INJECTION DE FAUTES DANS LES SYSTEMES DISTRIBUES

William Hoarau

Université de Paris-Sud 11

THÈSE

pour l'obtention du grade de : **Docteur**
Spécialité : INFORMATIQUE

Injection de Fautes dans les Systèmes Distribués

William Hoarau

Thèse soutenue le 21 mars 2008 devant le jury composé de :

- Joffroy Beauquier, Professeur, Univ. Paris-Sud, examinateur
- Christian Perez, Chargé de Recherche INRIA (HDR), IRISA, rapporteur
- Franck Petit, Professeur, Univ. de Picardie, examinateur
- Pierre Sens, Professeur, Univ. Pierre et Marie Curie, rapporteur
- Sébastien Tixeuil, Professeur, Univ. Pierre et Marie Curie, directeur

Laboratoire de Recherche en Informatique
Bât 490 Université Paris-Sud - 91405 Orsay Cedex - France

Résumé

Dans un réseau constitué de plusieurs milliers d'ordinateurs, l'apparition de fautes est inévitable. Etre capable de tester le comportement d'un programme distribué dans un environnement où l'on peut contrôler les fautes (comme le crash d'un processus) est une fonctionnalité importante pour le déploiement de programmes fiables. Dans cette thèse, nous présentons FAIL (pour FAult Injection Language), un langage qui permet d'élaborer des scénarios de fautes complexes relativement facilement, tout en déchargeant l'utilisateur de l'écriture de code de bas niveau. En outre, il est possible de construire des scénarios de fautes probabilistes (pour des tests quantitatifs) ou déterministes et reproductibles (pour étudier le comportement de l'application dans des cas particuliers). Ensuite, nous présentons FCI (FAIL Cluster Implementation), notre injecteur de fautes, qui consiste en un compilateur, une bibliothèque d'exécution et une plate-forme pour l'injection de fautes dans des applications distribuées. FCI est capable de s'interfacer avec de nombreux langages de programmation sans nécessiter la modification de leur code source. Nous présentons également les tests que nous avons conduits sur différentes applications distribuées.

Mots-clefs : Systèmes Distribués, Tolérence aux Fautes, Injection de Fautes.

Abstract

In a network consisting of several thousands computers, the occurrence of faults is unavoidable. Being able to test the behavior of a distributed program in an environment where we can control the faults (such as the crash of a process) is an important feature that matters in the deployment of reliable programs. In this thesis, we present FAIL (for FAult Injection Language), a language that permits to elaborate complex fault scenarios in a simple way, while relieving the user from writing low level code. Besides, it is possible to construct probabilistic scenarios (for average quantitative tests) or deterministic and reproducible scenarios (for studying the application's behavior in particular cases). Then we present FCI, the FAIL Cluster Implementation, that consists of a compiler, a runtime library and a middleware platform for software fault injection in distributed applications. FCI is able to interface with numerous programming languages without requiring the modification of their source code. We also present the tests that we conducted on different distributed application.

Mots-clefs : Distributed Systems, Fault Tolerence, Fault Injection.

Remerciements

Je tiens tout d'abord à remercier les membres de mon jury : Messieurs Joffroy Beauquier et Franck Petit, qui ont accepté avec plaisir le rôle d'examinateur, et Messieurs Christian Perez et Pierre Sens, qui en rapportant sur ce manuscrit, ont permis de lui donner une forme beaucoup plus agréable. Qu'ils trouvent ici l'expression de ma reconnaissance.

Je remercie Monsieur Sébastien Tixeuil pour son encadrement et sa bonne humeur. Il a su aiguiller mes réflexions et éclaircir mes idées sur les différents problèmes rencontrés tout au long de ma thèse.

Je remercie toute l'équipe "Parall" pour leur accueil chaleureux et l'aide qu'ils ont pu m'apporter. Je remercie également les membres de l'équipe "Architectures Parallèles" pour leur aide et les moments de rigolade qu'on a pu passer ensemble.

Je remercie en particulier Benjamin Quétier, Ala Rezmerita et Vincent Néri pour les divertissements qu'on a eus ensemble et pour toutes les blagues qu'ils ont pu faire au LRI (surtout quand ils ont attaché mon clavier avec un cable usb et qu'ils ont tiré dessus alors que je n'avais rien vu et que je surfais sur internet... ça va me marquer à vie ! excellent ! :-D).

Je remercie Monsieur Luis Silva et son équipe pour leur accueil au Portugal.

Je tiens à porter un hommage au système d'enseignement universitaire qui permet à ceux qui le désirent de s'instruire dans de très bonnes conditions et à l'informatique en général qui a constitué la majeure partie de ma vie ces dernières années (et, a priori, ce n'est pas fini... enfin, je l'espère).

Je remercie tous mes amis qui ont su m'apporter les indispensables moments de détente et avec qui j'ai pu m'amuser tout au long de ma scolarité.

Je remercie Monsieur Yves Boudier et Madame Francine Gérard pour m'avoir accueilli à mon arrivée en France métropolitaine et soutenu le long de ces cinq années passées loin de ma famille.

Je remercie Monsieur et Madame Ayé qui ont été comme ma deuxième famille pendant mon enfance et Yoan Ayé (leur fils) avec qui, accompagné de mon frère David, j'ai fait les quatre cents coups.

Je remercie ma famille qui a su me soutenir tout le long de mes études.

Et je remercie ma fiancée qui a toujours été présente, même dans les moments les plus difficiles.

Table des matières

Chapitre 1

Introduction

Un système centralisé est un système où toutes les ressources sont localisées sur la même machine et accessible par un programme. Dans ce type de système, un logiciel est exécuté sur une seule machine et accède localement aux ressources nécessaires (données, code, périphériques, mémoire, etc). Par opposition, un système distribué est constitué d'un ensemble d'ordinateurs connectés en réseau et communiquant via ce réseau apparaissant, du point de vue de l'utilisateur, comme une unique entité. Un système distribué permet ainsi de profiter d'une énorme quantité de ressources. C'est pourquoi ce type de systèmes connaît de nos jours un essor considérable.

On peut considérer deux visions d'un système distribué. La première est une vision matérielle :

- machine multi-processeurs avec mémoire partagée,
- cluster d'ordinateurs dédiés au calcul/traitement massif parallèle,
- ordinateurs standards connectés en réseau.

La deuxième est une vision logicielle : système logiciel composé de plusieurs entités s'exécutant indépendamment et en parallèle sur un ensemble d'ordinateurs connectés en réseau.

