportement de l'injecteur est le suivant :

1. établir une connexion réseau avec le SUT ;
2. vérifier que le SUT joue bien le rôle de service de désignation, en lui envoyant une requête `_is_a` pour l'interface `IDL:omg.org/CosNaming/NamingContext:1.0`[2] ;
3. lier le nom `ABCDE.FGH` au contexte racine du service de désignation (opération `rebind`) ;

[2]Cette méthode `_is_a` est généralement invoquée de manière transparente par un courtier à l'occasion de la première utilisation d'un IOR. Elle permet au courtier d'assurer que le type de l'objet invoqué est compatible avec celui souhaité par le client.

4. demander au service de résoudre le nom `ABCDE.FGH` (opération `resolve`). C'est ce message qui subit une corruption.
5. vérifier que la référence renvoyée par le SUT joue le rôle de service de désignation, c'est à dire qu'elle identifie la même interface que celle liée à l'étape 3.

Nous avons injecté deux types de faute : des inversions de bits et des mises à zéro de deux octets successifs dans le message IIOP. Ces types d'erreurs ont été choisis puisqu'ils sont parmi les plus fréquemment observés dans l'étude de Stone et Partridge [Stone & Partridge 2000] que nous avons déjà mentionnée.

✠ℬ — Taille d'une requête CORBA en fonction du courtier

On pourrait supposer qu'une même requête CORBA, pour la même opération sur la même interface, ait la même représentation au niveau de la couche transport, indépendamment du courtier utilisé. En pratique il n'en est rien : la taille d'une requête CORBA, lorsqu'elle est transmise à la couche transport, varie en fonction des courtiers utilisés côté client et côté serveur. Pour comprendre pourquoi, considérons les éléments qui composent une requête CORBA :

- un en-tête, qui contient la signature GIOP, le numéro de version du protocole, et le type GIOP du message (`RequestMessage`, `ReplyMessage`, `ErrorMessage`, *etc.*) ;
- la taille de la requête en nombre d'octets, codée sur 32 bits ;
- l'identifiant de la requête, un entier non signé codé sur 32 bits ;
- le « service context », qui permet au courtier client de transmettre des informations concernant le contexte dans lequel s'effectue la requête ;
- la clé de l'objet invoqué (l'« object key ») ;
- le nom de l'opération invoquée ;
- les arguments de la requête.

L'en-tête GIOP et la taille du message sont de longueur fixe, mais la taille des autres éléments est variable. Pour une « même » requête, le nom de l'opération et les arguments ont une longueur fixe. Le contenu du *service context* est déterminé par le courtier côté client. La taille de la clé de l'objet invoqué dépend du courtier qui héberge cet objet côté serveur.

Lorsque plusieurs servants partagent la même capsule (par exemple, le même processus Unix), la clé (une séquence d'octets dont le format n'est connu que du courtier qui la génère) permet d'identifier à quel servant est destinée la requête. La clé d'un objet est générée par le courtier serveur, et fait partie de toute IOR le désignant. Tous les courtiers ne génèrent pas les clés de la même façon, et en particulier les clés n'ont pas toutes la même longueur. En conséquence, la « même » requête sur le même objet aura une longueur qui varie en fonction du courtier qui héberge cet objet.

3.2.2 Distribution des manifestations de fautes

La classification générale des manifestations de fautes présentée dans la figure 3.1 page 74 s'instancie de la manière suivante pour ces expériences :

- Une propagation d'erreur (manifestation de *ApplicationFailure* ou de *RemotePropagation*) correspond soit à la levée d'une exception « pas de référence pour ce nom » (`NotFound`), soit à une résolution de nom qui renvoie une référence incorrecte.
- Pour identifier les expériences où la faute injectée n'a pas été activée, il faudrait détecter si la partie corrompue du message IIOP a été déballée par le SUT, ce qui voudrait dire que la donnée au format CDR a été transformée en une donnée au niveau applicatif. Observer explicitement ce type de non-activation nécessiterait un fort degré d'instrumentation du SUT, ce qui ne correspond pas à notre objectif de caractérisation de type boîte noire ou boîte grise. Nous assimilons donc les *NonActivations* à des manifestations de *NonObservation*.
- Nous n'avons pas de catégorie pour les expériences avec une action de recouvrement d'erreur. Les intergiciels que nous ciblons comportent des mécanismes de recouvrement très basiques, qui se limitent à la retransmission d'invocations qui ont provoqué la levée d'une exception. La responsabilité d'une action de recouvrement est donc reportée chez le client. Comme nos expériences visent à caractériser le comportement d'un serveur, nous n'avons pas défini de catégorie associée à un éventuel recouvrement chez le client.

La classification disjointe de chaque expérience dans une des catégories de manifestation de faute se fait en testant séquentiellement les conditions suivantes :

1. si l'un des processus composant la charge de travail a constaté un comportement incorrect du SUT, l'expérience est classifiée en *RemotePropagation*. Dans ce cas, la faute qui a affectée l'interaction entre le client injecteur et le SUT s'est propagée via l'intergiciel à d'autres clients du SUT, s'exécutant sur d'autres nœuds du système, donc le SUT a servi de canal de propagation d'erreur.
2. si le nœud hébergeant le SUT s'est arrêté ou s'est gelé, classification respectivement en *KernelCrash* ou *KernelHang* ;
3. si le SUT s'est arrêté à la fin de l'expérience (arrêt inopiné du processus hébergeant le SUT, ou refus de réponse à une demande de service à la fin de l'expérience), l'expérience donne une manifestation de *ServiceCrash* ;
4. si le client injecteur a constaté un dysfonctionnement du SUT, l'expérience est classifiée en *ApplicationFailure* ;
5. si à la fin de l'expérience le SUT ne répond pas à une demande de service, il s'agit d'une manifestation de *ServiceHang* ;
6. si le client injecteur n'a pas reçu de réponse du SUT avant la fin de l'expérience, l'expérience est classifée en *ConnectionHang* ;

7. si le client injecteur a reçu une exception au cours de son interaction avec le SUT, classification en *DetectionException* ;
8. si aucune des précédentes conditions n'est vérifiée, il s'agit d'une expérience avec *NonObservation*.

la figure 3.3 page ci-contre présente les résultats d'expériences d'injection de fautes visant les services de désignation de plusieurs fournisseurs. Les fautes injectées sont des inversions de bits. Pour chaque cible, nous avons une expérience pour chaque position successive de bit dans le message corrompu émis par le client injecteur. La longueur de ce message varie en fonction de la cible (*c.f.* l'encadré ✠B page 76) ; cette figure regroupe pour chaque cible des résultats comprenant entre 640 et 1000 expériences.

Un premier résultat positif est que nous n'avons jamais observé d'arrêts inopinés ni de gels du noyau Linux au cours de nos expériences. Aucune des implémentations que nous avons testées n'utilise une stratégie résidante noyau, et il semble que les mécanismes de confinement d'erreur entre le noyau et les niveaux supérieurs fonctionnent bien.

Le très faible taux de propagation d'erreur vers d'autres clients du service (manifestation de *RemotePropagation*) est un autre résultat positif. Les cas de propagation multiple que nous avons observés sont des scénarios avec arrêt inopiné du SUT suite à l'injection de la faute, où un client reçoit une exception `COMM_FAILURE` mais l'autre client reçoit une exception `NotFound`. Il est probable que la faute injectée ait entraîné une corruption de l'état interne du SUT, qui a conduit à ce qu'une valeur erronée soit retournée à l'un des clients. La propagation de cette erreur a alors conduit à l'arrêt inopiné du SUT. Les expériences avec *RemotePropagation* sont les seules à ne pas être déterministes : la même faute injectée provoque parfois une manifestation de *ServiceCrash* sans propagation, et parfois une *RemotePropagation* avant l'arrêt inopiné du SUT.

Les manifestations de *ApplicationFailure* (propagation d'erreur vers l'injecteur, c'est-à-dire vers le client par lequel est arrivée la faute) peuvent être facilement expliquées par le fait que la faute injectée a corrompu le paramètre de la requête `resolve`, donc le service ne trouve pas de liaison pour ce nom. Le taux de ces manifestations est très proche entre les différentes implémentations, sauf pour le candidat *TAO*, où il est plus élevé.

Les expériences classifiées en *NonObservation* peuvent être dues à la non activation de la faute, ou au fait que la faute soit restée latente (l'erreur qu'elle a engendrée n'a pas eu le temps de se propager à l'interface du système). Le taux relativement élevé de manifestations de faute par *NonObservation* dans nos expériences s'explique par le fait que certaines parties des messages IIOP sont non-significatives, et peuvent donc être ignorées des courtiers lors du déballage d'une requête. En effet, le format CDR

Cible	RemoteProp	AppFailure	SCrash	SHang	ConHang	DétExc	NonObs
TAO	0	13.12	5.94	0	10.42	39.69	30.83
ORBacus	0	6.25	0.56	0	5.80	64.73	22.66
omniORB	0	8.75	0	0	4.53	66.41	20.31
MICO	0.31	8.75	0.16	1.41	9.06	58.91	21.41
ORBit	0.27	5.84	6.93	0	18.48	38.72	29.76
Sun JDK	0	5.00	0	0	25.40	37.90	31.70
IBM JDK	0	8.33	0.39	2.60	26.82	32.03	29.82

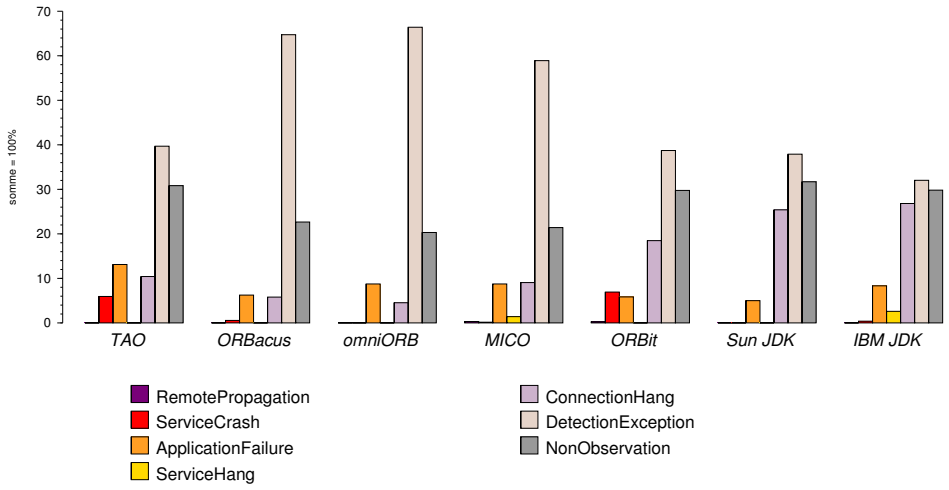

FIG. 3.3 – *Distribution des manifestations de faute par cible.*
Expériences d'inversion de bits pour le service de désignation.

impose des contraintes d'alignement sur les champs[3], ce qui impose le recours à des bits de « padding » lors de l'emballage de certains types de données. Par exemple, un paramètre de type booléen doit être mis à plat sur 8 bits ; 7 bits dans le message IIOP sont donc non-significatifs. De même, certains bits dans l'en-tête IIOP sont réservés à une utilisation future, ce qui implique que la corruption de leur valeur peut passer inaperçue.

Une partie des manifestations de faute par gel d'une connexion (les cas de *ConnectionHang*) peut être expliquée par le fait que la faute a touché la partie de l'en-tête GIOP qui code la longueur du message. Si nos expériences duraient suffisamment longtemps, ces cas seraient détectés par levée d'une exception `TRANSIENT` dans le client injecteur. Le temps de détection dépend principalement de la configuration du courtier client ; dans nos expériences, nous avons observé que la levée d'une telle exception s'effectuait au bout d'environ 6 minutes. Nous n'avons pas considéré ce cas dans notre classification, puisque le temps de détection de l'erreur est bien trop long par rapport au temps de traitement nominal. Nous considérons cette durée de détection comme étant incompatible avec la réactivité nécessaire à la mise en œuvre de mécanismes de détection d'erreur.

Les implémentations Java du service de désignation présentent un taux de manifestations de *ConnectionHang* très élevé, ce qui n'est pas très positif du point de vue de la sûreté de fonctionnement. Nous n'avons pas d'explication pour ce phénomène.

D'après ces résultats, les implémentations *TAO* et *ORBit* montrent des comportements peu robustes face à des corruptions réseau. En effet, près de 6% des expériences provoquent immédiatement l'arrêt inopiné du service. Ces résultats sont reproductibles, ce qui implique qu'un utilisateur malveillant de ce service peut facilement provoquer sa défaillance, simplement en lui envoyant une requête mal formée. Nous reviendrons dans le chapitre suivant sur cette observation, et sur ses conséquences du point de vue de la sécurité informatique.

À l'exception des cas de *RemotePropagation*, ces expériences sont répétables. Nous avons constaté peu de sensibilité par rapport au comportement de la charge applicative ; les résultats sont semblables même en l'absence d'activité pour stimuler le SUT. Nous n'avons pas non plus constaté de sensibilité vis-à-vis du mode de déclenchement de l'injection de la faute.

[3] Les contraintes d'alignement dans le format CDR avaient pour objectif de favoriser la mise en œuvre de primitives haute-performance d'emballage et de déballage des requêtes, puisqu'en se rapprochant des représentations des données dans la mémoire vive, ce format facilite le transfert direct de données entre la mémoire et le tampon emballé. L'avantage réel procuré par ces contraintes d'alignement est moins important que celui escompté.

3.2.3 Analyse des diagnostics d'erreur

Nous analysons maintenant plus en détail les résultats des expériences où l'erreur injectée a été détectée. Cette détection est rapportée par l'intergiciel côté client via la levée d'une exception CORBA. La qualité de ce diagnostic est déterminante pour la sûreté de fonctionnement du système, puisqu'elle détermine la pertinence des mécanismes de recouvrement d'erreur qui peuvent être mis en œuvre au niveau applicatif par le client.

TRANSIENT	une erreur de communication transitoire s'est produite
COMM_FAILURE	une erreur non-transitoire de communication s'est produite (généralement provoquée par l'arrêt inopiné du serveur)
BAD_OPERATION	l'opération désignée par la requête (le nom de la méthode) est invalide
OBJECT_NOT_EXIST	l'objet désigné par la requête n'existe pas
OBJ_ADAPTER	l'adaptateur d'objets a rencontré une erreur (typiquement ceci indique que la clé objet contenue dans la requête ne correspond à aucun adaptateur d'objets)
INV_OBJREF	la référence objet est mal formée (par exemple le Repository-ID est invalide)
MARSHAL	une erreur est survenue lors de l'emballage ou du déballage des données vers ou depuis le format CDR
NO_MEMORY	l'objet serveur ne dispose pas de suffisamment de mémoire pour accomplir la requête
INTERNAL	une erreur interne au courtier s'est produite
UNKNOWN	une erreur inconnue (non diagnostiquée par le courtier) s'est produite du côté serveur

FIG. 3.4 – *Exceptions* CORBA *levées au niveau intergiciel*

La figure 3.4 présente les différentes exceptions CORBA qui peuvent être levées par l'intergiciel. Il s'agit d'exceptions dites de type système, dont la sémantique est définie par la norme CORBA. La figure 3.5 page suivante donne la répartition par cible des exceptions CORBA observées, pour les expériences avec détection.

Une première information apportée par cette figure est la proportion d'exceptions de type INTERNAL et UNKNOWN levées par les différents candidats. Ces exceptions sont « mauvaises » dans le sens où elles ne fournissent pas d'information utile pour le diagnostic de l'erreur, et ne guident pas le niveau applicatif dans un éventuel choix d'action de recouvrement. Le candidat *Sun JDK* est peu robuste d'après ce critère.

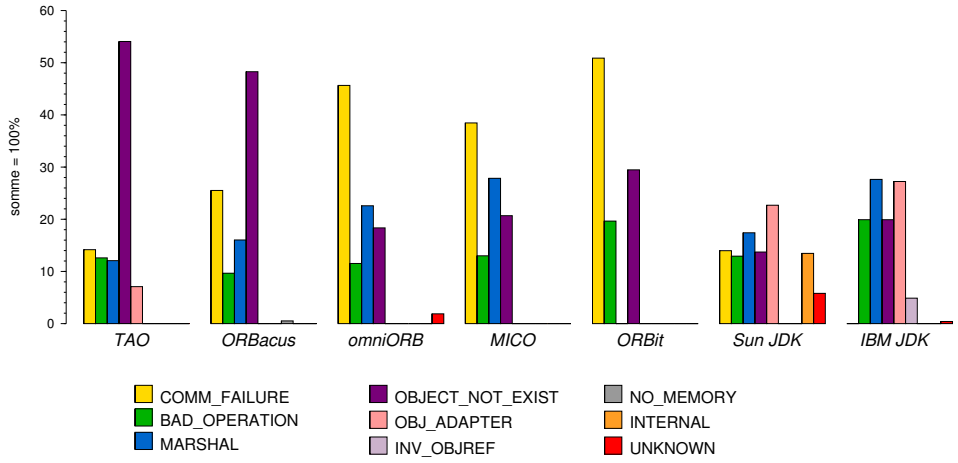

FIG. 3.5 – *Distribution des exceptions par cible, expériences avec détection. Expériences d'inversion de bits pour le service de désignation.*

La proportion d'exceptions `BAD_OPERATION` varie peu entre les candidats. Ceci s'explique si l'on considère que la partie du message IIOP qui identifie l'opération à invoquer doit forcément être décodée par le courtier ; il n'est donc pas surprenant d'observer que le pouvoir de détection d'erreurs pour cette zone est proche pour des implémentations différentes.

Les exceptions `OBJECT_NOT_EXIST`, `OBJ_ADAPTER` et `INV_OBJREF` signalent la même situation de corruption de la clé objet. La proportion de ces exceptions n'est pas équivalente entre implémentations, même en cumulant ces trois exceptions pour chaque implémentation. La différence s'explique par le fait que *omniORB* et *MICO* génèrent des clés d'objet bien plus courtes que les autres courtiers, et donc que la proportion de la requête qui représente la référence objet est plus faible pour eux que pour les autres cibles.

Influence de la position de la faute

La figure 3.6 page ci-contre présente les diagnostics d'erreur observés en fonction de la position dans le message IIOP où la faute a été injectée. Cette figure correspond aux expériences d'inversion de bits visant le service de désignation, toutes cibles confondues. On constate que les corruptions qui affectent les premiers 32 bits du message sont détectées par le SUT et toujours signalées (quelle que soit l'implémentation visée) par une exception de type `COMM_FAILURE`. Cette observation s'explique par le fait que les requêtes CORBA commencent par quatre octets « magiques » (les caractères 'G', 'I', 'O', 'P'encodés en ASCII), dont la corruption peut facilement être détectée par l'intergiciel.

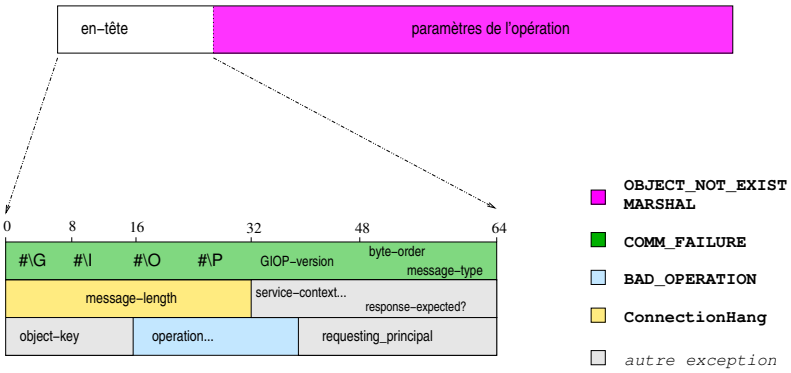

FIG. 3.6 – *Diagnostics d'erreur en fonction de la position de la faute*

Les exceptions de type `BAD_OPERATION` (qui indiquent que le nom de l'opération invoquée est invalide) sont levées principalement lorsque la corruption touche la partie de l'en-tête qui identifie l'opération à invoquer. De la même manière, les exceptions de type `OBJECT_NOT_EXIST` et `INV_OBJREF` sont levées essentiellement lorsque la corruption touche la portion du message qui identifie l'objet à invoquer.

Des exceptions de type `NO_MEMORY` sont levées par le candidat *ORBacus* lorsque l'inversion de bits touche à la longueur annoncée du message. L'inversion de bits rend la longueur annoncée du message très grande[4]. Le courtier cherche à allouer un tampon mémoire de grande taille pour contenir le message, et le système d'exploitation informe le courtier qu'il ne peut pas allouer autant de mémoire ; le service renvoie cette information au client par une exception `NO_MEMORY`.

Les exceptions de type `MARSHAL` sont surtout levées lorsque les paramètres du message (ici un nom, qui est composé d'une séquence de chaînes de caractères) sont corrompus.

Qualité du diagnostic pour recouvrement d'erreur

Analysons maintenant ces résultats du point de vue des mécanismes de recouvrement d'erreur qui peuvent être mis en œuvre par les clients. Les diagnostics d'erreur fournis par les différentes implémentations varient en fonction de leur pertinence pour décider d'une éventuelle action de recouvrement.

Une exception CORBA comporte plusieurs champs, qui peuvent véhiculer des informations annexes sur l'erreur qui a provoqué la levée de l'exception. Un premier champ annexe de toute exception CORBA est le *minor code*, un code entier qui permet au

[4]En effet, la longueur en octets du message est un entier non-signé codé sur 32 bits, et les messages affectés par la corruption sont de taille inférieure à 125 octets, donc inférieure à 2^7. La majorité des bits codant cette longueur étant à zéro, leur inversion a une forte probabilité d'augmenter la taille annoncée du message.

courtier de fournir une information complémentaire sur la nature de l'erreur. Par exemple, une exception `COMM_FAILURE` associée à un code « échec de la tentative de connexion réseau » indique un problème sur le nœud distant, alors qu'associée au code « impossible d'accéder au réseau », elle indique plutôt un problème sur le nœud local. Les valeurs autorisées pour le code mineur sont spécifiques à chaque fournisseur. Certains courtiers ne renseignent pas ce champ de l'exception, et fournissent ainsi moins d'information pour le recouvrement au niveau applicatif.

Une seconde information complémentaire associée à une exception est son champ *completion status*, qui permet à l'adaptateur d'objets de signaler si le traitement de l'opération invoquée a été terminé ou non. L'état `COMPLETED_NO` indique que la demande d'exécution n'a pas été transmise au servant, et donc que le client peut répéter l'invocation, ou basculer vers un autre serveur. L'état `COMPLETED_YES` indique que le traitement de l'opération a été terminé. L'état le moins utile du point de vue du recouvrement est `COMPLETED_MAYBE`, qui indique que le courtier n'a pas pu dire si l'opération a été effectuée ou non.

La principale cause des exceptions avec `COMPLETED_MAYBE` est liée aux exceptions `COMM_FAILURE` levées par le courtier côté client. Lors de la rupture de la connexion réseau avec le nœud distant (suite par exemple à l'arrêt inopiné du processus léger qui gérait cette connexion), le courtier côté client n'est pas en mesure de savoir quel point du traitement de la requête a été atteint avant que la connexion ne soit interrompue. Dans cette situation, il lève une exception `COMM_FAILURE` avec le champ *completion status* à `COMPLETED_MAYBE`.

La figure 3.7 page suivante illustre la distribution des valeurs du champ *completion status* en fonction du fournisseur. Pour chaque cible, toutes les `SystemException` observées pour la campagne d'inversion de bits sur le service de désignations sont analysées. On voit sur cette figure que le candidat *TAO* est le seul à répondre par exception alors que l'opération a été terminée. Les candidats *ORBit*, *IBM* et *omniORB* présentent un plus fort taux de `COMPLETED_MAYBE`, ce qui est négatif.

3.2.4 Latence de détection d'erreur

Lorsque l'intergiciel détecte l'erreur injectée dans le message (expériences classées dans la catégorie *DetectionException*), nous pouvons calculer le temps moyen de détection d'erreur : il s'agit du nombre de secondes écoulées entre l'instant de démarrage de l'injecteur et la notification de l'exception CORBA dans l'injecteur. Cette information est importante, puisqu'il s'agit de l'un des aspects de le transparence de la distribution (ou de l'illusion de localité mise en place par le courtier) qui est le plus fragile.

La figure 3.8 page 86 présente, pour chaque classe de diagnostic, le temps moyen de détection d'erreur et l'écart-type, toutes cibles confondues. Les figures 3.9 et 3.10 présentent les mêmes informations, mais pour les seules expériences visant les candidats *IBM* et *omniORB* respectivement. On constate que la latence de détection d'erreur varie considérablement d'un candidat à l'autre. Le comportement du candidat *omniORB* est rassurant, puisque la latence de détection des erreurs est uniformément faible. Le

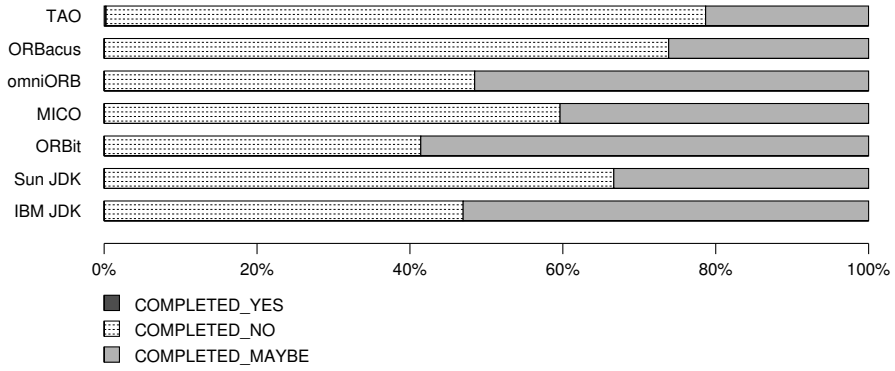

FIG. 3.7 – *Completion status par cible*

candidat *IBM* présente un comportement peu satisfaisant, puisque sa latence moyenne de détection d'erreur est particulièrement élevée par rapport aux autres candidats, avec des écarts types importants.

3.2.5 Modèle de faute « double zéro »

Dans [Marsden & Fabre 2001], nous avons rapporté les résultats d'expériences d'injection de fautes de type « double zéro », à la place d'inversions de bits. Dans ces expériences, l'insertion de l'erreur consiste à mettre deux octets successifs du message IIOP à zéro. Nous avons constaté que ce modèle de fautes provoque des manifestations de faute moins graves que les inversions de bits, et que la diversité des diagnostics d'erreur provoqués est moins importante.

3.2.6 Impact de fautes multiples

Tous les résultats que nous avons présentés jusqu'à maintenant concernent des fautes isolées, c'est-à-dire des expériences où une seule faute est injectée. Dans la pratique, un système peut être soumis à de multiples fautes, surtout lorsqu'il s'agit de fautes intentionnelles d'origine malveillante. Nous avons donc mené des expériences avec un grand nombre d'exécutions de l'injecteur de fautes.

Pour la plupart des implémentations visées, les expériences avec fautes multiples provoquent les mêmes effets que les expériences avec une seule faute. La seule exception que nous ayons constatée concerne le service de désignation de *MICO*, qui présente un mode de défaillance nouveau par *RemotePropagation* après 200 fautes injectées. Contrairement aux défaillances par *RemotePropagation* que nous avons observées pour cette cible lors de l'injection d'une seule faute, il ne s'agit pas d'un cas de *ServiceCrash*.

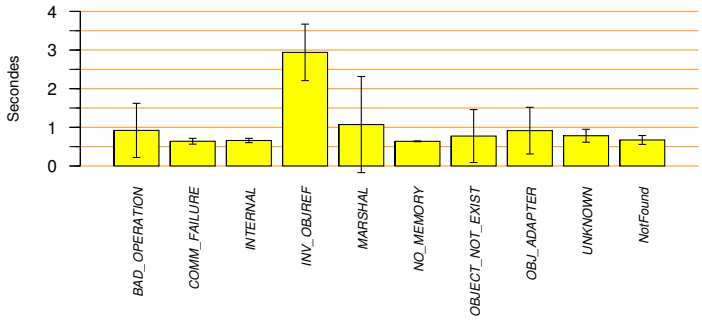

FIG. 3.8 – *Latence moyenne de détection d'erreur, toutes cibles confondues*
En secondes, avec écart-type

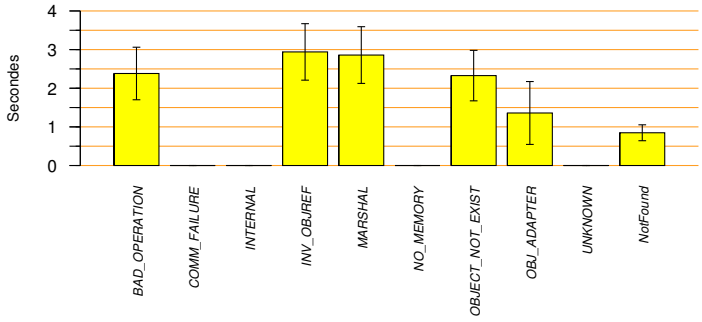

FIG. 3.9 – *Latence moyenne de détection d'erreur, IBM JDK*
En secondes, avec écart-type

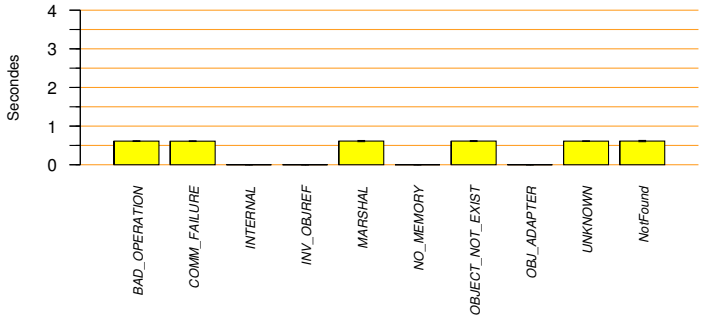

FIG. 3.10 – *Latence moyenne de détection d'erreur, omniORB*

En effet, le SUT atteint un état où il répond encore aux requêtes des clients, mais seulement par la levée d'une exception `OBJECT_NOT_EXIST`. Ce comportement pourrait s'expliquer par le fait que la table de l'adaptateur d'objets contenant la liste des servants hébergés par ce courtier a été corrompue suite au traitement d'une requête mal-formée.

3.3 Impact de corruptions IIOP sur le service de diffusion d'événements

Le service de diffusion d'événements CORBA est basé sur la notion de *canal à événements*, qui sert d'intermédiaire entre les producteurs et les consommateurs d'événements. Les producteurs et les consommateurs sont découplés dans l'espace, puisqu'un producteur ne sait pas quel est le nombre de consommateurs qui reçoivent ses événements, ni leur identité ; et les consommateurs ne peuvent pas savoir quel producteur est à l'origine d'un événement donné.

Le rôle du canal à événements est d'accepter des demandes d'inscription provenant de consommateurs ou de producteurs, puis de diffuser tous les événements des producteurs vers les consommateurs. Les consommateurs peuvent choisir d'utiliser un fonctionnement en mode *pull* (où ils interrogent régulièrement le canal afin de savoir si des événements sont disponibles), ou un modèle *push* (où le canal leur envoie un message à chaque fois qu'un événement leur est destiné). Symétriquement, les producteurs d'événements peuvent choisir d'envoyer les événements au canal ou demander à ce que le canal les interroge régulièrement pour récupérer les événements en attente.

3.3.1 Configuration expérimentale

La charge applicative que nous avons développée est constituée d'un producteur d'événements et de deux consommateurs. Les événements créés par le producteur contiennent une estampille indiquant l'heure où ils ont été créés, ainsi qu'un numéro de séquence. Lorsqu'il reçoit un événement, un consommateur compare l'estampille du message avec l'heure courante, afin de déterminer le temps de propagation de l'événement. Les consommateurs comparent également le numéro de séquence du message à un compteur qu'ils maintiennent localement, afin de déterminer si des événements se perdent.

Dans notre banc de test, le service de diffusion d'événements, le producteur et les consommateurs s'exécutent sur des machines différentes. Nous utilisons un modèle

de communication *push*-producteur, *pull*-consommateur, avec des événements non-typés[5]. La charge de travail est capable de détecter des événements invalides qui lui sont délivrés par le service en examinant leur numéro de séquence. L'injection de fautes est effectuée par un producteur d'événements, qui invoque la méthode `push` sur le canal d'événements. La suite du déroulement d'une expérience est semblable au déroulement d'une expérience ciblant le service de désignation.

3.3.2 Résultats

La figure 3.12 page suivante présente les résultats d'une campagne d'injection d'inversions de bits, visant les services de diffusion d'événements d'*ORBacus*, *TAO* et *OpenORB*. Ces expériences sont une extension de celles rapportées dans [Marsden et al. 2002b]. Comme pour les résultats que nous avons présentés pour le service de désignation (*c.f.* § 3.2.2 page 77), la figure regroupe les résultats d'expériences où chaque position de bit possible dans la requête `push` a été corrompue, soit environ 1500 expériences par cible.

Ces expériences montrent que le service de diffusion d'événements d'*ORBacus* est plus robuste que celui de *TAO*, puisque son taux de manifestation de *ServiceCrash* est plus faible, et que son taux de détection est plus élevé. Le service de diffusion d'événements de *OpenORB* est peu robuste, puisqu'il présente un fort taux d'arrêts inopinés.

Les manifestations de *ApplicationFailure* dans cette campagne sont des cas d'arrêt inopiné du consommateur d'événements : la corruption du message d'un producteur

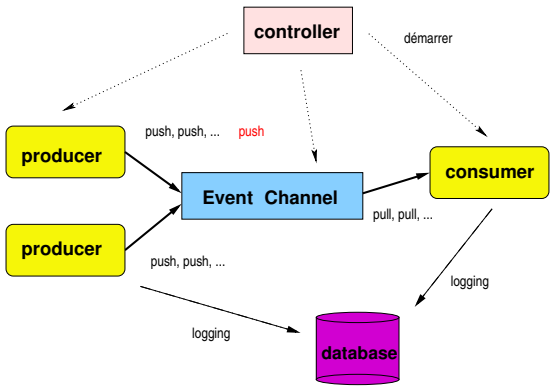

FIG. 3.11 – *Configuration des expériences ciblant le service de diffusion d'événements*

[5]Le service de diffusion d'événements CORBA permet le choix entre des événements non-typés et des événements typés. Dans le premier cas, les événements sont emballés dans un conteneur `Any`, ce qui permet de transporter n'importe quel type de données IDL, mais qui nécessite un accord entre les producteurs et les consommateurs pour emballer les données sous un format particulier. Dans le cas d'événements typés, le programmeur fournit une interface IDL décrivant les opérations qui pourront être invoquées par les consommateurs et les producteurs d'événements.