Il existe de nombreux types de systèmes distribués utilisant différents types de ressources. On peut noter par exemple :

Les serveurs de fichiers : ils permettent, notamment, à l'utilisateur d'accéder à ses fichiers quelle que soit la machine utilisée. Les fichiers se trouvent physiquement sur le serveur, mais l'accès à ces fichiers peut se faire à partir de n'importe quelle machine cliente en faisant « croire » que ces fichiers sont stockés localement (par exemple, des répertoires distants montés via le protocole NFS). Les serveurs de fichiers ont donc plusieurs in-

térêts tels que l'accès à des fichiers à partir de n'importe quelle machine, les systèmes de sauvegarde associés et la transparence pour l'utilisateur.

les serveurs web : ce sont des serveurs auxquels se connecte un nombre quelconque de navigateurs web (clients) permettant ainsi un accès à distance de l'information. L'accès est simple ; le serveur renvoie une page HTML statique qu'il stocke localement. Le traitement peut être plus complexe ; le serveur peut, par exemple, interroger une base de données pour générer dynamiquement le contenu de la page, mais cela reste transparent pour l'utilisateur : les informations s'affichent dans son navigateur quelle que soit la façon dont le serveur les génère.

Les calculs scientifiques : ils sont généralement utilisés sur deux types d'architectures matérielles :

1. *Les clusters*, qui sont des ensembles de machines, généralement identiques, reliées entre elles par un réseau dédié et très rapide.
2. *Les grilles*, qui sont des ensembles de clusters, généralement hétérogènes, connectés entre eux par un réseau local ou bien encore par Internet.

Le principe général des calculs scientifiques est l'utilisation d'un ou de plusieurs serveurs qui distribuent des calculs à effectuer aux machines clientes. Un client exécute son calcul puis renvoie le résultat au serveur. L'intérêt est l'utilisation d'un maximum de ressources de calcul. Notamment, MPI (Message Passing Interface) s'est imposé comme un standard pour la programmation d'applications parallèles utilisant le passage de message comme paradigme de communication grâce à sa disponibilité sur de nombreuses machines parallèles allant du cluster à faible coût aux clusters de multiprocesseurs vectoriels. Il permet au même code d'être exécuté sur différentes architectures et sur des machines de générations différentes, assurant ainsi une grande durée de vie au code. De plus, MPI concorde bien au style populaire de programmation haute performance qu'est le passage de message. Même si de nombreuses applications suivent le paradigme de programmation SPMD (Single Process, Multiple Data), MPI est souvent utilisé pour des exécution de type *Master-Worker*, où les nœuds MPI jouent des rôles différents. Pour ces raisons, MPI est le plus utilisé des environnements de programmation pour applications hautes performances.

Les systèmes pair-à-pair : ce sont des ensembles constitués d'utilisateurs et de machines, du protocole qui leur permet de communiquer et du fonctionnement du protocole entre les machines ; les participants sont indépendants et mettent en commun des

ressources afin d'aboutir à un résultat. Le terme nœud permet de désigner le logiciel présent sur une machine, donc souvent un utilisateur. Le terme de lien désigne une connexion (souvent TCP) entre deux nœuds. Le terme d'objet désigne ce qui est partagé dans un système pair-à-pair :

- puissance de calcul (XtremWeb, SETI@home, Decrypthon, Boinc, etc)pour exécuter de longs et/ou nombreux calculs ;
- fichiers (Gnutella et tant d'autres) ;
- services (DNS) ;
- etc.

Dans un système pair-à-pair, les nœuds ne jouent pas exclusivement les rôles de client ou de serveur mais peuvent assurer parallèlement les deux fonctions. Ils sont en effet simultanément clients et serveurs des autres nœuds du réseau, contrairement aux systèmes de type client/serveur. Ils jouent aussi le rôle de routeur, en passant les messages de recherche voire les données vers leur(s) destinataire(s). Ces systèmes n'ont pas d'administration centrale, tous ont les mêmes droits et obligations. Les systèmes pair à pair sont divisés en deux grandes familles d'architecture :

- l'architecture centralisée, où un client (un logiciel utilisé par les membres) se connecte à un serveur qui gère les partages, la recherche, l'insertion d'informations, bien que celles-ci transitent directement d'un utilisateur à l'autre. Ce type d'architecture rend le système complètement dépendant du serveur central. Ainsi, s'il est défaillant, tout le réseau s'effondre.
- l'architecture décentralisée permet de résister à la défaillance d'un serveur, puisque le logiciel client ne se connecte pas à un unique serveur mais à plusieurs (le rôle de serveur pour un participant est effectué par d'autre participants). Le système est ainsi plus robuste, mais la recherche d'informations est plus difficile et, dans des systèmes comme Gnutella, la recherche nécessite un nombre de messages élevé, proportionnel au nombre d'utilisateurs du réseau et à la profondeur de recherche. Cependant, des protocoles optimisés ont pu être mis en place, basés sur les tables de hachage distribuées, permettant de réaliser des recherches en un nombre de messages croissant de façon logarithmique avec le nombre d'utilisateurs du réseau, comme CAN, Chord, Freenet, GNUnet, Tapestry, Pastry et Symphony.

Le principe des tables de hachage distribuées (DHT) est d'associer des clés à des objets dispersés sur le réseau. L'intérêt de cette méthode est la faible contrainte imposée aux objets, il suffit juste de pouvoir leur associer une valeur numérique unique. Ces clés et objets pouvant être disséminés sur tout le réseau. Le coeur du problème de ce modèle réside

dans la fonction de localisation des objets sur le réseau à partir de leur clé. Concrètement, une DHT implémente juste une opération H(x) qui retourne l'identité du noeud qui détient l'information (par exemple son adresse IP). La personne qui voudra partager une ressource sur le réseau devra convertir le nom de cette ressource en une clé (avec des fonctions de hachage telles que SHA-1). Ce qui entraîne un certain nombre de problèmes liés aux tables de hachage :

- l'association des clés aux noeuds. Comment répartir les clés sur les noeuds ? Dans l'absolu, il faudrait que chaque noeud ait la même charge de travail que les autres noeuds.
- faire suivre une recherche jusqu'au noeud concerné. Chaque noeud doit faire suivre la requête de recherche au noeud le plus proche. Pour ce faire, chaque noeud conserve une table de routage avec un nombre limité de noeuds.
- la construction de la table de routage. Pour faire suivre les messages de recherche, chaque noeud conserve des informations sur les noeuds voisins.