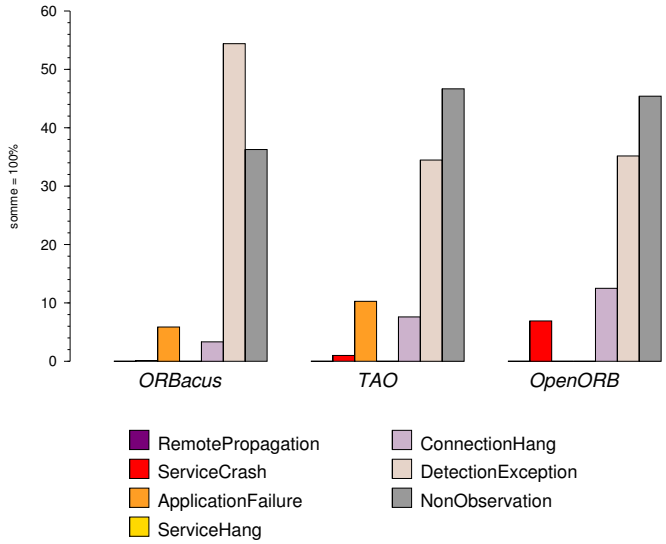

FIG. 3.12 – *Distribution des manifestations de faute par cible.*
Expériences d'inversion de bits pour le service de diffusion d'événements.

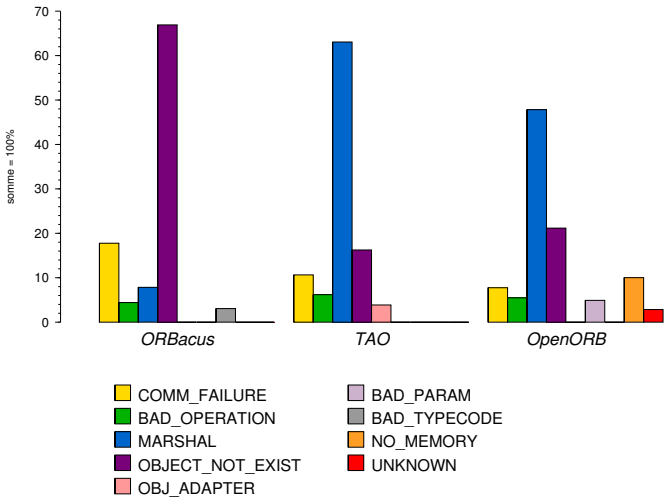

FIG. 3.13 – *Distribution des exceptions par cible, expériences avec détection.*
Expériences d'inversion de bits pour le service de diffusion d'événements.

d'événements s'est propagée via le service vers un consommateur. Le taux non négligeable de *ApplicationFailure* pour les candidats *ORBacus* et *TAO* indique que ces services analysent peu le contenu des événements qu'ils diffusent, permettant à des événements corrompus d'atteindre le consommateur d'événements. Par contre, aucune propagation d'erreur n'a été constatée pour le service d'*OpenORB*, qui semble s'arrêter avant de transmettre une donnée corrompue.

Il est intéressant de constater que la variété des exceptions levées est plus restreinte pour ce service que pour les expériences concernant le service de désignation. Ceci peut s'expliquer en partie par le fait que l'opération `push`, qui subit une corruption dans ces expériences, est une opération `oneway`. Le courtier peut traiter ces invocations sans retour différemment des opérations ordinaires. Nous observons la présence d'une nouvelle exception `BAD_TYPECODE` levée par le service de diffusion d'événements de *ORBacus*, pour signaler que le typecode de l'événement qu'on lui demande de diffuser n'est pas valide ; le candidat *OpenORB* semble signaler cette corruption par la levée d'une exception `BAD_PARAM`.

3.4 Impact de fautes GIOP sur des instances de service synthétique

Ces expériences ont pour objectif d'évaluer l'impact de messages inter-courtier ayant subi une corruption qui a affecté les données dans la couche d'application (au niveau GIOP). La motivation pour ces expériences a été présentée au § 2.2.5. Rappelons que ce modèle de fautes simule principalement les fautes du logiciel.

3.4.1 Configuration Expérimentale

Nous avons travaillé avec les mêmes cibles CORBA que dans la section précédente, et avec les mêmes charges applicatives. Le composant d'injection de faute prend en paramètre la référence objet du SUT, et lui envoie un message avec certains champs contenant des valeurs peu courantes ou mal formées.

Nous avons injecté les fautes suivantes :

- un message GIOP de type `LocateReply` est envoyé au SUT, alors qu'il n'a pas émis le message `LocateRequest` correspondant ;
- un message GIOP de type `CancelRequest` est envoyé au SUT, pour un identifiant de requête qui ne correspond à aucune requête qu'il a traitée ;
- un message GIOP de type `MessageError` est envoyé au SUT, pour un identifiant de requête qu'il ne connaît pas ;
- l'émission d'un message de type `MessageError` comportant des informations quelconques à la fin du message ;

- un message GIOP version 1.0 de type `MessageError` avec le bit fragmenté activé, alors que cette version du protocole n'autorise pas la fragmentation (*c.f.* l'encadré ✠ *C* ci-dessous) ;
- un message GIOP de type `MessageFragment`, alors que la version de protocole employée (1.0) n'autorise pas l'utilisation de messages fragmentés ;
- un message GIOP avec un numéro de version GIOP élevé ;
- le champ de l'entête GIOP qui identifie l'encodage petit-boutien ou grand-boutien (le « byte-order » de la requête) positionné à un nombre différent de 0 et 1 ;
- un message avec un nom d'opération contenant des caractères inattendus (caractères nuls, de contrôle, caractères non-ASCII etc).

> ✠ *C* — **La fragmentation GIOP**
>
> La *fragmentation* est un mécanisme qui est apparu dans la version 1.2 du protocole GIOP. Il permet à un courtier client de décomposer un message IIOP de grande taille en plusieurs *fragments*, chacun étant envoyé dans un message GIOP séparé. Dans le premier fragment, un bit réservé de l'entête GIOP est utilisé pour indiquer que d'autres fragments du message vont suivre. Les fragments suivants ont un type GIOP nouveau, le `MessageFragment`.

3.4.2 Résultats

Dans presque tous les cas, le SUT ferme la connexion réseau qui transporte la requête. Ceci provoque la levée d'une exception `COMM_FAILURE` chez le client qui a émis cette requête corrompue. Certains courtiers répondent avec un message GIOP de type `MessageError`, indiquant l'identifiant du message invalide, avant de fermer la connexion réseau.

Les seuls défauts de robustesse que nous avons observés dans nos expériences sont :

- l'arrêt inopiné du service de désignation de *TAO* lorsqu'il reçoit plusieurs messages fragmentés alors que la version GIOP du message n'autorise pas la fragmentation ;
- l'arrêt inopiné du service de désignation de *Fnorb* lorsqu'il reçoit un message `LocateForward` contenant une IOR invalide.

Ces résultats sont reproductibles. Les résultats sont indépendants de la nature du niveau applicatif ; nous observons le même comportement pour le service de désignation que pour des instances synthétiques du service d'écho.

3.5 Évaluation de la propagation d'erreur depuis l'exécutif

Dans ces expériences, nous cherchons à évaluer l'impact de fautes qui atteignent l'intergiciel par le biais de son interaction avec le système d'exploitation (*c.f.* § 2.2.4). L'objectif de ces expériences est triple :

i) évaluer la robustesse du candidat intergiciel vis-à-vis de comportements corrects, mais rares, du système d'exploitation ;

ii) caractériser le comportement de l'intergiciel en présence d'erreurs rapportées par le système d'exploitation ;

iii) caractériser les modes de défaillance de l'intergiciel lorsque le système d'exploitation défaille.

Dans le cas *(i)*, nous souhaitons que toutes les expériences soient classifiées en *Non-Observation* : le service, tel qu'il est perçu par les clients du SUT, ne devrait pas être perturbé (sauf éventuellement dans le domaine temporel).

Dans le cas *(ii)* de signalement d'une erreur, le comportement attendu du SUT dépend de la nature de l'erreur que nous simulons. Dans la plupart des cas, il est acceptable de voir un seul client affecté par l'erreur, mais le SUT devrait continuer de servir d'autres clients. Lorsque l'erreur signalée par le système d'exploitation se rapporte à un phénomène qui pourrait être transitoire (l'échec d'une demande d'allocation mémoire par exemple), le SUT peut attendre quelques secondes avant de réessayer la demande. Dans certains cas, l'erreur peut être rapportée au client via une exception CORBA, telle que `NO_MEMORY` ou `NO_RESOURCES`.

Dans le cas *(iii)* nous cherchons surtout à savoir comment peut réagir l'intergiciel face à des comportements erronés du système d'exploitation.

Il est difficile avec cette technique d'injection de faute d'effectuer des comparaisons directes entre candidats. En effet, les candidats n'interagissent pas tous de la même manière avec le système d'exploitation sous-jacent : ils n'utilisent pas forcément les mêmes appels système. Notre analyse se limite donc à l'identification de défauts de robustesse ou de fautes du logiciel dans chaque candidat, sans que nous cherchions à effectuer une analyse comparative.

3.5.1 Configuration Expérimentale

Une campagne mettant en œuvre cette technique d'injection de faute consiste à observer les effets sur le SUT de ces comportements inattendus du système d'exploitation. Nous cherchons à savoir si la faute est détectée par l'intergiciel. Lors d'une détection, quel est le comportement de l'intergiciel : levée d'exception CORBA, tentative de recouvrement d'erreur, arrêt inopiné ou gel ?

Notre banc de test expérimental insère une couche d'interception entre l'intergiciel et le système d'exploitation, qui permet d'intercepter les demandes de service provenant de l'intergiciel. En exécution nominale, cette couche d'interception propage la demande de service au système d'exploitation, puis transmet le résultat à l'intergiciel. Lorsque l'injection de la faute est déclenchée, la couche d'interception ne propage pas l'appel au système d'exploitation, et retourne – éventuellement après un délai prédéterminé – un code erreur à l'intergiciel.

La mise en œuvre du mécanisme d'interception des appels système est décrite dans l'annexe B (page 129).

3.5.2 Sensibilité des courtiers CORBA au comportement de l'exécutif

Dans ces campagnes, nous avons caractérisé le comportement des services de désignation de différents fournisseurs, avec une configuration expérimentale proche de celle décrite au § 3.2.

La première famille de fautes injectées consiste en l'émulation de comportements corrects, mais inattendus, du système d'exploitation. Nous nous sommes focalisés sur les services orientés communication du système d'exploitation, en simulant des lectures et des écritures incomplètes et l'interruption d'une opération d'entrée/sortie (*c.f.* la page 59).

La majorité des candidats intergiciel que nous avons testés ont un comportement robuste face à ces situations. Nous avons observé les défauts de robustesse suivants :

- les lectures et écritures incomplètes ont provoqué une manifestation de *Remote-Propagation* avec le candidat *ORBit*. Tous les clients connectés à l'instant où la faute est injectée reçoivent une exception `NotFound`, mais le service continue de fonctionner correctement pour de nouveaux clients.
- les opérations d'entrée/sortie interrompues (erreur `EINTR` pour les opérations `read`, `write`, `send`, `recv`, `sendto`, `recvfrom`) ont provoqué une manifestation de *RemotePropagation* pour les candidats *TAO* et *ORBit*.

La seconde famille de fautes injectées consiste à émuler des situations d'erreur (de ressources ou des erreurs de l'environnement) qui sont rapportées au SUT par des retours de codes d'erreur sur des appels système. Nous n'avons pas constaté de défauts de robustesse lors de ces expériences ; tous les candidats exhibent un comportement acceptable lorsqu'ils sont confrontés aux situations que nous avons émulées. Par exemple, lorsque le système d'exploitation renvoie un code erreur tel que `EPIPE` sur un appel système `write`, la connexion réseau correspondante est fermée, mais l'erreur ne se propage pas aux autres clients connectés à cet instant.

La dernière famille de fautes que nous avons injectées consiste à évaluer l'impact sur le SUT de fautes transitoires du système d'exploitation. Le comportement le plus

intéressant que nous ayons observé concerne l'appel système `select`. En sortie de cet appel système, notre saboteur modifie le paramètre `timeout` (employé pour préciser une durée maximale d'attente dans le noyau), de façon à ce qu'il contienne des valeurs erronées. Ce saboteur a provoqué une manifestation de *ServiceCrash* chez les candidats *TAO* et *MICO*, et aucun signalement pour les autres candidats.

À l'exception des candidats *ORBit* et *TAO*, les implémentations que nous avons testées sont robustes dans leur manière d'interagir avec le système d'exploitation. Dans la suite, on verra que l'on ne peut pas dire la même chose pour bien d'autres logiciels.

3.5.3 Résultats pour d'autres logiciels de communication

Nous avons effectué le même type de campagne d'injection de faute pour d'autres types de logiciels de communication, afin d'évaluer le degré de robustesse de leur interaction avec le système d'exploitation. Nous avons observé les résultats suivants :

- la bibliothèque *OpenSSL*, qui implémente des mécanismes de chiffrement et d'authentification des communications TCP, est sensible à des lectures incomplètes[6]. Une lecture ou écriture partielle entraîne la coupure de la connexion sécurisée. Nous avons constaté ce défaut en testant plusieurs clients HTTPS.
- un client HTTP distribué avec l'environnement Perl (logiciel GET distribué avec libwww-perl) est sensible à des lectures incomplètes. Nous avons observé une **corruption silencieuse des données** téléchargées depuis le serveur web, puisque le client oublie les données affectées par une transmission incomplète.
- le serveur web *Apache* présente un comportement robuste face à toutes les situations auxquelles nous l'avons soumis.

Il ne s'agit pas d'intergiciels de communication, puisque ces logiciels n'ont pas pour objectif de rendre transparente la répartition. Toutefois, il s'agit de logiciels utilisés intensivement pour la communication entre systèmes, et qui pourraient servir de substrat à des environnements de « services web ».

3.5.4 Résultats pour des logiciels grand public

Par curiosité, nous avons également appliqué cette technique de caractérisation à des logiciels visant un public plus large, avec des résultats parfois surprenants. Voici quelques exemples des comportements non-robustes que nous avons observés :

- le compilateur C de la suite GCC, version 3.3 ne gère pas correctement l'interruption d'opérations d'entrée/sortie : ces comportements de l'exécutif provoquent un arrêt inopiné compilateur sur violation de segmentation.

[6] Ce défaut a été corrigé dans la version 0.9.7 de OpenSSL, qui a été annoncée au cours de la rédaction de cette thèse.

- certaines machines virtuelles Java sont sensibles à des lectures incomplètes lorsqu'elles lisent les fichiers « timezone », qui contiennent des informations sur les horaires. Une autre JVM rencontre une violation d'assertion interne en lisant des informations sur les polices de caractère disponibles, et refuse de démarrer.
- la machine virtuelle Java gij, version 3.2, ne gère pas bien les lectures incomplètes, levant une exception de type `ClassNotFound` au démarrage.
- la version 20.2 de l'éditeur Emacs ne gère pas correctement les lectures partielles lorsqu'il lit les fichiers Emacs Lisp compilés.
- le logiciel graphique *GIMP* est sensible à des lectures incomplètes lorsqu'il lit des fichiers graphiques du type `.gbr`.
- les bibliothèques graphiques *Xt* et *Motif* sont sensibles à des lectures incomplètes lorsqu'elles lisent leurs fichiers de configuration.

Au vu des résultats médiocres de ces logiciels, qui sont largement déployés, les bonnes réactions de la majorité des courtiers CORBA que nous avons testés sont d'autant plus louables.

3.6 Expériences de mutation de code

Nous avons effectué un nombre limité d'expériences en appliquant des opérateurs de mutation à des interfaces OMG IDL, ceci afin de caractériser la robustesse des compilateurs IDL et des référentiels d'interface de différents fournisseurs. La motivation pour ces expériences a été présentée dans la Section 2.2.2. Les interfaces IDL sont générées de manière automatique à partir d'une grammaire de génération stochastique dérivée de la spécification OMG IDL ; nous décrivons ce processus de génération stochastique dans l'Annexe A page 125. Ces entrées sont « stressantes » pour le SUT, d'une part puisque nous injectons des fautes dans les interfaces, et d'autre part puisque les interfaces générées par le processus stochastiques – bien qu'elles soient légales d'après la spécification du langage OMG IDL – ne ressemblent pas toutes à des interfaces « réelles ».

Nous avons utilisé ces interfaces synthétiques pour tester :

- les compilateurs IDL des différents fournisseurs, afin de vérifier sa capacité à générer des souches et squelettes correspondants ;
- les référentiels d'interfaces, afin de vérifier sa capacité à stocker et à restaurer la description du service.

3.6.1 Compilateurs IDL et interfaces mutées

Le principe de ces expériences est de générer de manière automatique une interface IDL synthétique, d'y insérer une faute à l'aide d'un opérateur de mutation, et de soumettre l'interface mutée au compilateur cible. Nous utilisons une classification naïve des modes de défaillance, qui ne considère que les arrêts inopinés et les gels du compilateur. En particulier, nous n'avons pas cherché à évaluer la correction du code généré par les compilateurs (cette question a été abordée sous l'angle du test d'un système réflexif basé sur CORBA dans la thèse de Juan-Carlos Ruiz-García [Ruiz-García 2002]).

Les mutations effectuées sur les interfaces synthétiques sont les suivantes :

- l'utilisation de noms d'identifieurs invalides, contenant des caractères qui ne sont pas autorisés par la spécification OMG IDL ;
- l'utilisation de noms d'identifieurs extrêmement longs ;
- l'utilisation de mots-clés réservés à la place de noms d'identifieurs ;
- la présence d'expressions arithmétiques invalides, par exemple des expressions mal typées (`'A' + 33`) ou des divisions par zéro ;
- la présence d'opérateurs de portée non-déclarés.