Les grilles d'ordinateurs de bureau : les grilles d'ordinateurs de bureau sont une sous-catégorie des systèmes pair-à-pair (introduits précédemment). Elles utilisent les cycles oisifs des ordinateurs de bureau pour effectuer du calcul à grande échelle et du stockage de données pour un coût relativement faible. Ces types de plate-forme informatique sont les plus grands systèmes informatiques distribués au monde. Un des projets les plus populaires, SETI@home [79], est composé de centaines de milliers d'ordinateurs de bureau distribués à travers le monde. De nombreux autres projets utilisent cette puissance de calcul pour exécuter des applications appartenant à des domaines scientifiques très vastes, tel que FOLDING@home [72], EINSTEIN@home [29], BOINC [6], et XtremWeb [15, 31]. XtremWeb est une plate-forme générale qui peut être utilisée pour l'exécution de calculs distribués. Une liste de tâches (ou jobs) est décrite par l'utilisateur et distribuée sur les différents noeuds disponibles du système. XtremWeb est basé sur l'utilisation de machines volontaires, *i.e* il permet à de grandes écoles, des universités ou des entreprises de configurer et d'exécuter un calcul global ou un système distribué pair-à-pair pour une application dédiée ou une panoplie d'applications diverses. La version officielle de l'application XtremWeb est écrite en Java, mais une version C++ est également disponible. Les participants de XtremWeb sont divisés suivant leur fonction : le `dispatcher` distribue les tâches qui seront exécutées par les `workers`, chaque `worker` demande et exécute les tâches et les `clients` proposent des tâches à exécuter. Comme d'autres plate-formes, XtremWeb utilise *(i)* des ressources distantes (PCs, stations de travail, serveurs) connectées à internet, ou *(ii)* un groupe de ressources (PCs, stations de travail, serveurs) d'un

réseau local.

Cependant l'utilisation de tels systèmes entraîne un problème majeur car dans un réseau comprenant plusieurs milliers de machines, l'apparition de défaillances est inévitable. Aucune machine, y compris en électronique et en informatique, n'est fiable à 100%, ni inusable. C'est pourquoi un système distribué, qui n'est autre qu'un ensemble de ces machines, est particulièrement soumis à l'apparition de fautes car une faute peut apparaître sur chacun de ses composants (*i.e* chaque machine). Ainsi, plus un système a une grande échelle (*i.e.* un nombre de machines participantes élevé), plus la probabilité d'apparition de défaillances est grande. Le degré de gravité des défaillances est classé comme suit :

1. ***panne franche (« fail stop »)*** **:** soit le système fonctionne normalement (les résultats sont corrects), soit il ne fait rien. Il s'agit du type de panne le plus simple ;
2. ***panne par omission ou panne transitoire*** **:** des messages sont perdus en entrée ou en sortie ou les deux. Elle est considérée comme une panne temporelle de durée infinie ;
3. ***panne temporelle*** **:** le temps de réponse du système ne respecte pas les exigences des spécifications ;
4. ***panne byzantine*** **:** le système donne des résultats aléatoires.

Ces systèmes doivent donc tolérer l'apparition de pannes car celles-ci sont intrinsèques au fonctionnement du système.

Fautes, erreurs, défaillances : une défaillance (ou panne) survient quand le comportement d'un système ne répond pas à sa spécification. Une erreur est une partie de l'état d'un système susceptible de causer une défaillance. Une faute est une cause (événement, action, circonstance) pouvant provoquer une erreur.

Une erreur est susceptible de provoquer une défaillance, mais ne la provoque pas nécessairement (ou pas immédiatement) parce qu'il y a un mécanisme interne suffisant pour que le système continue de fournir le service ou parce que la partie erronée de l'état n'est pas utilisée pour telle ou telle fonction. Une erreur est latente tant qu'elle n'a pas provoqué de défaillance. Le temps entre l'apparition de l'état d'erreur et la défaillance est le délai de latence. Plus le délai de latence est long, plus la recherche des causes d'une défaillance est difficile.

Propagation entre composants (ou sous-systèmes) : un composant A utilise un composant B si le bon fonctionnement de A dépend de celui de B. Pour A, la défaillance de B

(service incorrect) constitue une faute, qui peut à son tour provoquer une erreur interne à A, puis une défaillance de A. On obtient ainsi le cycle suivant :
faute -> erreur -> défaillance -> faute -> erreur -> ...

La tolérance aux fautes : quelles que soient les précautions prises, l'occurrence de fautes est inévitable (erreur humaine, malveillance, vieillissement du matériel, catastrophe naturelle, etc.). Cela ne veut pas dire qu'il ne faut pas essayer de prévenir ou d'éliminer les fautes, mais les mesures prises peuvent seulement réduire la probabilité de leur occurrence. Il faut donc assurer la fourniture du service malgré l 'occurrence de fautes. Le concept de tolérance aux fautes se réfère à une méthode de conception d'un système de telle façon qu'il puisse continuer à fonctionner, potentiellement de manière réduite, au lieu de tomber complètement en panne dès que l'un de ses composants est victime d'une faute. Plusieurs mécanismes de tolérance aux fautes ont donc été définis :

- **la reprise :** la reprise est basée sur un retour en arrière vers un état antérieur dont on sait qu'il est correct. Cette technique nécessite la sauvegarde de l'état, mais elle est générale. La reprise implique plusieurs problèmes : 1) les sauvegardes et restaurations doivent être atomiques (i.e aucunes interférences durant ces étapes), 2) l'état des points de reprise doit lui-même être protégé contre les fautes, 3) dans un système répartie, les points de reprise doivent être créés pour chaque processus et l'ensemble de ces points de reprise doit constituer un état global cohérent du système. Deux mécanismes pour la constitution de l'état de reprise cohérent ont été définis : la constitution « A priori » (approche planifiée) et la constitution « A posteriori » (approche non planifiée). La constitution « A priori » nécessite que les sauvegardes des différents processus soient coordonnées pour constituer un état global cohérent, qu'une synchronisation explicite soit donc assurée entre les processus (coût lors de la sauvegarde), mais qu'un seul point de reprise par processus soit conservé (le dernier). Pour la constitution « A posteriori », les sauvegardes des processus sont indépendantes, la reconstitution d 'un état cohérent est faite lors de la reprise (coût lors de la restauration), plusieurs points de reprise par processus sont nécessaires et les traces de communications doivent être conservées.
- **la poursuite :** la poursuite est basée sur une tentative de reconstitution d'un état correct, sans retour arrière. La reconstitution est souvent seulement partielle, le service peut donc être dégradé. Cette technique est spécifique et dépend donc de l'application.
- **la duplication passive :** la duplication est un mécanisme où la tolérance aux fautes est obtenue par redondance, par exemple, l'utilisation de N serveurs pour

résister à la panne franche de $N-1$ serveurs. Il existe deux types de duplication : la duplication active et la duplication passive. Le principe de la duplication passive, pour la tolérance aux fautes d'un serveur, est qu'un serveur primaire joue un rôle privilégié. Les clients connaissent le serveur primaire et lorsqu'une panne du primaire survient, un serveur de secours devient le serveur primaire. La panne du primaire est visible par les clients, implique un changement de serveur, et, dans ce cas, il y a des pertes possibles de requêtes.