Notre classification des modes de défaillance est plutôt naïve, et ne comprend que deux catégories :

- **Crash** : l'arrêt inopiné du compilateur. Il s'agit généralement de la violation de segmentation ou d'une assertion interne non valide pour un compilateur implémenté en C++, ou d'un `NullPointerException` pour un compilateur implémenté en Java.
- **Hang** : gel du compilateur (compilation qui dure plus de 20 secondes).

La figure 3.14 page ci-contre donne les résultats d'expériences de mutation d'interfaces IDL. Elle regroupe les résultats de 1000 expériences, indiquant pour chaque candidat le pourcentage des expériences qui a provoqué une manifestation de faute par arrêt ou par gel. On constate que les candidats *Visibroker*, *TAO* et *omniORB* sont peu robustes vis-à-vis d'interfaces IDL mal-formées. Rappelons toutefois que les problèmes soulevés par ce type d'expérience devraient être détectés pendant la phase de développement d'un système, et donc qu'ils peuvent être considérés comme étant moins graves que les défauts de robustesse identifiés dans les sections précédentes.

Cible	**Crash**	**Hang**
TAO	18.6	3.4
ORBacus	0	0
omniORB	9.1	11
MICO	0.1	0
ORBit	3.5	0.3
Sun JDK	2.7	10.9
Visibroker	22.4	0
Fnorb	0	1.6

FIG. 3.14 – *Manifestations de faute pour les compilateurs IDL. Valeurs en pourcentages, sur 1000 expériences.*

3.7 Récapitulatif

Dans ce chapitre, nous avons présenté les résultats de nos expériences d'injection de faute appliquées à des intergiciels de type CORBA. Nous avons commencé par livrer les résultats d'une campagne visant à évaluer l'impact de corruptions au niveau IIOP sur des implémentations du service de désignation et de diffusion d'événements CORBA. Ces expériences provoquent un taux relativement élevé de propagation de l'erreur injectée vers le client dont l'interaction avec la cible a été perturbée. Certains candidats présentent un comportement peu robuste face à ces requêtes corrompues, avec un fort taux d'arrêts inopinés. Nous avons observé une diversité considérable dans la distribution et la pertinence des diagnostics d'erreurs fournis par les différents candidats, ainsi que dans leur latence de détection d'erreur.

Le deuxième type d'expérience que nous avons menée concerne l'impact d'erreurs de protocole GIOP. Ce type de faute provoque moins de défauts de robustesse chez les courtiers que les inversions de bit au niveau IIOP.

La troisième catégorie de fautes que nous avons injectées consiste à perturber l'interaction de l'intergiciel avec le système d'exploitation. Nous avons simulé différents types de comportements peu fréquents du système d'exploitation, en cherchant à mettre en défaut les mécanismes de recouvrement d'erreur des intergiciels candidats. Deux des implémentations que nous avons ciblées présentent des défauts de robustesse face à certains comportements corrects mais peu fréquents du système d'exploitation.

Nous avons ensuite cherché à caractériser la robustesse des outils de développement des différents fournisseurs, en utilisant une approche par mutation d'interfaces IDL. Ces expériences ont provoqué un grand nombre de manifestations d'arrêt inopiné ou de gel des compilateurs IDL testés.

Les types d'expériences que nous n'avons pas menées sont l'émulation de fautes de la mémoire et les tests de robustesse à l'API. Les techniques d'émulation de fautes mémoires sont bien connues dans la communauté de caractérisation de la sûreté de

fonctionnement, et nous avons préféré concentrer nos efforts sur des fautes qui sont plus spécifiques à des intergiciels de communication et aux interactions entre objets répartis. Quant aux tests de robustesse, une étude dans ce sens a été mené sur des courtiers CORBA dans le cadre du projet Ballista [Pan et al. 2001].

Dans le chapitre suivant, nous analysons ces résultats plus en détail, en comparant les différentes techniques d'injection de faute, et proposons une classification globale de la robustesse des courtiers que nous avons évalués.

Chapitre 4

Analyse comparative

DANS ce chapitre, nous analysons les résultats obtenus au chapitre précédent. Nous comparons tout d'abord la capacité des techniques d'injection de fautes utilisées dans ce mémoire à révéler des défauts de robustesse chez les candidats. Nous étudions au § 4.2 la variation des mesures de robustesse en fonction de la version du composant cible, et constatons que le choix de cette version induit des disparités importantes dans les résultats. L'examen d'informations collectées lors de notre banc de test et qui portent sur les scénarios provoquant des défaillances met en lumière un certain nombre de fautes du logiciel qui expliquent les modes de défaillance observés. Nous commentons au § 4.4 l'impact de l'architecture du courtier et du support exécutif sur lequel il s'appuie sur la robustesse des systèmes répartis à base d'intergiciel.

Dans la section 4.5, nous proposons une méthode pour intégrer nos expériences dans une perspective d'étalonnage de la sûreté de fonctionnement d'un composant intergiciel. Cette méthode se base sur le développement d'une métrique qui rassemble les différents défauts de robustesse observés, évalue leur impact sur un système à base d'intergiciel, et permet d'obtenir une mesure quantitative de la robustesse de différents candidats. Nous proposons quelques réflexions sur l'application de notre méthode de caractérisation à d'autres types d'intergiciels de communication, tels que DCOM et *.NET Remoting*.

Dans la section 4.6, nous analysons les vulnérabilités que nous avons identifiées et leurs conséquences sur la sécurité informatique. Enfin, nous proposons un ensemble de techniques d'encapsulation qui peuvent permettre à un intégrateur de systèmes semi-critiques d'augmenter la robustesse des composants COTS qu'il intègre dans ses systèmes.

4.1 Comparatif des techniques d'injection de fautes

Dans cette thèse, nous avons proposé un ensemble de techniques expérimentales de caractérisation des modes de défaillance d'un intergiciel de communication. Dans cette section, nous cherchons à comparer ces différentes techniques, en analysant les résultats que nous avons obtenus au chapitre 3. Nous voulons savoir si les modes de défaillance provoqués par les différentes techniques sont équivalents, ou si l'une des techniques est plus efficace pour mettre en défaut les intergiciels candidats. Nous évaluons le coût de mise en œuvre des différentes techniques, en comparant le coût initial de développement de l'outillage logiciel nécessaire aux expériences, et la durée de chaque campagne expérimentale.

Modes de défaillance provoqués. La technique d'injection de fautes qui a provoqué le plus grand nombre de défauts de robustesse est la corruption de messages au niveau IIOP. En effet, c'est la seule technique qui ait provoqué des manifestations de *RemotePropagation*, le mode de défaillance le plus grave, et cette technique a provoqué une proportion relativement élevée d'autres manifestations non robustes : *ApplicationFailure*, *ConnectionHang*, *ServiceHang*, *ServiceCrash*. Les autres techniques d'injection de fautes ont surtout provoqué des manifestations de *ServiceCrash*.

Concernant les modèles de fautes employés, nous avons constaté lors des campagnes de corruption de messages IIOP que les inversions de bits provoquent un plus grand spectre de diagnostics d'erreur que la mise à zéro de deux octets successifs.

Coût de mise en œuvre. La figure 4.1 page suivante compare les différentes techniques d'injection de fautes, du point de vue du coût de développement des outils et du banc de test, puis vis-à-vis de l'exécution d'une campagne employant cette technique. Les techniques les plus coûteuses en terme de développement sont celles qui cherchent à émuler des fautes du logiciel, puisque chaque faute injectée nécessite un effort de conception et de développement. À l'opposé, les techniques qui simulent des fautes matérielles sont peu coûteuses en effort de développement, puisque l'injection de la faute est automatisée.

Cette tendance s'inverse lorsqu'on s'intéresse au temps d'exécution d'une campagne. En effet, les techniques qui simulent des fautes matérielles comportent un grand nombre d'expériences, en particulier lorsqu'il s'agit d'émuler des corruptions mémoire. Les campagnes de ce type peuvent nécessiter plusieurs semaines d'exécution.

Complémentarité des techniques d'injection. Comme nous le verrons au § 4.3, différentes techniques d'injection de fautes peuvent révéler de manière indépendante les mêmes fautes du logiciel sous-jacentes. C'est le cas en particulier des problèmes concernant la gestion de la mémoire, qui ont été révélés par l'injection de corruptions au niveau IIOP et par les saboteurs du système d'exploitation. Si l'ensemble des fautes révélées était identique entre les deux techniques, nous aurions pu encourager l'utilisation exclusive de la technique la moins coûteuse. Nos campagnes montrent qu'aucune des techniques que nous avons testées n'est rendue redondante par une autre.

Technique d'injection	Coût de développement	Coût d'exécution
Corruption mémoire	bas	élevé
Mutation de code	élevé	élevé
Corruption à l'API	moyen	élevé
Injection dans l'exécutif	moyen	bas
Corruption de messages IIOP	bas	élevé
Corruption de messages GIOP	moyen	bas

FIG. 4.1 – *Coût des différentes techniques d'injection de fautes*

4.2 Impact de la version du composant

Nous avons constaté un impact considérable de la version du produit sur les résultats de nos campagnes de caractérisation, autant concernant la gravité des manifestations de fautes que sur la répartition des diagnostics d'erreur. Par exemple, entre la version 3 et la version 4 du courtier *ORBacus*, on constate une différence significative dans le profil des diagnostics d'erreur (*c.f.* la figure 4.2). Cette figure (qui utilise une classification des manifestations de faute différente de celle du chapitre 3) montre que la version la plus récente est la plus robuste, puisque son taux de *ServiceCrash* et *ServiceHang* est nettement plus faible que la version précédente. On voit également que la distribution des diagnostics d'erreur a évolué.

FIG. 4.2 – *Impact de la version pour le candidat ORBacus*

Les expériences que nous menons permettent de quantifier la robustesse d'un logiciel lorsqu'il approche d'un état où il peut être mis en production. Elles peuvent donc fournir des informations pour savoir si une nouvelle version est digne d'être diffusée à ses clients (pour un développeur d'intergiciel) ou mise en production (pour un intégrateur de systèmes critiques). Ces expériences permettent également de mesurer l'impact de procédures de correctifs de fautes qui sont utilisées chez le développeur.

4.3 Analyse a posteriori des fautes du logiciel

Afin de mieux comprendre les manifestations de fautes que nous avons observées, nous avons cherché à identifier la cause des différents comportements non robustes que nous avons observés. Cette analyse s'appuie sur plusieurs sources d'information : la pile d'exécution du SUT à l'instant où il s'arrête, la trace des appels système effectués par le SUT, la trace des messages réseau transmis et l'analyse du code source du SUT, lorsqu'il est disponible.

On peut ainsi identifier des fautes du logiciel dans les cibles :

- les manifestations de *ServiceCrash* du candidat *ORBacus* semblent être dues à l'exécutif C++ employé (ou peut-être au compilateur C++). Le problème concerne l'interaction entre les exceptions C++ et les constructeurs de types paramétrés (les *templates* du langage C++). Le courtier fait appel au constructeur d'un type template, et ce constructeur provoque une violation de segmentation, entraînant l'arrêt brutal du processus par le système d'exploitation. Nous avons observé cette manifestation de faute pour les types paramétrés suivants :

  ```
  OB::VarSeq<CosNaming::NameComponent, CosNaming::OBUnique_Name>

  OB::FixSeq<unsigned char, IOP::ServiceContext::OBUnique_context_data>
  ```

- les manifestations de *ServiceCrash* du candidat *TAO* semblent être majoritairement liées à des problèmes d'allocation mémoire. Le courtier effectue une demande d'allocation mémoire qui échoue, et renvoie donc un pointeur nul. Le courtier ne vérifie pas le résultat avant de poursuivre le traitement, ce qui provoque une violation de la segmentation.
- les manifestations de *ServiceCrash* du candidat *MICO* sont principalement liées à une interaction peu robuste avec le sous-système d'allocation mémoire. En particulier, lorsqu'une opération `realloc` échoue, une assertion dans le code source du courtier provoque l'arrêt immédiat du processus.
- certaines manifestations de *ServiceCrash* du candidat *ORBit* sont dues au suicide du processus lorsqu'une demande d'allocation mémoire échoue. Le code à l'origine du suicide se trouve dans une bibliothèque partagée (la *GLib*) utilisée par ce courtier. On pourrait attribuer ces défaillances à une réutilisation inconsidérée de fonctions, dans un contexte où leur comportement en présence d'erreurs n'est pas adapté.

Cette analyse indique que la gestion de l'allocation mémoire est un point déterminant pour la robustesse d'un intergiciel.

4.3.1 Fautes liées à l'allocation mémoire

Nous allons examiner en détail certains des défauts de robustesse relevés qui proviennent d'une mauvaise interaction du courtier avec le service d'allocation mémoire du système d'exploitation. Ces défauts ont été identifiés dans plusieurs candidats par le même type d'expérience : la corruption d'une partie d'un message IIOP qui identifie une longueur. La corruption de ce champ amène le courtier à faire une demande d'allocation d'un grand tampon mémoire, et la demande échoue.

Prenons les cas du candidat *MICO*. Le courtier détecte que la demande d'allocation mémoire a échoué, mais plutôt que de renvoyer une exception `NO_MEMORY` au client, le courtier se suicide. Considérons l'extrait suivant du fichier `mico/orb/buffer.cc`, qui traite de la gestion de tampons mémoire :

```
CORBA::Octet *
CORBA::Buffer::alloc (ULong sz)
{
    Octet *b = (Octet *)::malloc (sz);
    assert (b);
    return b;
}
```

Le courtier effectue une demande d'allocation mémoire via la fonction `malloc`. En opération normale, cette fonction renvoie un pointeur vers la zone mémoire allouée, mais lorsque la demande n'a pas pu être satisfaite, le pointeur renvoyé est nul. Le programmeur a inséré une assertion pour vérifier que le pointeur renvoyé n'est pas nul ; lorsque l'assertion n'est pas vérifiée, c'est à dire que l'allocation mémoire a échouée, l'exécution du processus est abandonnée, entraînant la défaillance immédiate de tous les objets CORBA logés dans ce processus. Ce code devrait être réécrit en remplaçant la vérification de l'assertion par la levée d'une exception CORBA de type `NO_MEMORY`, ce qui éviterait la défaillance du service, tout en informant le client à l'origine de la requête qu'un problème de ressources a été rencontré sur le serveur.

Le candidat *TAO*, quant à lui, ne vérifie pas si la demande d'allocation mémoire a réussi, ce qui l'amène à dé-référencer un pointeur nul, donc à accéder à un segment mémoire protégé, donc à être tué par le système d'exploitation. Nous avons identifié plusieurs instances de cette faute du logiciel dans le code source de *TAO*.

Le comportement du SUT dépend des options de compilation. Par exemple, nous avons observé que certaines versions du courtier *ORBacus*, compilées avec les options par défaut du fournisseur, déclenchaient un arrêt inopiné lorsqu'une assertion interne est violée :

```
non-compliant application detected:
null value passed as parameter.
assertion failed: value != 0
file ../../include/OB/Stream.h, line 208
```

Ce type de réaction à une assertion interne peut être adapté à un environnement de développement, puisqu'il permet de détecter au plus tôt les erreurs, en fournissant un diagnostic sur le contexte dans lequel s'est produit l'erreur. Par contre, il n'est pas adapté dans un système destiné à la production, puisque le courtier devrait continuer à servir des requêtes provenant d'autres clients que celui qui a provoqué l'erreur.

La rigueur des développeurs (et plus particulièrement des programmeurs) a de fortes conséquences sur la robustesse d'un intergiciel CORBA.

4.4 Influence de l'architecture du courtier

4.4.1 Architecture orientée dæmon

L'implémentation *MICO* se distingue des autres candidats que nous avons ciblés par une architecture partiellement à base de dæmon. Elle s'appuie sur un référentiel d'implémentations (IMR, pour *Implementation Repository*), qui est un processus dæmon. Les différents services s'inscrivent auprès du IMR, en fournissant le nom de l'interface qu'ils sont prêts à implémenter et le nom de l'image exécutable correspondante. Les clients des services s'adressent à l'IMR, qui démarre les serveurs à la demande.

Nous avons constaté dans nos expériences que lorsque la faute injectée provoquait la défaillance du service de désignation et donc l'arrêt des processus de charge, le service de désignation fonctionnait correctement à la fin de l'expérience, à ceci près qu'il a perdu la mémoire des noms enregistrés au cours de l'expérience. En effet, l'IMR constate que le service de désignation est mort, et le redémarre.

Ce type d'architecture comporte donc des points positifs pour la sûreté de fonctionnement. Rappelons toutefois qu'au § 3.2.6 nous avons constaté une défaillance par *RemotePropagation* de ce candidat en présence de fautes multiples, qui semble être imputable à l'IMR et qui affecterait alors *tous* les services enregistrés auprès de lui.

4.4.2 Influence du support exécutif

Nous avons constaté une influence considérable du support d'exécution utilisé par les courtiers sur leur comportement en présence de faute. En particulier, les candidats qui s'appuient sur une machine virtuelle Java présentent des manifestations de faute moins graves que les autres candidats testés.

Nous expliquons cette observation par plusieurs facteurs :

- **meilleure détection d'erreur** : la machine virtuelle Java offre un niveau de détection d'erreur bien meilleur que celui des exécutifs C et C++. Par exemple, une tentative d'accès à une position invalide de tableau provoque la levée immédiate d'une exception `ArrayIndexOutOfBoundsException` dans un environnement Java, alors que dans un programme C++ cette même erreur ne sera pas détectée, et renvoie des valeurs arbitraires.
- **bon niveau de confinement d'erreur** : la machine virtuelle Java fournit un bon niveau de confinement d'erreur entre processus légers, et l'arrêt inopiné d'un processus léger n'entraîne pas l'arrêt d'autres processus légers. Par exemple, un processus léger dans une implémentation en Java qui dé-référence une référence nulle provoquera la levée d'une exception `NullPointerException`, ce qui entraînera l'arrêt de ce processus léger, mais pas celui des autres processus légers. Dans un programme C++, par contre, un processus léger qui dé-référence un pointeur nul va entraîner l'arrêt inopiné de tous les processus légers du courtier.
- **gestion d'erreur par exceptions** : le langage Java et son support d'exécution s'appuient largement sur les exceptions pour gérer les situations d'erreur. Lors de l'appel d'une fonction qui est déclarée comme pouvant lever une exception, l'appelant doit traiter toutes les exceptions qui peuvent être levées par cette fonction[1]. Cette contrainte, qui est vérifiée lors de la compilation d'un programme Java, est à comparer à la gestion d'erreur par code de retour lors de l'interaction d'un programme C++ ou C avec les bibliothèques système de Unix, où le programmeur peut facilement omettre le traitement d'une erreur (*c.f.* la figure 2.2 page 59).
- **gestion automatique de la mémoire** : la machine virtuelle Java fournit un mécanisme de gestion automatique de l'allocation mémoire (le *garbage collector*), ce qui permet d'éviter les erreurs de programmation liés à la gestion de la mémoire. Nous avons vu qu'une large proportion des défauts de robustesse observés dans les courtiers implémentés en langage C ou C++ sont imputables à une mauvaise gestion de l'allocation mémoire.

Rappelons également qu'un défaut de robustesse de l'exécutif C++ GNU semble être à l'origine de certaines des manifestations de *ServiceCrash* que nous avons observées pour le candidat *ORBacus*.

[1] Il existe dans le langage Java une distinction entre exceptions *checked*, qui doivent obligatoirement figurer dans la signature des fonctions qui peuvent les lever, et être traitées par l'appelant, et les exceptions *unchecked* qui échappent à ces contraintes. Toutefois, la majorité des exceptions définies par la bibliothèque standard Java sont obligatoires.

4.5 Vers une méthode d'étalonnage d'intergiciels de communication

L'un des objectifs de nos travaux, évoqué dans le chapitre 1, est de proposer une *méthode d'étalonnage* de la sûreté de fonctionnement d'un composant intergiciel qui permette de comparer la robustesse de différents candidats. Que manque-t'il à notre travail pour qu'un intégrateur de systèmes répartis semi-critiques puisse choisir un intergiciel candidat adapté à ses contraintes ?

Afin d'améliorer la lisibilité de nos résultats, nous avons cherché à développer une *métrique* pour caractériser la robustesse d'un candidat. Cette métrique est basée sur plusieurs critères :

- la proportion d'expériences d'injection de fautes qui provoquent une réaction non satisfaisante du candidat ;
- la sévérité des modes de défaillance résultants ;
- le nombre de fautes sous-jacentes distinctes qui produisent ces défaillances.

Dans la section suivante, nous donnons des valeurs d'une telle métrique de la robustesse pour les courtiers CORBA que nous avons testés.

4.5.1 Comparaison de courtiers CORBA sur étagère

La figure 4.1 page ci-contre rassemble les défauts de robustesse observés dans nos différentes campagnes d'injection de fautes. Nous avons attribué un poids à chaque type de défaut, en fonction de sa gravité vis-à-vis de la sûreté de fonctionnement d'un système qui s'appuie sur un intergiciel de communication. Ce poids est exprimé en pourcentage de la contribution de ce critère à la note globale d'un candidat. Le cumul des défauts observés par candidat, en tenant compte du poids de chaque critère[2], permet d'établir une **classification globale du degré de robustesse** des candidats, représentée sous forme graphique dans la figure 4.3 page 108.

Il faut insister sur le fait que cette classification est incomplète, puisque nous n'avons pas évalué l'impact de toutes les fautes possibles ; il manque en particulier les résultats de campagnes émulant des corruptions mémoire. D'autre part, la classification finale des candidats peut varier considérablement en fonction des poids attribués aux critères ; un intégrateur devra adapter les poids en fonction de l'impact pour son système des différentes manifestations de faute. Enfin, nous avons vu que les résultats de nos campagnes d'injection de fautes peuvent varier considérablement en fonction de la version du composant.

[2]Certaines des valeurs numériques du tableau, tels que les taux de différentes manifestations de fautes, sont issus directement des résultats expérimentaux. D'autres expériences fournissent des points singuliers, que nous avons traduits numériquement en fonction d'une appréciation de la gravité des observations expérimentales. Pour chaque critère, un facteur multiplicatif est calculé en fonction de la moyenne géométrique des valeurs pour ce critère et de sa contribution souhaitée à la somme finale. La note finale pour un candidat est la somme de ses valeurs pour chaque critère, chaque valeur étant pondérée par son facteur multiplicatif.

Critère expérimental	Contribution	*TAO*	*ORBacus*	*omniORB*	*MICO*	*ORBit*	*Sun JDK*	*IBM JDK*
Expériences avec corruptions IIOP (§ 3.2)								
taux de *RemotePropagation*	15%	0	0	0	0.31	0.27	0	0
taux de *ServiceCrash*	10%	5.94	0.56	0	0.16	6.93	0	0.39
taux de *ServiceHang*	8%	0	0	0	1.41	0	0	2.60
taux de *ApplicationFailure*	7%	13.12	6.25	8.75	8.75	5.84	5.00	8.33
taux de *ConnectionHang*	5%	10.42	5.80	4.53	9.06	18.48	25.40	26.82
exceptions `INTERNAL` et `UNKNOWN`	8%	0	0	1.66	0	0	17.02	0.32
richesse des diagnostics d'erreur	5%	0	0	0	0	2	0	0
grande latence et écart type de détection d'erreur	5%	0	0	0	0	0	5	10
taux de `COMPLETED_MAYBE`	4%	21.3	26.2	51.5	40.4	58.6	33.33	53.0
fautes multiples IIOP	5%	0	0	0	1	0	0	0
Expériences avec corruptions GIOP (§ 3.4)								
manifestations de *ServiceCrash*	10%	1	0	0	0	0	0	0
Robustesse de l'interaction avec l'OS (§ 3.5)								
manifestations de *RemotePropagation* et *ApplicationFailure*	11%	3	0	0	0	5	0	0
autres interactions non-robustes	4%	2	1	1	1	2	3	3
Robustesse des outils de développement (§ 3.6)								
taux de *Crash* et *Hang*	3%	22.0	0	20.1	0.1	3.8	13.6	–
Somme pondérée		180	17	32	147	217	102	104

TAB. 4.1 – *Métriques de pondération pour les intergiciels CORBA*

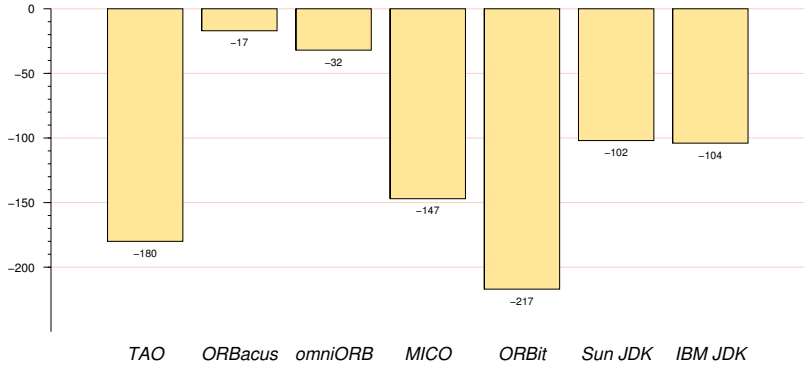

FIG. 4.3 – *Défauts de robustesse constatés pour différents courtiers* CORBA *(Grande valeur absolue implique faible robustesse)*

Nous voyons d'après cette table que le candidat le plus robuste semble être *ORBacus*, suivi de *omniORB* puis des courtiers sur JVM. Les candidats *ORBit* et *TAO* présentent des scores peu flatteurs. Le mauvais score du candidat *TAO* peut surprendre, puisqu'il s'agit d'un logiciel qui est déployé dans plusieurs systèmes semi-critiques, notamment dans le secteur du contrôle-commande militaire. Ce score résulte de comportements non robustes vis-à-vis de plusieurs types de fautes, et nous avons obtenu des résultats concordants pour des campagnes visant les versions 1.2.1 et 1.3.1 de ce candidat. L'analyse du code source de ce candidat permet de vérifier qu'il ne traite pas certains scénarios d'erreur qui sont signalés par le système d'exploitation.

Il est important de noter que la classification finale dépend fortement du poids qui est attribué à chaque critère, et que ces poids (qui représentent la gravité du défaut correspondant) peuvent varier en fonction de la configuration du système. Dans un système qui comporte un mécanisme de détection du gel d'une connexion CORBA (comme ceux spécifiés par le volet *Messaging* de la norme CORBA), il serait judicieux de réduire le poids accordé au critère *ConnectionHang*.

4.5.2 Extension à d'autres intergiciels de communication

Notre approche de caractérisation des modes de défaillance pourrait être appliquée à d'autres catégories d'intergiciels orientés communication :

- à la plate-forme DCOM de Microsoft, et les fonctionnalités orientées communication de plates-formes telles que Java ou *.NET*. Ces plates-formes ont une architecture proche de celle de CORBA, mais simplifiée par l'absence de souci d'interopérabilité entre langages de programmation.

- aux « services web » et leur mécanisme d'appel de procédure *SOAP*. Ces environnements imposent une granularité de la répartition qui se place au niveau d'un service plutôt qu'au niveau d'un objet. Ils sont davantage destinés à un fonctionnement sur des réseaux à grande échelle faiblement couplés qu'à une utilisation sur un réseau local.
- aux serveurs d'application Java qui implémentent la norme *J2EE*. Cette norme précise que les implémentations doivent permettre aux clients d'interroger le service en utilisant le protocole RMI-over-IIOP (le mécanisme d'invocation de méthodes à distance de Java qui utilise IIOP comme transport). Par conséquent, nos tests d'injection de fautes au niveau IIOP seraient directement transposables à ces cibles, à condition de disposer d'une charge applicative adaptée.

Parmis les techniques d'injection de fautes que nous avons proposées, celles qui sont facilement transposables à de nouvelles cibles sont :

- la corruption de zones mémoire : les fautes injectées sont génériques, donc facilement transposables à de nouvelles cibles.
- les corruptions de messages au niveau applicatif : les fautes injectées sont indépendantes du protocole employé, donc cette approche pourrait être appliquée à des protocoles tels que les messages RMI de Java ou des appels à distance de *.NET*, ou le protocole SOAP.
- l'interception d'appels système : à condition que la cible utilise les mêmes appels système, aucune modification de l'injecteur n'est nécessaire.

Il suffit pour ces techniques d'adapter le banc de test, en développant une charge applicative appropriée et des outils d'observation adaptés. Seules des modifications mineures de l'injecteur seraient nécessaires.

Les autres techniques que nous avons décrites dans cette thèse sont plus difficilement adaptables à d'autres intergiciels de communication, puisqu'elles sont fortement liées à des particularités de la plate-forme CORBA. En particulier, les techniques de corruption de messages au niveau session et de mutation de code nécessitent une phase d'élaboration d'un modèle de fautes adapté à la cible, et ne pourraient être appliquées qu'au prix d'un redéveloppement des outils correspondants.

4.6 Intergiciels et malveillances

Dans le chapitre précédent, nous avons vu que la réception d'un message corrompu pouvait entraîner la défaillance immédiate d'un service CORBA. Ces arrêts inopinés de certains des candidats testés sont provoqués par la phase de décodage du contenu des messages IIOP, qui ne prend pas suffisamment de précautions vis-à-vis des données provenant du réseau.

Ce type de faute dans les implémentations de courtier les expose à des attaques de la part de participants malicieux dans le système. En particulier, il est facile de mener des

attaques en *déni de service* : l'exemple que nous donnons en encadré (✠$\mathcal{D}$) montre qu'il est possible de provoquer la défaillance d'un serveur *TAO* en une seule ligne de commande Unix. Il est possible que les défauts de robustesse que nous avons identifiés puissent être exploités pour des attaques plus sévères de type « buffer overflow » (débordement mémoire), qui permettraient à un attaquant distant d'exécuter du code arbitraire sur le nœud cible.

✠$\mathcal{D}$ — **Provoquer la défaillance d'un serveur TAO à distance**

La commande Unix suivante provoque immédiatement la défaillance d'un serveur CORBA qui utilise le courtier *TAO*, version 1.2.1.

```
perl -e 'print pack("H*", "47494F50010201006C000080"), "\n"' \
      | telnet <machine> <port>
```

Cette commande construit une requête GIOP simple, et l'envoie au nœud qui héberge le serveur CORBA. En effet, 47, 49, 4F sont la représentation en hexadécimal des codes ASCII 'G', 'I', 'O', qui forment le début d'une requête GIOP. La partie longueur de l'en-tête GIOP est très grande, ce qui amène *TAO* à demander l'allocation d'un très grand tampon mémoire pour stocker la requête. Cette allocation échoue, mais le courtier ne vérifie pas le résultat, et par conséquent accède à une plage mémoire non autorisée.

Il est intéressant de noter qu'une faute analogue dans des implémentations d'agents SNMP (le *Simple Network Management Protocol*, largement déployé dans les logiciels de supervision et de contrôle des équipements de réseaux de télécommunications), a entraîné en 2002 la diffusion d'un *advisory* du CERT [CERT 2002b]. La faute, qui permettait à des utilisateurs malveillants de provoquer à distance le redémarrage de routeurs, était due à un bogue dans l'implémentation de référence d'un module de décodage du format ASN.1.

En l'état actuel de l'industrie informatique, les vulnérabilités sont généralement découvertes suite à des analyses ad-hoc menées par des utilisateurs malveillants. Une fois que des attaques sont observées, la faille qui est en train d'être exploitée est analysée par les « gentils », et un correctif est annoncé. Cette réaction « en retard » est clairement non-satisfaisante.

Au-delà de techniques d'implémentation plus sûres, qui visent à éliminer les vulnérabilités lors de la production de systèmes [Bellovin 2003], il nous semble nécessaire aujourd'hui de développer des techniques de test qui permettent d'identifier les vulnérabilités avant la mise en production des systèmes. Il nous semble que l'approche par *tests de vulnérabilité*, préconisée par le projet *PROTOS* (*c.f.* § 1.5.2), est un pas positif dans ce sens. Nos expériences d'injection de fautes de protocole réseau sont proches de cet esprit.

Toutefois, cette approche ne permet pas de découvrir tous les types de vulnérabilité : les défauts de conception, les *trapdoors* et autres *backdoors*, et les vulnérabilités qui

exhibent des modes de défaillance complexes (avec plusieurs états erronés non observables depuis l'extérieur avant d'aboutir à une défaillance) ne peuvent pas être révélés par cette technique.

4.7 Mécanismes d'encapsulation pour un intergiciel

Dans cette thèse, nous avons identifié un certain nombre de défauts de robustesse dans des intergiciels CORBA. Pour certains de ces défauts de robustesse, nous avons pu identifier une faute du logiciel à l'origine du problème, facile à corriger. Dans d'autres cas, en revanche, la faute est difficile à identifier (par exemple, puisqu'on n'a pas accès au code source du courtier) ou difficile à corriger (la corriger reviendrait à s'approprier le composant, donc à perdre une bonne partie des attraits de l'intégration d'un composant COTS).

Pour ces fautes résiduelles, non-corrigeables, il faut utiliser d'autres approches pour tolérer la faute. La technique la plus prometteuse dans le contexte d'un composant COTS est l'utilisation de mécanismes architecturaux d'empaquetage (ou de « wrapping » en anglais) [Arlat et al. 2000]. Un wrapper est un logiciel qui forme une couche intermédiaire au niveau de la frontière du COTS avec le reste du système. L'objectif du wrapper est de contrôler les interactions avec le composant empaqueté : il peut contrôler les fonctions qui sont appelées, la validité des paramètres d'entrée, et les résultats produits par le composant empaqueté.

Un premier objectif de l'empaquetage peut être de restreindre les possibilités d'activation du composant empaqueté, afin de limiter son utilisation à ce qui est strictement nécessaire, ou à ce qui a été précédemment validé. Le contrôle de l'activation peut se faire en interdisant l'accès à certains points d'entrée du composant (par exemple, bloquer certaines fonctions dans une interface), ou en filtrant les valeurs de paramètres en entrée. On peut, par exemple, interdire l'activation d'une fonction avec des paramètres dont on sait (par des expériences d'injection de fautes) qu'ils provoquent la défaillance du composant.

Un deuxième objectif des wrappers est d'améliorer les mécanismes de détection ou de confinement d'erreur du composant empaqueté (qui en est souvent totalement dépourvu). On peut, par exemple, implémenter des assertions exécutables, ou des pré- et post-conditions sur les accès au composant, qui détectent des situations anormales. Lorsque la violation d'un prédicat est détectée, on peut – en fonction de la gravité de la défaillance observée – renvoyer un code erreur à l'appelant, ou déclencher une procédure de recouvrement (par exemple redémarrer un processus). Clairement, la puissance des assertions mises en œuvre dépend de la connaissance qu'on a du fonctionnement du composant empaqueté. Lorsque le wrapper peut observer l'état interne et le comportement du composant qu'il surveille, il peut prendre des décisions plus fines sur le comportement à imposer.