- **la duplication active :** elle a pour principe l'utilisation de serveurs déterministes (*i.e* les réponses sont déterminées par la séquence des requêtes et le comportement du serveur est indépendant du temps physique). Lorsque des requêtes sont traitées par un serveur, elles doiventt être traitée dans le même ordre par tous les serveurs en état de marche. Du point de vue du client, l'ensemble des N serveurs se comporte comme un serveur « abstrait » unique et les pannes lui sont masquées. Avec une duplication active, tous les serveurs sont mobilisés pour la même tâche, mais la reprise suite à une faute est immédiate alors que pour une duplication passive, les serveurs de secours peuvent exécuter une tâche de fond non prioritaire mais la reprise est non immédiate.

Les applications pair à pair, introduitent précédemment, impliquent de grandes quantités d'utilisateurs. Dans ce type d'application, l'apparition et la disparition des PCs participants sont très imprévisibles, extrêmement fréquentes et s'effectuent inévitablement pendant un calcul. Pouvoir tester le comportement des programmes répartis dans un environnement où l'on peut contrôler l'apparition de défaillances (comme le crash d'un processus) est un élément important dans le déploiement de programmes fiables. De plus, les mécanismes de tolérance aux pannes de ces programmes doivent être les plus performants possibles et il est donc nécessaire de disposer d'un environnement configurable pour les tester. D'autre part, les preuves formelles de telles applications sont très délicates à élaborer, étant donné leur caractère instable. Le fait que leur implantation a un impact considérable sur leurs performances pousse vers une expérimentation en conditions réelles. Un environnement de tests paramétrables et reproductibles peut alors s'avérer nécessaire pour repérer et corriger les erreurs d'implantation ou pour évaluer par l'expérience les performances de l'application dans un environnement sujet aux fautes. Dans les systèmes asynchrones, la détection des processus fautifs se fait généralement à l'aide de « timeout » (temps au bout duquel un processus est considéré comme ayant subit une défaillance), ce qui peut entraîner des incohérences étant donné l'impossibilité de différencier un processus lent d'un processus défaillant, d'où la nécessité de tester l'im-

pact du choix de la valeur des « timeout » pour vérifier les performances des applications face à des arrêts momentanés des processus paramétrés différemment. Un autre point important est l'essor actuel de la mobilité dans les réseaux. Il est fondamental de vérifier le comportement d'une application dans un environnement où des fautes intermittentes apparaissent de façon aléatoire et plus ou moins fréquemment. De plus, les fautes injectées aléatoirement doivent pouvoir se comporter suivant un scénario particulier. En effet, les fautes ont généralement tendance à se propager localement avec une forte probabilité (par exemple lorsqu'un ver réussit à contourner le pare-feu d'un réseau, toutes les machines de ce réseau seront très probablement touchées). Il faut donc un moyen pour tester la réaction d'applications distribuées face à cette propriété d'apparition de fautes. Notre but est de fournir un intergiciel pour l'injection de fautes distribuées et coordonnées afin de permettre l'étude d'applications distribuées dans un environnement plus proche de la réalité. Cet intergiciel permet de spécifier le contexte d'apparition de défaillances pour une application répartie s'exécutant dans un émulateur d'environnement parallèle, par exemple GRID eXplorer. Pour cela, nous avons défini un langage de description de fautes que nous avons nommé FAIL.

Le chapitre 2 présente un état de l'art sur l'injection de fautes dans les systèmes distribués. La première partie de cette thèse est consacrée à la présentation de notre intergiciel : le chapitre 3 présente les fondements théoriques qui ont permis sa conception, le chapitre 4 présente le langage FAIL qui permet la description de scénarios de fautes, et le chapitre 5 présente l'implantation de la plate-forme qui permet l'injection de fautes effectives dans un système distribué selon le scénario décrit en FAIL. La seconde partie de cette thèse est consacrée aux différentes expérimentations que nous avons effectuées sur différentes applications distribuées : le chapitre 6 présente et quantifie le surcoût induit par l'utilisation de notre outil, le chapitre 7 présente les expérimentations d'injection de fautes que nous avons effectuées sur XtremWeb, MPICH-V et FreePastry, le chapitre 8 présente les expérimentations de stress qui ont été faites sur OGSA-DAI et XtremWeb, et enfin, le chapitre 9 présente les expérimentations que nous avons effectuées sur XtremWeb pour simuler des utilisateurs dans un réseau, *i.e.* l'arrivé et le départ de nœuds dans le système. Finalement, le chapitre 10 conclut cette thèse et présente les perspectives pour notre injecteur de fautes.

Chapitre 2

Etat de l'Art

Le test est devenu un passage quasiment obligatoire dans le processus de développement d'une application. En effet, même après une étude approfondie en amont et une implantation rigoureuse, des bogues d'implantation peuvent subsister. La manière la plus cohérente de découvrir et de corriger ces bogues est de tester si l'application répond bien aux spécifications en utilisant un jeu de test couvrant au maximum le code de l'application (i.e. impliquant l'exécution du maximum de lignes de code différentes de l'application testée) en utilisant des entrées adéquates pour l'application testée. Cependant, pour des applications tolérantes aux fautes, le code correspondant à l'implantation de la tolérance aux fautes ne dépend pas de leurs entrées mais de l'apparition de fautes lors de leur exécution. Une manière d'inclure le test du code correspondant à la tolérance aux fautes lors d'un test est donc de générer ces fautes. L'injection de fautes est alors nécessaire pour obtenir une couverture maximum lors de la phase de test d'une application. Nous avons fait un état de l'art sur l'injection de fautes dans [40] et nous allons dans ce chapitre présenter cet état de l'art. Ainsi, quand on désire injecter des fautes, un premier problème se pose : quand déclencher les fautes ? Différents types de déclenchement ont ainsi été définis :

- Déclenchement par rapport au temps (*Time-based trigger*) : c'est le type de déclenchement le plus simple. Les fautes sont injectées à un temps donné, généralement dépendant d'une variable aléatoire.
- Déclenchement par rapport à un événement (*Event-based trigger*) : le déclenchement de l'injection de fautes se fait lorsque l'application testée exécute des événements particuliers.
- Déclenchement par rapport à la charge de travail (*Stress-based trigger*) : l'injection de fautes peut être déclenchée lorsque la charge de travail du système participant

au test dépasse un certain seuil.

Ensuite il faut adapter l'injection de fautes au type du système visé. Pour l'injection de fautes, on peut distinguer deux grandes catégories de systèmes : les systèmes centralisés et les systèmes distribués.