Nous avons vu dans les sections précédentes que les problèmes de robustesse varient considérablement, non seulement d'une implémentation à une autre, mais également

de version en version du même produit. Il paraît donc fortement souhaitable que les mécanismes d'encapsulation mis en place soient facilement configurables et « pluggables ».

Nous proposons une méthode pour mettre en œuvre des techniques d'encapsulation d'un courtier CORBA. L'empaquetage peut se faire à trois niveaux : à l'interface entre l'application et l'intergiciel, à l'interface entre l'intergiciel et le système d'exploitation, et au niveau de l'interaction entre le composant cible et les objets distants. Le type de propriétés qui peuvent être assurées, et les techniques pour vérifier et appliquer ces propriétés, sont différents dans chaque cas.

4.7.1 Contrôler l'interaction entre l'application et l'intergiciel

De la même manière qu'un système d'exploitation peut défaillir lorsqu'il est soumis à un paramètre invalide dans un appel système, un courtier peut défaillir lorsque l'application transmet des paramètres invalides dans un appel à l'intergiciel. Considérons, par exemple, la méthode `object_to_string` définie par la spécification CORBA, et qui s'utilise dans un programme C++ de la manière suivante :

```
orb->object_to_string(obj);
```

Certaines implémentations s'arrêtent brutalement lorsque la référence objet `obj` est invalide. Le projet *Ballista* a étudié l'impact de telles fautes [Pan et al. 2001] pour plusieurs implémentations de la norme CORBA, et identifié un nombre considérable de failles dans les produits testés. Les auteurs ont trouvé que l'addition de mécanismes d'encapsulation simples à ces méthodes était suffisant pour améliorer la robustesse du produit vis-à-vis de ces fautes. Leur technique d'implémentation des wrappers est intrusive, puisqu'elle consiste à modifier le code source de l'intergiciel afin d'ajouter des contrôles de la validité des paramètres de certaines méthodes. Il serait toutefois possible de mettre en œuvre des mécanismes de détection d'erreur semblables à l'aide de techniques moins intrusives, tels que le mécanisme de résolution de symboles que nous avons présenté au § 3.5.1.

Malheureusement, cette approche d'encapsulation ne peut être appliquée qu'à un sous-ensemble des fonctionnalités fournies par un courtier CORBA, puisque l'interface entre le code applicatif et le code de l'intergiciel est difficile à identifier. En effet, il existe une liaison très diffuse entre le code applicatif et le code de la souche ou du squelette, passant souvent par de l'héritage au sens de la programmation orientée-objet. La nature de cette liaison dépend à la fois du langage de programmation utilisé au niveau applicatif et des choix effectués par le fournisseur de l'intergiciel, ce qui fait qu'il est difficile d'identifier une interface où l'on pourrait insérer des mécanismes d'encapsulation. Par conséquent, toute forme d'encapsulation qui serait implémentée à ce niveau serait spécifique à un langage de programmation et à un fournisseur d'intergiciel donné (et vraisemblablement à une version de son produit).

4.7.2 Contrôler l'interaction entre l'intergiciel et le système d'exploitation

L'intergiciel dépend de services fournis par le système d'exploitation. Nous avons vu au § 3.5 que cette interaction est un vecteur potentiel de propagation d'erreurs. Lorsque les expériences d'injection de fautes permettent l'identification de fautes du logiciel liées à cette interaction de l'intergiciel avec le système d'exploitation, comme nous l'avons vu pour les implémentations *ORBit* et *TAO*, nous pouvons utiliser un wrapper pour rendre cette interaction plus robuste.

Lorsque l'intergiciel est intégré au système sous la forme d'un COTS, il est peu souhaitable de corriger ces fautes du logiciel en modifiant son code source (voire impossible lorsque le code source n'est pas disponible). Il est possible d'utiliser une technique d'encapsulation basée sur l'interception des appels système (*c.f.* l'annexe B page 129).

Dans le cas particulier des lectures partielles ou interrompues que nous avons observées, il est facile de mettre en place une couche d'interception entre l'intergiciel et le système d'exploitation, qui détecte les cas de figure risquant d'être mal traités par l'intergiciel, et appliquer une stratégie de recouvrement adaptée. Dans le cas d'une interruption par `EINTR`, par exemple, le wrapper peut rejouer l'appel système jusqu'à ce qu'il réussisse. Dans le cas d'une lecture incomplète, le wrapper peut répéter l'appel afin de lire les données restantes avant de transmettre les données à l'intergiciel.

En revanche, il est difficile d'utiliser une technique d'encapsulation pour se prémunir contre une mauvaise utilisation de l'allocation mémoire. L'action de recouvrement à effectuer en cas d'échec d'une demande d'allocation mémoire varie en fonction de l'utilisation prévue par l'application pour cette mémoire. Pour ces fautes de programmation, nous ne voyons pas d'alternative à la correction individuelle de chaque site d'appel à l'allocateur.

4.7.3 Contrôler l'interaction entre l'intergiciel et les objets distants

Un dernier candidat pour la mise en place de mécanismes d'encapsulation pour un intergiciel concerne l'interaction avec les objets distants, via des requêtes IIOP. Nous avons vu au § 3.2 que certains courtiers CORBA sont très sensibles à la corruption de messages IIOP. Deux techniques d'encapsulation peuvent être utilisées pour se prémunir contre l'impact de fautes arrivant via cette interaction avec des objets distants :

- l'addition d'un *CRC* au niveau applicatif à tous les messages IIOP. Ce CRC serait calculé par le courtier à l'origine de la requête, et si le courtier côté serveur détecte une anomalie, il peut lever une exception `COMM_FAILURE` pour indiquer qu'une erreur de communication est survenue.

  Une technique prometteuse pour implémenter ces CRCs de manière transparente à l'application serait d'utiliser les *Portable Interceptor*s, qui ont été intégrés dans la version 2.4 de la norme CORBA. Le CRC serait ajouté au *service context* par un intercepteur côté client, et vérifié par un intercepteur côté serveur. Malheureusement, cette approche n'est pas opérante puisque les intercepteurs côté serveur ne

sont appelés qu'après déballage des paramètres de la requête. Étant donné que la majorité des défaillances que nous avons observées concernaient justement la phase de déballage de la requête, cette technique ne permettrait pas une protection contre les défauts de robustesse que nous avons constatés.

Cette approche ne fournit évidemment pas de protection contre des fautes malveillantes, puisqu'une entité malveillante dans le système peut émettre des requêtes corrompues, mais avec un CRC correct, qui ne seraient pas détectées par les wrappers.

- l'utilisation d'une couche d'encapsulation au niveau de la couche de transport, en utilisant IIOP sur TLS. Cette approche protège contre les fautes accidentelles, puisque le protocole TLS intègre des mécanismes de détection d'erreur supplémentaires à ceux de TCP. Elle protégerait également contre les malveillances, puisque les messages dont la signature n'est pas valide pourraient être rejetés par les serveurs.

  Cette approche est loin d'être parfaite. Le coût en performance de cette technique d'encapsulation n'est pas négligeable. D'autre part, nous avons vu au § 3.5.3 (page 94) que l'utilisation de la bibliothèque *OpenSSL* (qui implémente le chiffrement et l'authentification TLS) n'est elle-même pas exempte de faute.

On le voit, la nature répartie du rôle joué par un intergiciel de communication, et sa manière (presque incestueuse) de s'immiscer dans le code applicatif, font qu'il est difficile d'utiliser des méthodes d'encapsulation pour contrôler toutes les interactions du système avec cette « glue » logicielle.

L'intégration étroite d'un intergiciel CORBA avec le code applicatif qu'il soutient, rend difficile l'encapsulation du courtier dans une zone de confinement qui lui serait dédiée.

4.8 Récapitulatif

Comme nous l'avons vu au chapitre 2, un intergiciel de communication fournit un service qui dépend de la communication entre plusieurs nœuds de calcul, ce qui l'expose à une plus grande variété de classes de fautes que les couches exécutives plus basses. Il est donc nécessaire de mettre en œuvre un ensemble de techniques d'injection de fautes, pour fournir une caractérisation expérimentale qui soit complète.

Nous avons analysé les résultats de campagnes expérimentales qui ont appliqué ces techniques sur des services et des courtiers CORBA, en cherchant à identifier la cause des comportements non robustes observés. Cette analyse indique que la gestion de la mémoire est un aspect déterminant pour la robustesse d'un courtier : une proportion importante des défaillances observées sont dues à l'absence de vérification du bon déroulement d'une demande d'allocation mémoire. Ces résultats illustrent également

l'importance du support exécutif sur la robustesse. Un exécutif langage comme la machine virtuelle Java offre plusieurs avantages.

Ces différentes techniques ont des coûts de développement et d'utilisation différents, et une capacité variable à mettre en défaut la robustesse des intergiciels candidats. Lorsqu'il n'est pas possible d'appliquer l'ensemble des techniques que nous avons proposées, les techniques d'injection de messages IIOP corrompus, associées à l'utilisation de saboteurs au niveau du système d'exploitation, nous paraît comme étant un bon choix de techniques, qui devrait donner une bonne idée des modes de défaillance des différents candidats.

Chapitre 5

Conclusions et perspectives

Nous dressons dans ce chapitre les conclusions de notre étude, à la fois pour les intégrateurs d'intergiciel dans les systèmes répartis semi-critiques, et pour les développeurs d'intergiciel. Nous analysons également les perspectives qui pourraient prolonger cette thèse.

Nous avons vu que les composants intergiciel sont exposés à un ensemble de fautes plus large que d'autres couches exécutives, telles que les systèmes d'exploitation et les exécutifs de langage de programmation. De plus, les fonctionnalités fournies par un intergiciel de communication sont de par leur nature plus difficiles à caractériser, puisqu'elles sont implémentées et utilisées de manière distribuée ; comme d'autre part l'intergiciel doit principalement assurer des propriétés de transparence, ces fonctionnalités sont fournies implicitement plutôt que suite à un appel explicite. Pour toutes ces raisons, une gamme de techniques d'injection de faute est nécessaire afin de caractériser de manière complète le comportement de ces composants logiciels en présence de faute.

Dans ce manuscrit, nous avons analysé les systèmes à base d'intergiciel de communication, afin de proposer un modèle de fautes adapté à leurs caractéristiques et de déterminer une classification de leurs modes de défaillance. À partir de cette étude, nous avons motivé les lignes directrices d'une méthode expérimentale pour évaluer la robustesse et caractériser le comportement en présence de fautes de ces intergiciels de communication. Cette méthode repose sur la définition d'un ensemble de techniques d'injection de fautes, certaines classiques, et d'autres plus originales et plus spécifiques à la cible. Ces techniques ciblent les fautes internes au niveau du matériel et du logiciel, ainsi que les fautes qui se propagent depuis les nœuds distants.

Nous avons focalisé nos travaux sur la référence en matière d'intergiciels de communication, la plate-forme CORBA. Afin de valider notre approche, nous avons mené des campagnes d'injection de fautes ciblant un certain nombre d'implantations de cette norme. Nous avons évalué le comportement des services de désignation et de

diffusion d'événements de différents fournisseurs face à la réception de messages corrompus. Nous avons examiné l'impact de comportements inattendus du système d'exploitation sur différents candidats intergiciels. Nous avons évalué la robustesse des outils de développement d'un fournisseur d'intergiciel en appliquant des techniques de mutation de code à des interfaces IDL.

Notre travail d'analyse de la sûreté de fonctionnement permet de caractériser les modes de défaillance de l'intergiciel, d'identifier les canaux de propagation d'erreur par l'intergiciel, et accessoirement d'analyser ses vulnérabilités en termes de sécurité informatique. En recoupant les résultats de nos différentes campagnes d'injection de fautes, nous avons proposé une métrique permettant d'obtenir une mesure quantitative de la robustesse d'un candidat intergiciel.

Finalement, nous avons démontré comment les travaux de caractérisation peuvent servir à l'identification de fautes de conception et d'implantation dans les composants intergiciels, et nous avons illustré des techniques d'empaquetage permettant de renforcer la robustesse de courtiers du commerce, afin de les rendre mieux adaptés à des applications de type semi-critique.

5.1 Remarques pour les intégrateurs d'intergiciel

Les intergiciels de communication constituent une technologie privilégiée pour l'intégration de systèmes hétérogènes dans un contexte réparti. Ils permettent la communication entre des systèmes implémentés dans des langages de programmation différents, et s'exécutant sur des plates-formes logicielles et matérielles hétérogènes. Toutefois, l'utilisation de cette « glue » logicielle n'est pas sans conséquences sur la robustesse des systèmes. En effet : le code de l'intergiciel peut contenir des fautes du logiciel ; l'intergiciel peut servir de canal de propagation d'erreur dans le système ; par son intégration étroite avec le niveau applicatif, il peut entraîner la défaillance d'objets ou de services.

La propriété de transparence qui est fournie par un intergiciel de communication simplifie la mise en œuvre d'applications réparties, puisqu'elle permet de s'abstraire d'un certain nombre de problèmes liés à la répartition. Toutefois, cette transparence de la répartition des calculs n'est qu'une illusion, et toute illusion a un prix : ici d'abandonner la maîtrise des interactions entre les objets du niveau applicatif et le code de l'intergiciel. L'intergiciel ne se laisse pas facilement apprivoiser ; il est difficile de le confiner dans une boîte dont on maîtrise toutes les entrées et toutes les sorties.

La transparence est l'ennemi du contrôle. Un intergiciel transporte les requêtes entre objets de manière transparente, mais peut également propager les erreurs de manière sournoise.

Pour une utilisation dans un système critique, il nous semble qu'une architecture d'intergiciel offrant moins de transparence serait préférable au niveau d'intégration élevé qui est fourni par CORBA. Le transfert de contrôle vers l'intergiciel devrait être explicite, et il devrait être possible de contrôler l'interaction entre le niveau applicatif et le niveau de l'intergiciel[1].

5.1.1 Guider le choix d'un intergiciel candidat

Pour un intégrateur de systèmes répartis semi-critiques, les résultats de nos expériences peuvent être considérés d'un point de vue **étalonnage de la robustesse** d'un candidat intergiciel, en guidant le choix du candidat le mieux adapté aux contraintes du système hôte. Nous avons vu que le comportement en présence de fautes des différents courtiers CORBA que nous avons testés varie considérablement, certains candidats présentant un bon niveau de robustesse, d'autres des comportements plus critiquables.

De façon surprenante, certaines implémentations CORBA parmi celles les plus utilisées dans l'industrie exhibent un comportement inacceptable en présence de fautes. L'analyse des modes de défaillance n'étant pas une information disponible, elle fait cruellement défaut lors de la sélection d'un candidat. Nous essayons dans cette thèse de lever le voile sur ce type de comportements, ce qui offre probablement beaucoup d'intérêt pour les utilisateurs d'intergiciel.

Il existe une disparité considérable dans le comportement en présence de faute des intergiciels CORBA. Certains candidats sont à proscrire pour une utilisation dans un environnement semi-critique.

Bien entendu, le critère de robustesse que nous avons cherché à quantifier n'est que l'un des facteurs à considérer lors du choix d'un candidat intergiciel : d'autres facteurs tels que la qualité du support technique disponible pour le produit, les éléments de documentation et de conception qui faciliteront la constitution du dossier de validation, le prix des licences pour le produit, et même la santé financière du fournisseur, doivent également être pris en compte [Arlat et al. 2000].

[1] Notons toutefois que ce recours à un transfert explicite de contrôle du niveau applicatif au niveau intergiciel pourrait conduire à un plus fort taux de fautes de programmation, puisque plusieurs études ont trouvé que le nombre de fautes du logiciel par ligne de code écrite par les programmeurs est approximativement constant. Un compromis satisfaisant entre, d'une part un meilleur confinement des erreurs provenant de l'intergiciel, et d'autre part la complexité de la tâche de programmation, n'est pas facile à établir.

5.1.2 Mécanismes de détection et de confinement d'erreur préconisés

Une fois le choix d'un candidat effectué, nos campagnes de caractérisation du comportement en présence de fautes fournissent des informations sur les solutions architecturales qui sont adaptées au confinement d'erreur et au masquage de fautes, et sur les mécanismes de détection et de recouvrement d'erreur qu'il conviendrait de mettre en œuvre.

Apports de la norme FT-CORBA. La norme *Fault Tolerant CORBA* [OMG 2002a] spécifie un ensemble d'interfaces qui peuvent être utilisées pour la mise en œuvre de techniques de tolérance aux fautes dans un système basé sur CORBA. Chaque nœud dans le système comporte un *FailureDetector* chargé de détecter la défaillance d'objets CORBA hébergés sur ce nœud. À la place des IOR, les clients peuvent utiliser des *IOGR* qui permettent de désigner un groupe d'objets. La norme FT-CORBA est basée sur l'utilisation de techniques de réplication, et repose donc sur **l'hypothèse de silence sur défaillance** des serveurs CORBA.

Nos expériences ne nous permettent pas d'affirmer que les intergiciels CORBA du marché respectent cette hypothèse dans tous les cas. Concernant les services de désignation et de diffusion d'événements que nous avons testés, nous pouvons affirmer que deux des candidats testés ne sont **pas** silencieux sur défaillance, puisqu'ils ont provoqué des situations de *RemotePropagation*. Les trois autres candidats semblent assurer un bon taux de couverture de cet hypothèse.

Détection d'erreurs. Concernant les fautes qui provoquent des situations de *Detection-Exception* et de *ServiceCrash*, nous n'avons besoin d'aucun mécanisme supplémentaire de détection d'erreur, puisque le client est immédiatement averti qu'une erreur est survenue au cours de son interaction avec l'objet distant. Considérons maintenant les mécanismes de détection d'erreur supplémentaires qui pourraient permettre un traitement acceptable des cas restants.