2.1 Injection de fautes dans les systèmes centralisés

Il existe déjà de nombreux outils pour l'injection de fautes, les mesures de performance et de résistance aux défaillances. On remarque ainsi deux grandes classes de méthodes pour injecter des fautes, la méthode « hardware » et la méthode « software ». Dans les méthodes dites « hardware », on peut distinguer l'injection de fautes directe dans les broches du processeur (*pin-level fault injection*), utilisée par exemple dans RIFLE [56] et la radiation du matériel par des ions (*heavy-ion radiation*). Cependant, ces méthodes sont devenues trop difficiles à utiliser et à mettre en place, principalement à cause de la complexité des ordinateurs actuels et de l'investissement matériel trop important. De plus les injections de fautes matérielles sont instables à cause des interférences physiques et peuvent entraîner des dommages sur le matériel. Ensuite, il y a les méthodes dites « software » (SFI : Sofware Fault-Injection) [1, 73, 81] utilisées par exemple dans Xception [16, 55]. En effet, ces méthodes sont plus faciles à mettre en place financièrement. De plus, la reproductibilité des expériences est plus facilement envisageable et il n'y a pas de risque d'endommager le matériel servant au test. Nous allons donc écarter les méthodes « hardware » de notre étude et nous focaliser sur les méthodes « software ».

2.2 Injection de fautes dans les systèmes distribués

Il existe différents types d'approche pour l'évaluation et la vérification de systèmes distribués :

L'approche orientée simulation

CECIUM [5] est un environnement de test simulant une exécution distribuée et des injections de fautes. L'exécution distribuée est ainsi simulée sur une machine unique avec un espace d'adressage unique. Le code source de l'application testée n'est pas nécessaire. L'exécution étant simulée sur une machine unique, la reproductibilité des tests devient triviale. En effet, une injection de fautes déterministes peut se révéler nécessaire afin de

pouvoir relancer le même scénario de fautes plusieurs fois si nécessaire, ce qui est plus facilement réalisable dans ce type d'approche.

De plus, lors de tests sur un système réellement distribué, des fautes involontaires peuvent plus facilement apparaître spontanément, comme la perte de certains messages, faisant ainsi diverger l'injection de fautes du scénario prévu. Ensuite, les propriétés des protocoles distribués sont souvent dépendantes d'un état global et lors d'un scénario de fautes, on peut donc vouloir injecter des fautes par rapport à un état global du système. Déterminer le bon moment pour injecter les fautes peut donc devenir très difficile dans un système distribué.

Dans les méthodes de simulation, on peut aussi remarquer MEFISTO [49] (Multi-level Error/Fault Injection Simulation Tool). MEFISTO est un outil permettant d'appliquer l'injection de fautes le plus tôt possible dans le processus de conception de systèmes tolérants aux fautes. Il permet ainsi d'injecter des fautes dans des modèles de simulation VHDL.

Néanmoins, ces approches restent de la simulation et ne remplacent pas l'exécution des applications sur un système distribué afin d'obtenir des résultats les plus proches possibles d'un système distribué réel. Nous voulons émuler des applications sur un système distribué et les approches orientées simulation ne correspondent donc pas à nos besoins.

L'approche orientée émulation

Quelques outils existent déjà pour émuler des apparitions de fautes dans des systèmes physiquement répartis. L'un de ces outils, DOCTOR [36] (integrateD sOftware fault injeCTiOn enviRonment), permet l'injection de fautes dans les systèmes temps réel . Ce dernier est capable d'injecter différents types de fautes et de synthétiser du « workload » (c'est-à-dire de simuler de la charge de travail sur une machine). Il supporte trois types de fautes : fautes de processeurs, de mémoires et de communications. Les fautes injectées peuvent être permanentes, passagères ou intermittentes. DOCTOR a été implanté sur un système temps réel appelé HART. Pendant les évaluations, DOCTOR collecte des informations de performance et de fiabilité. De plus, une interface graphique intuitive est fournie. Cependant, les types des fautes gérées ne sont pas suffisants pour l'étude des systèmes distribués à grande échelle. En effet, les fautes les plus fréquentes sont les fautes de type « crash » (arrêt total du processus) ; on doit donc pouvoir vérifier si l'application est résistante à ce type de fautes. De plus, on ne peut spécifier que des scénarios de fautes probabilistes (qui sont facilement paramétrables grâce à l'interface graphique). Nous allons donc nous intéresser plus particulièrement aux outils réellement

dédiés à l'étude de systèmes distribués, ORCHESTRA [24, 25, 26, 27, 28], NFTAPE [78] et LOKI [19, 20, 22, 37, 63].

2.2.1 ORCHESTRA

ORCHESTRA est un outil d'injection de fautes de type *software* qui permet de tester la fiabilité et la vivacité de protocoles distribués. Pour cela, une couche d'injection de fautes est insérée au dessous de celle du protocole afin de filtrer et de manipuler les messages qui sont échangés entre les participants. Les scripts de test peuvent être probabilistes ou déterministes. En effet, les messages peuvent être retardés, perdus, réordonnés, dupliqués, modifiés ou de nouveaux messages peuvent être introduits spontanément dans le système testé pour l'amener à un état particulier.

Architecture de ORCHESTRA

La couche d'injection de fautes est insérée entre la couche du protocole testé et les autres couches inférieures. Elle est composée d'un filtre pour les messages sortants et d'un filtre pour les messages entrants. Les filtres sont des scripts qui sont appliqués sur les messages entrants et sortants du protocole testé. Ils effectuent trois types d'opérations sur les messages :

- filtrage des messages (interception et analyse des messages),
- manipulation des messages (perte, retard, réordonnancement, duplication ou modification),
- injection de messages (introduction de nouveaux messages).

Les scripts de réception et d'envoi déterminent les opérations à effectuer sur les messages selon leur type, leur contenu, l'historique des messages ou suivant d'autres informations collectées lors du test (par exemple le temps local, le nombre de messages reçus, etc). Le principal composant d'ORCHESTRA est le *PFICore* (Protocol Fault Injection Core) qui sert à injecter les fautes. Il est utilisé pour construire la couche d'injection de fautes. Il est indépendant du format des messages car il voit les messages comme des objets abstraits. L'interprétation des messages se fait par la couche d'injection de fautes qui doit être implantée par l'utilisateur.