La détection des situations de *ConnectionHang* et de *ServiceHang* devrait pouvoir se faire à l'aide de mécanismes intégrés au courtier. C'est le cas par exemple de courtiers qui implémentent le volet *Messaging* de la norme CORBA, permettant à un objet client de spécifier une durée maximum de traitement d'une requête avant levée d'une exception `TIMEOUT`. D'autres courtiers proposent des mécanismes propriétaires équivalents.

Les autres situations sont les catégories *RemotePropagation* et *ApplicationFailure*. Rappelons que nous avons observé ces manifestations de faute uniquement lors des expériences de corruption de message (*c.f.* § 3.2). Pour se prémunir contre ces fautes de communication transitoires, nous préconisons l'utilisation d'un **CRC au niveau IIOP**. Dans la mesure où la majorité des défauts de robustesse observés concernent la phase de déballage des requêtes inter-objet, ce type de mécanisme n'offre pas une protection définitive contre les effets nuisibles de messages corrompus. En effet, lorsqu'il est implémenté à l'aide d'un mécanisme tel que les *Portable Interceptors* de la norme CORBA, l'analyse du CRC au niveau applicatif intervient trop tard dans la phase de

traitement d'un message. Toutefois, il devrait permettre de détecter les fautes qui ont conduit à une propagation d'erreur.

La mise en place d'un CRC au niveau de la couche session (en complément des mécanismes de détection d'erreur intégrés à la couche de transport) a déjà été implanté lors de la conception du protocole *iSCSI*, qui est destiné à la sauvegarde de données sur des réseaux IP [Krueger & Haagens 2002]. Ce protocole utilise un CRC de 32 bits en complément du checksum TCP, pour assurer l'intégrité des données enregistrées sur le support stable.

Mécanismes de recouvrement d'erreur. Les fautes qui provoquent des situations de *DetectionException* peuvent être masquées par des mécanismes de recouvrement d'erreur au niveau applicatif. Il est nécessaire pour cela que les serveurs CORBA soient répliqués, et que le serveur primaire et le secondaire synchronisent régulièrement leur état. Lorsque le champ *completion status* de l'exception indique que l'opération invoquée n'a pas été exécutée, le client peut décider de relancer l'appel, ou de basculer vers un serveur secondaire. Si on accepte l'hypothèse de silence sur défaillance des serveurs CORBA, ce recouvrement par retransmission de la requête peut également être utilisé pour les situations de *ServiceCrash*, et de *ServiceHang* lorsque ceux-ci sont détectés par une exception `TIMEOUT`.

5.2 Remarques pour les développeurs d'intergiciel

Nos travaux de caractérisation du comportement en présence de fautes des composants intergiciel sont utiles pour leurs fournisseurs, puisqu'ils permettent d'identifier des points faibles dans leurs produits. Des expériences d'injection de fautes peuvent être intégrées à la phase de test du produit, afin de vérifier que les développements réalisés depuis la dernière version du logiciel n'ont pas introduit des régressions au niveau de la robustesse du logiciel.

Comme nous l'avons vu au § 4.3, notre banc de test permet de collecter des informations pour faciliter le diagnostic des défaillances observées au cours d'une campagne d'injection de fautes, ce qui permet d'identifier des fautes du logiciel dans les courtiers. En particulier, il est possible de :

- détecter les parties dans le code de l'intergiciel où l'on ne vérifie pas suffisamment la validité d'informations obtenues depuis l'extérieur (par exemple lors du déballage des requêtes IIOP) ;
- détecter des parties dans le code source de l'intergiciel où les appels aux services du système d'exploitation ne comportent pas tous les chemins de traitement d'erreur nécessaires (*c.f.* § 3.5 à la page 93).

Ainsi, des tests de robustesse pourraient être intégrés à la phase de test du produit avant sa mise sur le marché. Une fois développés le banc de test et les outils d'analyse des données, le coût d'exécution d'une campagne d'injection de fautes n'est pas prohibitif ; des résultats utiles peuvent être obtenus après quelques dizaines d'heures d'expériences. Nous pensons que cette approche permettrait d'améliorer la qualité et la robustesse des intergiciels COTS disponibles sur le marché.

En ce qui concerne l'**influence de l'architecture** de l'intergiciel sur sa robustesse, et sur la capacité à contrôler ses interactions avec d'autres parties du système, nous avons vus que l'architecture partiellement à base de dæmon de *MICO* comporte des avantages vis-à-vis de la sûreté de fonctionnement. Nous avons également observé que le fait de s'appuyer sur un exécutif langage qui intègre des mécanismes de détection d'erreur, qui gère la mémoire de manière automatique, et qui fournit un bon niveau de confinement d'erreur entre processus légers, est un facteur positif pour la robustesse du service.

5.3 Perspectives des travaux

Nos travaux proposent plusieurs possibilités d'investigation qui mériteraient des travaux plus poussés, concernant aussi bien les techniques de caractérisation expérimentale que la mise en œuvre de mécanismes d'encapsulation :

- Comparer l'impact de corruptions mémoire avec celles des autres techniques d'injection de fautes ;
- Développer des outils d'analyse du code source qui permettent de détecter statiquement les sites d'appel aux services du système d'exploitation où le traitement d'erreur est incomplet, c'est-à-dire où l'application ne traite pas tous les codes d'erreurs possibles. Ces outils pourraient être basés sur des techniques de *program slicing* [Chou et al. 2000], ou s'appuyer sur les analyses de flux de contrôle implémentés par la phase d'optimisation d'un compilateur [Bigrigg & Vos 2002].
- Chercher des corrélations entre des métriques de qualité et de complexité du logiciel et notre métrique de robustesse ;
- Étudier l'impact de l'intergiciel sur le système d'exploitation. Par exemple, l'intergiciel DCOM est plus fortement couplé au noyau win32 que le sont les intergiciels CORBA que nous avons testés sur des plates-formes Unix ; il se pourrait que ceci se traduise par des manifestations de *KernelCrash* en présence de fautes.

Améliorer les techniques d'encapsulation. Nous avons vu que la nature entremêlée de la relation entre l'intergiciel et le niveau applicatif rend difficile le contrôle de toutes les interactions entre le niveau applicatif et l'intergiciel, et cause ainsi un degré de confinement d'un intergiciel COTS peu satisfaisant. Pour remédier à ce problème, il serait intéressant de développer des techniques visant à empêcher la propagation d'erreur entre l'intergiciel et le niveau applicatif. Le code de l'intergiciel pourrait être confiné dans une zone séparée de l'application. Les interactions entre ces deux zones de confinement seraient contrôlées par des souches et squelettes « durcies »

pour la tolérance aux fautes. Dans une telle architecture, l'arrêt inopiné ou le gel de l'intergiciel pourrait être détecté par le code applicatif, lui permettant d'exécuter des actions de recouvrement d'erreur pour tolérer la défaillance du logiciel COTS.

Étude de systèmes en opération. Des recherches pourraient également être menées en étudiant les types de fautes et de défaillances qui sont observées en opération sur des systèmes à base d'intergiciel. Ce retour sur l'exploitation opérationnelle permettrait de connaître la nature et la fréquence des défaillances, ainsi que le type d'utilisation qui a conduit l'intergiciel dans un état défaillant. Il permettrait également de valider la représentativité du modèle de fautes que nous avons utilisé dans cette thèse.

Annexe A

Construction d'interfaces synthétiques

Dans cette annexe, nous décrivons notre technique automatisée de construction d'interfaces IDL synthétiques. Rappelons que ces interfaces sont crées spécifiquement pour nos expériences qui utilisent des techniques de mutation de code pour évaluer la robustesse du compilateur IDL et le référentiel d'interfaces d'un fournisseur.

La technique que nous utilisons pour générer des interfaces est la réécriture stochastique de chaînes. Le point de départ de cette technique est la grammaire du langage OMG IDL, extraite de la spécification CORBA [OMG 2002c]. La figure A page suivante donne un extrait de cette spécification, décrite sous forme BNF. Cette grammaire est constituée d'un ensemble de symboles non-terminaux, qui sont représentés sous la forme ⟨*definition*⟩, et de symboles terminaux qui sont représentés dans une police de machine à écrire `interface`. La grammaire est composée d'un ensemble de *règles de production*, qui décrivent la transformation d'un symbole non-terminal en suite de symboles.

En ajoutant une probabilité à chaque transition de la grammaire, nous obtenons une grammaire de réécriture stochastique. Une *grammaire de réécriture stochastique* est un quadruplet $G = (V_t, V_a, S, P)$, où :

- le vocabulaire V_t est un ensemble fini non vide de symboles terminaux, qui représente les mots du langage ;
- le vocabulaire V_a est un ensemble de symboles auxiliaires qui représentent les non-terminaux ;
- S est un élément distingué de V_a qui représente le point de départ du processus de génération ;

⟨*specification*⟩ ::= ⟨*definition*⟩ '{' ⟨*specification*⟩ '}'

⟨*definition*⟩ ::= (⟨*type_dcl*⟩ | ⟨*const_dcl*⟩ | ⟨*except_dcl*⟩ | ⟨*interface*⟩ | ⟨*module*⟩) ';'

⟨*interface*⟩ ::= ⟨*interface_dcl*⟩ | ⟨*forward_dcl*⟩

⟨*interface_dcl*⟩ ::= ⟨*interface_header*⟩ { ⟨*interface_body*⟩ }

⟨*interface_header*⟩ ::= `interface` ⟨*identifier*⟩ [⟨*inheritance_spec*⟩]

⟨*module*⟩ ::= `module` ⟨*identifier*⟩ { ⟨*specification*⟩ }

⟨*type_dcl*⟩ ::= `typedef` ⟨*type_declarator*⟩ | ⟨*struct_type*⟩ | ⟨*union_type*⟩ | ⟨*enum_type*⟩

FIG. A.1 – *Extrait de la grammaire OMG IDL sous forme BNF*

- P est un ensemble de règles de production. Une règle de production est de la forme $\alpha \rightarrow \beta\ (p)$, où $\alpha \in V_a$ et $\beta \in (V_t \cup V_a)^\star$. Chaque règle a une probabilité p qui lui est associée, et la somme des probabilités des règles pour chaque non terminal est égal à 1.

Cette grammaire définit un *langage*, c'est-à-dire un ensemble de phrases qui peuvent être obtenues en appliquant les règles de production au symbole de départ S. Dans le cas qui nous intéresse, cet ensemble de phrases est infini. Chaque phrase représente un squelette d'interface OMG IDL.

Notre processus de génération automatique d'interfaces synthétiques consiste donc à partir du symbole de départ de la grammaire, et à sélectionner aléatoirement l'une des règles de production qui peut s'appliquer à ce symbole (en fonction des probabilités des différentes règles). On obtient alors une phrase composée de symboles non-terminaux et terminaux, et on applique récursivement le même processus de réécriture en parallèle pour chaque symbole non-terminal. Le processus de génération s'arrête lorsque la phrase ne contient plus que des symboles terminaux.

Les symboles terminaux dans notre grammaire sont de deux types :

- des mots clefs du langage IDL (comme par exemple `interface`, `oneway` et `exception`) et des éléments syntaxiques (comme ; et {).
- des représentants de classes sémantiques, comme `identifier`, `integer-literal` et `character-literal`. Pour chaque classe sémantique, nous avons défini une fonction de génération d'éléments de cette classe. Par exemple, la fonction de génération associée au symbole terminal `integer-literal` génère aléatoirement un nombre entier et la représente sous l'une des syntaxes définies par la spécification OMG IDL : 42, 038 ou 0x23.

Pour transformer la liste d'éléments en fichier IDL, on parcours séquentiellement la liste en insérant les mots clés sans modification dans le fichier, et en appliquant la fonction de génération à chaque représentant d'une classe sémantique. Certains des opérateurs de mutation que nous avons appliqués dans nos expériences consistent à modifier les fonctions de génération de ces classes syntaxique ; nous avons par exemple utilisé un opérateur de mutation qui génère des identifieurs qui ne respectent pas la spécification OMG IDL, en incluant des caractères non autorisés.

A.1 Difficultés de la technique

Cette technique de génération de programmes à partir d'une grammaire stochastique pose deux types de problèmes :

- déterminer les probabilités à affecter aux règles de transition ;
- détecter les « exécutions » de la grammaire qui ne terminent pas.

Le choix des probabilités influence considérablement la nature des interfaces obtenues, et un choix peu judicieux entraîne la génération de candidats qui ressemblent peu à des interfaces « réelles », et qui couvrent mal l'ensemble des transitions de la grammaire. Il existe des techniques qui, à partir d'une grammaire stochastique non pondérée et d'un ensemble de phrases réels faisant partie du langage défini par cette grammaire, de déterminer des probabilités de transition qui correspondent aux phrases réels. Ces techniques sont issues des recherches sur le traitement informatique du langage naturel. Dans le cadre de notre étude, nous avons déterminé les poids manuellement, en adaptant les probabilités jusqu'à obtenir des résultats acceptables.

Annexe B

Mise en œuvre des saboteurs

Dans cette annexe, nous décrivons la technique d'implémentation du mécanisme d'interception des appels système décrit au § 3.5.1. Nous avons utilisé cette technique pour simuler des comportements inattendus ou erronés du système d'exploitation, afin d'évaluer la probabilité de propagation d'erreur via l'interaction entre l'intergiciel et les niveaux exécutifs sur lesquels il s'appuie.

Les techniques qui peuvent être utilisés pour l'implémentation de cette couche d'interception d'appels de service sont spécifiques à chaque système d'exploitation. Sur le système *GNU/Linux* que nous avons ciblé, il existe plusieurs moyens qui pourraient être utilisés :

- modifier le code source de l'intergiciel afin d'intervenir au niveau des sites d'appel au système d'exploitation. Cette technique permet d'avoir une bonne visibilité sur le contexte dans lequel s'effectue un appel de service du système d'exploitation, mais elle nécessite l'accès au code source.
- exploiter le mécanisme de résolution de symboles utilisé pour lier une application aux bibliothèques partagées qu'elle utilise.
- modifier la bibliothèque système (la *libc*), qui constitue le point d'entrée vers le noyau du système d'exploitation.
- utiliser le mécanisme *ptrace*, qui est utilisé par les débuggeurs pour contrôler l'exécution d'un programme.
- modifier le noyau du système d'exploitation.

Pour des raisons de facilité de mise en œuvre, nous avons choisi d'utiliser la technique par résolution de symboles. Cette approche consiste à créer une bibliothèque partagée d'interposition qui fournit une implémentations des fonctions que nous souhaitons intercepter. Cette bibliothèque d'interposition est chargée en premier lors du démarrage de l'application (grâce au mécanisme `LD_PRELOAD` de l'éditeur de liens dynamique), de sorte que les symboles qu'elle exporte sont utilisés en priorité à ceux de la bibliothèque

système. Ainsi, nos implémentations des fonctions ciblées, que nous appelons des *saboteurs* peuvent intercepter les demandes de service du SUT, avant qu'elles n'atteignent la bibliothèque système.

Les fautes émulées par nos saboteurs sont transitoires ou intermittentes. La faute est déclenchée à chaque fois que le SUT effectue l'une des demandes de service ciblées par le saboteur, à condition que le saboteur soit *autorisé*. L'état par défaut du saboteur est non autorisé, en quel cas il est passif, propageant toutes les demandes de service qu'il intercepte vers le système d'exploitation. Nous avons implanté plusieurs techniques de déclenchement de l'autorisation : la communication par mémoire partagée entre le contrôleur et le saboteur, le test de présence d'un fichier, ou le déclenchement après un nombre prédéterminé d'exécutions de l'appel système.

Un saboteur est généré à partir d'un ensemble de paramètres :

- la signature de l'appel système à viser ;
- le mode d'autorisation (présence d'un fichier, nombre d'exécutions) ;
- la nature de la faute : transitoire, ou intermittente ;
- une description de l'action à prendre lorsque la faute est déclenchée. Pour les saboteurs les plus simples, il suffit de renvoyer une code erreur particulier et de positionner la variable `errno`. Pour d'autres saboteurs, l'un des paramètres peut être modifié avant la propagation de l'appel vers le système d'exploitation.

Notre application de génération de saboteurs est implantée en langage Common Lisp, et combine ces différents paramètres en utilisant l'héritage multiple. Le programme génère une fonction C par saboteur, et fait appel au compilateur C pour obtenir une bibliothèque dynamique d'interposition. Cette application permet une description de relativement haut niveau des saboteurs. Par exemple, l'extrait suivant :

```
(defsaboteur "recv" repeat-errno/stat-triggered
     :errno unix:EINTR
     :repeat 5)
```

définit un saboteur visant l'appel système `recv`. Ce saboteur est déclenché par la présence d'un fichier sur le disque local. La faute émulée est intermittente, puisque le saboteur n'est autorisé à s'exécuter que cinq fois. Son comportement lorsqu'il est autorisé est de positionner la variable `errno` à `EINTR` puis de retourner le contrôle directement à l'appelant.