Définition des scénarios de fautes dans ORCHESTRA

Les scripts d'injection de fautes sont écrit en Tcl. Ce sont des machines à états qui sont appliquées sur chaque message (le scripte d'envoi pour les messages sortants et le script de réception pour les messages entrants) grâce à un interpréteur du langage Tcl. Lors de la réception (ou l'envoi) d'un message, la machine à états est généralement dans un état initial, mais elle peut se trouver dans n'importe quel autre état (cela dépend du script de l'utilisateur). En effet, le message précédent peut avoir laissé la machine à états dans un état quelconque. Ensuite la machine à états transite vers d'autres états suivant des informations telles que le type du message, le nombre de messages reçus, le temps local, etc, et applique des modifications sur le message suivant le script qui la définit. Le message peut ainsi être perdu (il ne sera simplement pas retransmis à la couche à laquelle il était destiné), retardé, transmis après le prochain message (réordonancement), etc. Le message sera alors transmis à la couche suivante de la pile du protocole (s'il ne devait pas être perdu ou réordonnancé) et l'interpréteur passera au message suivant de sa queue de réception. Un exemple typique de script Tcl pour ORCHESTRA est présenté dans la figure 2.1.

```
if { [catch {set state}] == 1} {
    set state 0
}
while {1} {
    if { $state == 0} {
        msg_delay 5.0 1
        return
    } elseif {$state == 1} {
        set state 0
        return
    }
}
```

Fig. 2.1 – Exemple de script Tcl pour ORCHESTRA

Dans cet exemple, les trois premières lignes du script placent la machine à états dans l'état 0 lors de la première réception d'un message. Les autres lignes définissent la machine à états. Lors de la réception des messages, ils seront, soit retardés de 5 secondes

si la machine à états se trouve dans l'état 0, soit restitués « normalement » (*i.e.* sans délai) si la machine à états se trouve dans l'état 1. En effet, la ligne « msg_delay 5.0 1 » du script permet d'appliquer un délai de 5 seconde sur le message et fait transiter la machine à états vers l'état 1. La ligne « set state 0 » fait transiter la machine à états vers l'état 0. Une interface graphique permettant de générer ces scripts à partir d'une représentation graphique de la machine à états est fournie avec ORCHESTRA.

Conclusion sur ORCHESTRA

ORCHESTRA est un injecteur de fautes de type « Message-level fault injector » car la couche d'injection de fautes est insérée entre deux niveaux dans la pile du protocole, ce qui permet de ne pas avoir à modifier le code source du protocole pour le déclenchement et l'injection de fautes. De plus, le choix du langage Tcl pour l'écriture des scripts permet aux utilisateurs de définir des extensions (généralement en C), ce qui permet l'injection de fautes fondées sur des mécanismes complexes. Cependant, l'utilisateur doit lui même implanter sa couche d'injection de fautes pour le protocole qu'il utilise, ce qui demande un travail de développement important. De plus, il n'y a pas de communication entre les différentes machines à états, ce qui limite énormément la description des scénarios de fautes. Ensuite, comme l'injection de fautes se fait par rapport aux messages circulant, l'utilisation d'ORCHESTRA nécessite une importante connaissance du type et du format des messages. Finalement, ORCHESTRA est bien adapté pour l'étude de protocole, mais reste trop complexe d'utilisation et pas assez expressif pour une utilisation dans le cadre d'applications distribuées générales.

2.2.2 NFTAPE

Le projet NFTAPE est né de la constatation suivante : bien qu'il existe de nombreux outils d'injection de fautes, ceux-ci sont adaptés pour injecter un unique modèle de fautes dans un unique système et souffrent ainsi de deux limitations majeures pour leur utilisation dans des systèmes distribués :

- aucun outil n'est suffisant pour injecter tous les modèles de fautes,
- il est difficile de porter ces outils sur différents systèmes.

Architecture de NFTAPE

Tout d'abord, NFTAPE est composé d'un injecteur de fautes, d'un système de déclenchement des injections, d'un synthétiseur de charge de travail, d'un mécanisme de

détection des erreurs et d'un mécanisme de « logging » des résultats. Ensuite, ces composants sont séparés, ce qui permet à NFTAPE d'être modulaire. En effet, les utilisateurs peuvent créer leur propre composant d'injection de fautes et de déclenchement de fautes grâce à des interfaces. NFTAPE introduit ainsi la notion d'injecteur de fautes *léger* (LWFI : Lightweight Fault Injectors). Les LWFIs sont plus simples que les traditionnels injecteurs de fautes car ils n'incluent pas le système de déclenchement des injections, le mécanisme de « logging » et le support de la communication, ce qui permet à NFTAPE d'injecter n'importe quel modèle de fautes, selon n'importe quelle méthode. Finalement, NFTAPE définit aussi des interfaces pour tous ses composants afin de faciliter la portabilité vers différents systèmes. Dans NFTAPE, l'exécution du scénario de test est centralisée. Une machine ne participant généralement pas au test est désignée comme *Control Host* et exécute un script Jython (Jython est un sous-ensemble du langage Python) définissant le scénario de fautes. Toutes les machines participant au test sont munies d'un *Process Manager* qui communique avec le *Control Host*. Le *Control Host* envoie des lignes de commande aux différents noeuds participant au test suivant le scénario de fautes. Lors de la réception d'une commande, le *Process Manager* l'exécute et renvoie un message au *Control Host* à la fin de son exécution ou lors d'un crash. Toutes les décisions sont faites par le *Control Host*, ce qui implique que chaque déclencheur de fautes de chaque noeud doit communiquer avec lui. Ensuite, suivant le scénario de fautes, le *Control Host* envoie un message d'injection de fautes au *Process Manager* adéquat qui peut alors injecter la faute grâce à l'injecteur de fautes qui lui est attaché.

Définition des scénarios de fautes dans NFTAPE

La définition d'un scénario dans NFTAPE se fait en écrivant le script pour le *Control Host*. Ensuite, il faut fournir un fichier contenant toutes les valeurs pour tous les paramètres nécessaires à la réalisation de l'expérience d'injection de fautes et implanter les différents composants de NFTAPE nécessaires au déroulement du test ou utiliser des composants déjà implantés (s'ils correspondent aux besoins de l'utilisateur).

Conclusion sur NFTAPE

Bien que NFTAPE soit modulaire et très portable, le choix d'une solution de décision complètement centralisée le rend très intrusif et peu approprié pour le passage à l'échelle, donc inadapté à l'émulation de grille distribuée à grande échelle. De plus, sa mise en place peut demander un travail d'implantation très important (du fait que l'utilisateur peut avoir à implanter chaque composant). Finalement, l'écriture d'un scénario devient

rapidement complexe du fait de la nature centralisée des décisions lors de tests impliquant de nombreux noeuds.