B.1 Défauts de ce mode d'interception

Ce mode d'interception est facile à implémenter, puisqu'il ne nécessite aucune modification ni du système d'exploitation (que ce soit son noyau ou sa bibliothèque système) ni de l'intergiciel. Il présente toutefois un certain nombre d'inconvénients :

- **degré d'intrusivité temporelle** : certains des modes d'autorisation des saboteurs perturbent considérablement l'activité de l'intergiciel cible. Par exemple, la vérification de la présence d'un fichier nécessite un appel système ; la bibliothèque d'interposition a donc pour effet de doubler le coût de chaque appel système qu'elle cible.
- **information sémantique réduit** : cette technique d'interception fournit peu de visibilité sur l'activité du SUT au moment où la faute est injectée. Par exemple, un appel système `write` peut être lié à une demande de résolution de nom DNS, ou à une écriture vers le support stable à des fins de logging, ou encore à une interaction avec un objet distant. Cette limitation entraîne une difficulté pour bien cibler les activités du SUT qui doivent être perturbées par une campagne d'injection de fautes, et limite la pertinence des analyses qui peuvent être menées.
- **mécanisme contournable** : une application peut accéder au noyau sans passer par la bibliothèque système, contournant ainsi notre bibliothèque d'interposition. Cette technique ne peut donc pas être utilisée pour la mise en place de procédures liées à la sécurité informatique.

Nous pensons qu'il serait intéressant d'étudier d'autres moyens d'implantation des saboteurs qui (au prix d'un coût de développement plus important) permettraient de s'affranchir de ces limitations.

Bibliographie

[Arlat et al. 2000] Arlat, J., Blanquart, J., Boyer, T., Crouzet, Y., Durand, M., Fabre, J., M.Founau, Kaaniche, M., Kanoun, K., Meur, P., Mazet, C., Powell, D., Scheerens, F., Thevenod-Fosse, P., & Waeselynck, H. (2000). *Composants logiciels et sûreté de fonctionnement : Intégration de COTS*. Hermès. 7, 111, 119

[Arlat et al. 1993] Arlat, J., Costes, A., Crouzet, Y., Laprie, J.-C., & Powell, D. (1993). Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Transactions on Computers*, 42(8) :913–923. 27

[Arlat et al. 2002] Arlat, J., Fabre, J.-C., Rodríguez, M., & Salles, F. (2002). Dependability of COTS microkernel-based systems. *IEEE Transactions on Computer Systems*, 51(2) :138–163. 29

[Bellovin 2003] Bellovin, S. (2003). RFC 3514 : The security flag in the IPv4 header. Documentation, Internet Engineering Taskforce. 110

[Bigrigg & Vos 2002] Bigrigg, M. W. & Vos, J. J. (2002). The set-check-use methodology for detecting error propagation failures in I/O routines. In *Proceedings of the Workshop on Dependability Benchmarking (DSN'2002)*, Washington D.C. IEEE Computer Society Press. 122

[Carreira et al. 1998] Carreira, J., Madeira, H., & Silva, J. G. (1998). Xception : A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2) :125–136. 29

[CERT 2002a] CERT (2002a). Integer overflow in XDR library. 7

[CERT 2002b] CERT (2002b). Multiple vulnerabilities in many implementations of the simple network management protocol (SNMP). 34, 110

[CERT 2003] CERT (2003). Buffer overflow in Microsoft RPC. 7

[Chevalley & Thévenod-Fosse 2002] Chevalley, P. & Thévenod-Fosse, P. (2002). A mutation analysis tool for java programs. *International Journal on Software Tools for Technology Transfer (STTT)*. Also available as LAAS Report 01356. 52

[Chevochot & Puaut 2001] Chevochot, P. & Puaut, I. (2001). Experimental evaluation of the fail-silent behavior of a distributed real-time run-time support built from COTS components. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01)*, pages 304–313, Washington - Brussels - Tokyo. IEEE. 29

[Cheynet et al. 2000] Cheynet, P., Nicolescu, B., Velazco, R., Rebaudengo, M., Reorda, M. S., & Violante, M. (2000). Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. *IEEE Transactions on Nuclear Science*, 47(6) :2231–2236. 50

[Chou et al. 2000] Chou, A., Chelf, B., Engler, D., & Heinrich, M. (2000). Using meta-level compilation to check FLASH protocol code. *ACM SIGPLAN Notices*, 35(11) :59–70. 122

[Chung et al. 1999] Chung, P. E., Lee, W., Shih, J., Yajnik, S., & Huang, Y. (1999). Fault-injection experiments for distributed objects. In IEEE, editor, *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 88–97. 29, 30, 63

[Costa et al. 2000] Costa, D., Rilho, T., & Madeira, H. (2000). Joint evaluation of performance and robustness of a COTS DBMS through fault-injection. In *Proc. of the International Conference on Dependable Systems and Networks (DSN'2000)*, pages 251–260. IEEE Computer Society Press. 29

[Daran & Thévenod-Fosse 1996] Daran, M. & Thévenod-Fosse, P. (1996). Software error analysis : a real case study involving real faults and mutations. In Zeil, S. J., editor, *Proc. of the International Symposium on Software Testing and analysis*, pages 158–171, New York. ACM Press. 52

[Dawson & Jahanian 1995] Dawson, S. & Jahanian, F. (1995). Probing and fault injection of dependable distributed protocols. *The Computer Journal*, 38(4) :286–300. 33, 65

[Floering et al. 2002] Floering, B., Brothers, B., Kalbarczyk, Z., & Iyer, R. (2002). An adaptive architecture for monitoring and failure analysis of high-speed networks. In *Proc. of the International Conference on Dependable Systems and Networks (DSN'2002)*. IEEE Computer Society Press. 64

[Fuchs 1998] Fuchs, E. (1998). Validating the fail-silence of the MARS architecture. In *Proc. Int. Working Conference on Dependable Computing for Critical Applications : DCCA-6*, pages 225–247. IEEE Computer Society Press. 28, 50

[ISO 1995] ISO (1995). Reference model of open distributed processing. Technical Report 10746-1 – 107464, ISO/IEC JTC1/SC21. ITU-T Rec. X.901 - X.904. 13

[Jarboui 2003] Jarboui, T. (2003). *Sûreté de Fonctionnement de Systèmes Informatiques : Étalonnage et représentativité des fautes*. PhD thesis, LAAS-CNRS, Toulouse, France. 43

[Kaksonen et al. 2001] Kaksonen, R., Laakso, M., & Takanen, A. (2001). Software security assessment through specification mutations and fault injection. In *Proceedings of Communications and Multimedia Security Issues of the New Century / IFIP TC6/TC11 Fifth Joint Working Conference on Communications and Multimedia Security (CMS'01)*, Darmstadt, Germany. 33

[Kanawati et al. 1992] Kanawati, G. A., Kanawati, N. A., & Abraham, J. A. (1992). FERRARI : A tool for the validation of system dependability properties. In *Proc. of the International Symposium on Fault-Tolerant Computing (FTCS'92)*, pages 336–344, Boston, MA. IEEE Computer Society Press. 28

[Karlsson et al. 1998] Karlsson, J., Folkesson, P., Arlat, J., Crouzet, Y., Leber, G., & Reisinger, J. (1998). Application of three physical fault injection techniques to the experimental assessment of the MARS architecture. In *Proc. Working Conference on Dependable Computing for Critical Applications : DCCA-6*, pages 267–287. IEEE Computer Society Press. 28

[Koopman & DeVale 1999] Koopman, P. J. & DeVale, J. (1999). Comparing the robustness of POSIX operating systems. In *Proc. of the International Symposium on Fault-Tolerant Computing (FTCS'29)*, pages 30–37, Los Alamitos, CA, USA. IEEE Computer Society Press. 31, 57

[Krueger & Haagens 2002] Krueger, M. & Haagens, R. (2002). RFC 3347 : Small computer systems interface protocol over the internet (iscsi) requirements and design considerations. Informational, Internet Engineering Taskforce. 121

[Laprie 1995] Laprie, J. C. (1995). Dependable computing : Concepts, limits, challenges. In *25th IEEE International Symposium on Fault-Tolerant Computing - Special Issue*, pages 42–54. IEEE Computer Society Press. 10

[Madeira et al. 2000] Madeira, H., Costa, D., & Vieira, M. (2000). On the emulation of software faults by software fault injection. In *Proc. of the International Conference on Dependable Systems and Networks (DSN'2000)*, pages 417–426. IEEE Computer Society Press. 28

[Marsden & Fabre 2001] Marsden, E. & Fabre, J.-C. (2001). Failure mode analysis of CORBA Service Implementations. In *Proc. of the International Conference on Distributed Systems Platforms (Middleware'2001)*, volume 2218 of *Lecture Notes in Computer Science*. Springer-Verlag. 85

[Marsden et al. 2002a] Marsden, E., Fabre, J.-C., & Arlat, J. (2002a). Characterization approaches for CORBA systems by fault injection. In *Proc. of the Workshop on Dependable Middleware-Based Systems*, Washington DC. Springer-Verlag. 44

[Marsden et al. 2002b] Marsden, E., Fabre, J.-C., & Arlat, J. (2002b). Dependability of CORBA Systems : Service characterization by fault injection. In *Proc. of 21st IEEE Symposium on Reliable Distributed Systems (SRDS2002)*, Osaka, Japan. IEEE Computer Society Press. 88

[Marsden et al. 2002c] Marsden, E., Perrot, N., Fabre, J.-C., & Arlat, J. (2002c). Dependability characterization of middleware services. In Kleinjohann, B., Kim, K. H., Kleinjohann, L., & Rettberg, A., editors, *Design and Analysis of Distributed Embedded Systems, IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002)*, volume 219 of *IFIP Conference Proceedings*, Montréal, Québec, Canada. Kluwer. 73

[Miller et al. 1990] Miller, B. P., Fredriksen, L., & So, B. (1990). An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12) :32–44. 31

[OMG 2002a] OMG (2002a). CORBA 3.0 : Fault tolerant CORBA. Documentation available at www.omg.org, Object Management Group. 120

[OMG 2002b] OMG (2002b). CORBA 3.0 : General Inter-ORB protocol. Documentation available at www.omg.org, Object Management Group. 63

[OMG 2002c] OMG (2002c). CORBA 3.0 : IDL syntax and semantics. Documentation available at www.omg.org, Object Management Group. 125

[Pan et al. 2001] Pan, J., Koopman, P., Siewiorek, D., Huang, Y., Gruber, R., & Jiang, M. L. (2001). Robustness testing and hardening of CORBA ORB implementations. In *Proc. of the International Conference on Dependable Systems and Networks (DSN'2001)*. IEEE Computer Society Press. 32, 55, 98, 112

[Rimén et al. 1994] Rimén, M., Ohlsson, J., & Torin, J. (1994). On microprocessor error behavior modeling. In *Proc. of the International Symposium on Fault-Tolerant Computing*, pages 76–85. IEEE Computer Society Press. 28

[Ruiz-García 2002] Ruiz-García, J.-C. (2002). *Contribution à la validation des systèmes réflexifs tolérants aux fautes : stratégie de test de protocoles à métaobjets*. PhD thesis, LAAS-CNRS, Toulouse, France. 96

[Salles 1999] Salles, F. (1999). *Sûreté de Fonctionnement des logiciels exécutifs à base de micronoyau : analyse des modes de défaillance et confinement des erreurs*. PhD thesis, LAAS-CNRS, Toulouse, France. 8

[Stone & Partridge 2000] Stone, J. & Partridge, C. (2000). When the CRC and TCP checksum disagree. In *Proc. of the ACM SIGCOMM Conference*, pages 309–319. 63, 64, 76

[Sullivan & Chillarege 1991] Sullivan, M. & Chillarege, R. (1991). Software defects and their impact on system availability - a study of field failures in operating systems. *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, pages 2–9. 44, 53

[Tay & Ananda 1990] Tay, B. H. & Ananda, A. L. (1990). A Survey of Remote Procedure Calls. *ACM Operating Systems Review*, 24(3) :68–79. 17

[Thévenod-Fosse et al. 1995] Thévenod-Fosse, P., Waeselynck, H., & Crouzet, Y. (1995). Software statistical testing. In Randell, B., Laprie, J.-C., Kopetz, H., & Littlewood, B., editors, *Predictably Dependable Computing Systems*, pages 253–272. Springer-Verlag. 43, 56

[Voas & McGraw 1997] Voas, J. & McGraw, G. (1997). *Software Fault Injection*. John Wiley and Sons. 52

Glossaire

CDR Le *Common Data Representation* est un ensemble de règles décrivant la manière dont les différents types OMG IDL doivent être emballés, ou mis à plat sous forme de séquence d'octets. (page 21)

CERT Le Centre de Coordination du CERT est une organisation qui surveille les tendances en terme d'attaques contre les systèmes informatiques en réseau, et publie des bulletins d'alerte sur des vulnérabilités logicielles connues. (page 7)

COTS Un composant *Commercial-Off-The-Shelf*, ou *sur étagère*, est un composant qui n'a pas été développé en interne pour les besoins spécifiques du projet auquel il est intégré. (page 7)

Courtier Le courtier (ou *ORB*, pour *Object Request Broker* en anglais), joue le rôle de bus à requêtes dans un système à base d'intergiciel de communication. (page 20)

GIOP Le *Generalized Inter-ORB Protocol* décrit la manière dont les éléments d'une invocation de méthode doivent être encapsulés dans la couche de transport. (page 21)

IIOP Le *Internet Inter-ORB Protocol* fait le lien entre GIOP et TCP. (page 21)

IMR Le *Implementation Repository* est un dæmon qui associe des noms d'interface aux paramètres nécessaires au démarrage d'un processus qui incarne cette interface (généralement une image exécutable et des paramètres de ligne de commande). (page 23)

Intergiciel Un intergiciel, ou *middleware* en anglais, est une couche de logiciel qui se situe entre les applications et le système d'exploitation. (page 5)

IOGR Un *Interoperable Object Group Reference* est une référence qui permet d'invoquer des méthodes sur un groupe d'objets. Cette extension de la notion de référence objet provient de la norme FT-CORBA. (page 120)

IOR Une *Interoperable Object Reference* est une référence d'un objet CORBA, une structure opaque pour le niveau applicatif qui désigne un objet. (page 13)

J2EE La plate-forme *Java 2 Enterprise Edition* spécifie un ensemble d'APIs pour la programmation en Java orientée serveur. (page 109)

LocateForward Type de message GIOP utilisé pour rediriger les requêtes CORBA. Il indique que l'objet désigné par une requête se trouve à une autre adresse réseau. (page 91)

Modèle de fautes Une abstraction des différentes classes de fautes qui peuvent affecter un système. (page 43)

OMG Le consortium *OMG* (*Object Management Group*) regroupe plus de 500 industriels dans le secteur informatique. Ce consortium est chargé de la définition des normes CORBA. (page 5)

OMG IDL Langage de description d'interfaces défini par l'OMG, et utilisé par la norme CORBA. (page 13)

one-way Une requête CORBA *one-way*, ou sans retour, est une requête sans valeur de retour. Contrairement aux requêtes ordinaires, l'invocation d'une méthode one-way n'est pas bloquante. (page 14)

Oracle Dans le contexte du test, un *oracle* est un composant du banc de test qui est responsable de décider si la valeur renvoyée par un composant testé est valide ou non. (page 36)

POA Le *POA* ou *adaptateur d'objets* est un élément d'un courtier CORBA qui reçoit les requêtes et les envoie au servant approprié. (page 19)

Projection langage La projection langage, ou *language mapping*, définit les aspects de l'interaction entre le niveau applicatif et l'intergiciel qui sont spécifiques à un langage de programmation donné. (page 18)

Référentiel d'interfaces Service CORBA qui fournit des méta-informations sur les interfaces OMG IDL. Surtout utilisé pour le mécanisme DII. (page 21)

Robustesse Capacité d'un système à délivrer un service acceptable en présence d'entrées invalides et de conditions environnementales stressantes. (page 31)

RPC Un *Remote Procedure Call*, ou *RPC*, est un mécanisme d'appel de procédure à distance. (page 7)

RT-CORBA La norme *Real-Time* CORBA intègre des mécanismes nécessaires aux traitements temps-réel « mou » dans un contexte réparti à base d'intergiciel CORBA. (page 70)

Servant Un *servant* est une représentation concrète, dans un langage de programmation donné, d'un objet CORBA qui joue un rôle de serveur. (page 18)

Service context Partie d'un message GIOP qui permet de transporter une information relative au contexte dans lequel s'exécute la requête (par exemple, un identifiant de la transaction dans laquelle s'inscrit la requête). (page 76)

SNMP Le *Simple Network Monitoring Protocol* est utilisé pour le contrôle et la supervision d'équipements réseau. (page 34)

SOAP Un mécanisme d'appel de procédure à distance destiné aux applications faiblement couplées qui communiquent sur des réseaux à grande échelle. Il décrit un format d'emballage des paramètres basé sur XML, et permet d'utiliser plusieurs transports (HTTP, SMTP). (page 109)

Souche Une *souche* (ou *stub* en anglais) sert de mandataire local pour l'interface qu'elle représente. (page 19)

Squelette Un *squelette* (ou *skeleton* en anglais) sert d'intermédiaire entre l'adaptateur d'objets et le servant, pour une interface donnée. (page 19)

SUT Le *System Under Test*, c'est à dire la cible de l'expérience. (page 71)

Test de conformité Le *test de conformité* consiste à évaluer la correction fonctionnelle d'une implémentation d'un système vis-à-vis d'une spécification de référence. (page 34)

TLS Le *Transport Layer Security* est un protocole de l'IETF permettant l'authentification et la communication chiffrée sur Internet. Il est basé sur une norme plus ancienne appelée *SSL*. (page 114)

UserException Famille d'exceptions CORBA qui regroupe les exceptions qui sont levées au niveau applicatif. Par convention, le nom de ces exceptions est composé de caractères minuscules et majuscules. (page 73)

Vulnérabilité Une faute de conception, intentionnelle ou accidentelle, qui favorise la réalisation d'une menace ou la réussite d'une attaque. (page 33)