2.2.3 LOKI

LOKI est un injecteur de fautes basé sur une vue partielle de l'état global du système distribué. Les fautes sont injectées par rapport à un état global du système. Une analyse posthume est exécutée à la fin des tests pour calculer un temps global à partir des différentes vues partielles et détermine ainsi si les fautes ont été injectées selon le scénario prévu. Bien qu'injecter des fautes par rapport à un état global d'un système distribué soit très difficile, voire impossible, sans un impact important au niveau du temps d'exécution, cette technique permet de vérifier la validité des fautes injectées et de limiter l'impact de l'injecteur de fautes.

Architecture de LOKI

Dans LOKI, chaque processus du système distribué est attaché à la bibliothèque d'exécution Loki pour former un nœud.

La bibliothèque d'exécution Loki est le code qui gère :

- la maintenance de la vue partielle de l'état global,
- l'injection de fautes quand le système arrive dans l'état désiré,
- la collecte d'informations sur les changements d'état et les injections de fautes.

Chaque nœud est constitué :

- d'une machine à états qui sert à conserver l'état local,
- d'une machine à état de transport qui sert à informer les machines à états distantes des changements d'états locaux et à recevoir les informations sur les changements d'état des machines à états distantes afin de maintenir la vue partielle de l'état global,
- d'un parser de fautes qui est le déclencheur de fautes,
- d'un enregistreur qui sert à enregistrer les dates de changement d'états locaux et les dates des injections de fautes.
- d'une sonde qui doit être fournie par l'utilisateur. Elle sert à instrumenter l'application pour avertir la bibliothèque d'exécution de l'apparition des événements et à injecter les fautes dans l'application.

LOKI propose trois modes d'exécution : centralisé, partiellement distribué et distribué.

Définition des scénarios de fautes dans LOKI

La définition des scénarios dans LOKI se fait par la spécification de la machine à états, la spécification des fautes à injecter et l'implantation de la sonde. Tout d'abord, l'utilisateur doit spécifier la machine à états en définissant la liste des différents états nécessaires, la liste des différents événements existants et la liste des transitions qui associent un état et un événement à un autre état. La syntaxe est présenté dans la figure 2.2.

```
global_state_list
LEAD
FOLLOW
end_global_state_list
event_list
LEADER
FOLLOWER
event
state LEAD notify
FOLLOWER FOLLOW
state FOLLOW notify
LEADER LEAD
```

Fig. 2.2 – Syntaxe des scénarios dans LOKI

Dans cet exemple, on définit une machine à états constituée de deux états (LEAD et FOLLOW) et de deux événements (LEADER et FOLLOWER). Lorsque la machine à états est dans l'état LEAD et que l'événement FOLLOWER se produit, elle se place alors dans l'état FOLLOW et lorsqu'elle est dans l'état FOLLOW et que l'événement LEADER se produit, elle se place dans l'état LEAD.

Remarque : une interface graphique est fournie pour assister cette phase de spécification. Ensuite, l'utilisateur doit spécifier quelles fautes injecter et quand les injecter. En fait, il doit définir des identificateurs de fautes et les associer à des états globaux du système testé. La syntaxe est la suivante :

```
bfault1 (black:LEAD \& green:FOLLOW) once in state machine green
```

Dans cet exemple, lorsque la machine à états *black* est dans l'état *LEAD* et la machine à états *green* est dans l'état *FOLLOW* alors la faute *bfault1* est injectée dans la machine

Critères	Orchestra	NFTAPE	LOKI	FAIL-FCI
Grande expressivité	non	oui	non	oui
Langage de haut niveau	non	non	oui	oui
Pas de modification du code source	oui	non	non	oui
Passage à l'échelle	oui	non	oui	oui
Scénario probabiliste	oui	oui	non	oui
Injection selon un état global	non	oui	oui	oui

FIG. 2.3 – Comparatif des systèmes d'injection de fautes

à états *green*. Le mot clef *once* sert à spécifier que la faute sera injectée uniquement la première fois où le système sera dans cet état global.

Finalement, l'utilisateur doit implanter la sonde et modifier l'application testée. En effet, des appels à des fonctions de la librarie LOKI doivent être insérés dans le code source de l'application testée pour communiquer à la machine à états locale les événements lui permettant ainsi de se positionner dans les états appropriés. De même, pour injecter les fautes l'utilisateur doit implanter la méthode *injectFault()* de la sonde qui prend en entrée le nom de la faute à injecter (un des noms de fautes déclarés dans la spécification des fautes) et retourne le temps où la faute a été injectée.

Conclusion sur LOKI

LOKI est un injecteur de fautes pour les systèmes distribués permettant d'injecter des fautes par rapport à un état global du système et de vérifier si celles-ci ont correctement été injectées (*i.e.* effectivement pendant l'état global défini). Cependant il nécessite la modification du code source de l'application à tester. Ensuite, l'utilisateur doit fournir un travail très important s'il veut définir un scénario de faute complexe. De plus il n'y a pas de gestion d'injection de fautes probabilistes.

2.2.4 Conclusion

Bien qu'il existe de nombreux outils d'injection de fautes, la plupart ne sont pas adaptés à l'étude d'applications distribuées. En ce qui concerne ceux réellement dédiés aux systèmes distribués, ils comportent chacun de nombreux inconvénients les rendant inadaptés à l'émulation de grille distribuée à grande échelle.

En effet, comme indiqué dans le tableau récapitulatif de la figure 2.3, pour certains il est difficile, voire impossible, de spécifier des scénarios complexes. Certains sont inadaptés

aux systèmes distribués à grande échelle du fait de leur architecture. Certains sont incapables de gérer l'aspect aléatoire de l'injection des fautes. Certains demandent un travail trop important pour l'utilisateur. Certains sont trop intrusifs ou demandent notamment la modification du code source de l'application testée. Notre but est de fournir un langage simple d'utilisation, appelé FAIL (pour FAult Injection Langage), qui ne nécessite pas la modification du code source de l'application à tester et permet la description de scénarios de fautes complexes et potentiellement probabilistes ou reproductibles. Notre langage sert d'interface à notre système d'injection de fautes dans GRID eXplorer et permet ainsi l'émulation d'applications dans un environnement avec injection de fautes.

Première partie

Un Intergiciel pour l'Injection de Fautes Distribuées et Coordonnées

Introduction

Notre but est de fournir un intergiciel pour l'injection de fautes distribuées et coordonnées afin de permettre l'étude d'applications distribuées dans un environnement plus proche de la réalité. Cet intergiciel permet de spécifier le contexte d'apparition de défaillances pour une application répartie s'exécutant dans un émulateur d'environnement parallèle. Un tel intergiciel doit être relativement simple à utiliser et interférer le moins possible sur l'exécution de l'application dont il permet l'étude. Nous avons donc choisi de définir un langage dédié à l'écriture de scénarios de fautes pour simplifier la définition d'une expérience d'injection de fautes. Un scénario écrit dans notre langage peut ainsi être compilé pour générer du code C++ et ensuite du code natif pour optimiser l'exécution de l'intergiciel et ainsi minimiser son intrusion. Nous avons également implanté un outil permettant d'automatiser la compilation, la configuration, l'exécution des expérimentations et la récupération des logs, ce qui permet ainsi de faciliter tout le processus de test.

Un tel intergiciel doit être autant que possible indépendant de l'application testée tout en permettant d'injecter des fautes selon l'état de celle-ci. Il doit permettre l'exécution de scénarios de fautes simples et probabilistes, mais également de scénarios plus complexes impliquant l'état de plusieurs composants de l'application.

Cette première partie est consacrée à la présentation de notre intergiciel : le chapitre 3 présente les fondements théoriques qui ont permis sa conception, le chapitre 4 présente le langage FAIL qui permet la description de scénarios de fautes, et le chapitre 5 présente l'implantation de la plateforme qui permet l'injection de fautes effectives dans un système distribué selon le scénario décrit en FAIL.

Chapitre 3

Fondements Théoriques

Dans cette partie, nous introduisons les automates communicants (en nous appuyant sur les travaux de Nathalie Bertrand [10]) et les automates temporisés (en nous appuyant sur les travaux de Patricia Bouyer [12]) qui serviront de support pour notre langage d'injection de fautes. Plusieurs entités s'envoient (et reçoivent) des messages et changent d'états en fonction de ces communications ou du temps. Plus particulièrement, nous considérons les systèmes communicants par canaux fiables (*i.e.* sans perte, corruption, insertion, etc, de messages) pour la construction de l'injecteur de fautes. A la fin d'un test, des mécanismes identiques à ceux utilisés dans LOKI pourraient permettre de détecter la présence de messages perdus ou reçus trop tardivement et le test pourra alors être marqué comme étant non valide.

3.1 Automates communicants

Les automates communicants sont des systèmes composés d'automates finis qui peuvent communiquer par l'intermédiaire de canaux non bornés. Les communications se font de façon asynchrone et l'ordre des messages est toujours préservé, *i.e* les canaux sont FIFO (*first in first out*). Ainsi, les messages sont reçus dans l'ordre où ils sont envoyés. Ce modèle permet de décrire naturellement des protocoles de communications entre plusieurs entités. Il est à la base de la sémantique de certains langage de spécification de protocoles tels que SDL [71] et Estelle [14] ainsi que de notre langage de spécification de scénarios de fautes que nous avons nommé FAIL (pour FAult Injection Language).

Le système représenté dans la figure 3.1 constitue un exemple d'automate communicant. Deux composant, le *serveur* et le *client* (représenté de part et d'autre des canaux), communiquent par l'échange de message au travers des canaux c_1 et c_2. Le serveur et

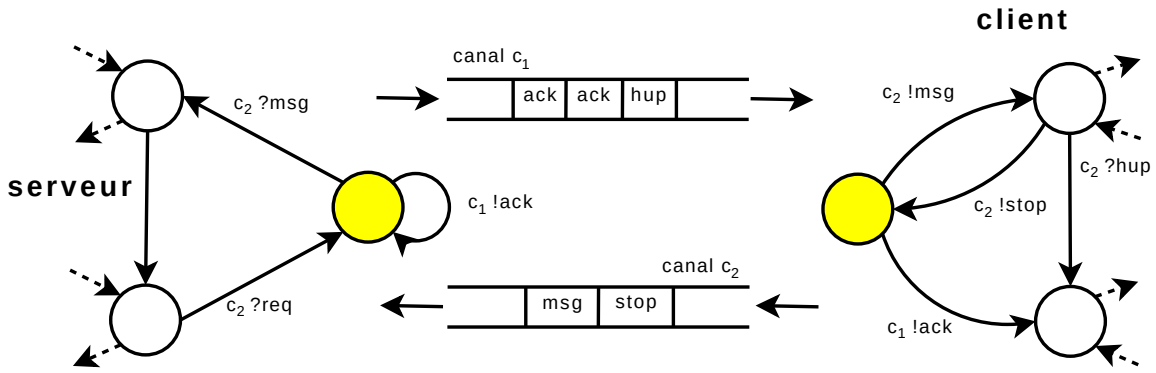

FIG. 3.1 – Un exemple d'automates communicants

le client peuvent changer d'état de contrôle en envoyant ou en recevant des messages. L'action consistant à écrire un message msg en queue du canal c_2 est notée $c_2!msg$ alors que celle correspondant à la lecture du message ack en tête du canal c_1 est notée $c_1?ack$.

La figure 3.2 représente un automate communicant composé d'un émetteur et d'un récepteur qui implante le protocole du Bit Alterné [9]. Ce protocole a pour but de pallier les pertes de messages en envoyant à plusieurs reprises les données et les accusés de réception, et utilise un bit (de valeur 0 ou 1) pour distinguer les messages.

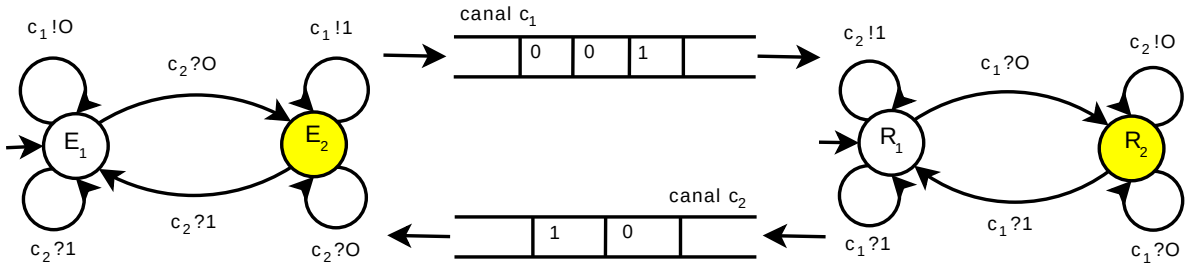

FIG. 3.2 – Une modélisation du protocole du Bit Alterné

Dans cette figure, les deux composants (émetteur et récepteur) sont représentés ainsi que les canaux c_1 et c_2 contenant des messages ; c_1 est utilisé pour les données et c_2 pour les accusés de réception.

Dans la définition formelle d'un automate communicant, on considère un unique automate fini dont le comportement est déterminé par les messages qu'il peut écrire et lire dans des canaux, qu'il utilise comme des files FIFO. Cette hypothèse n'est en aucun cas une restriction, puisqu'il suffit à partir de plusieurs automates de construire le produit pour obtenir un seul processus.

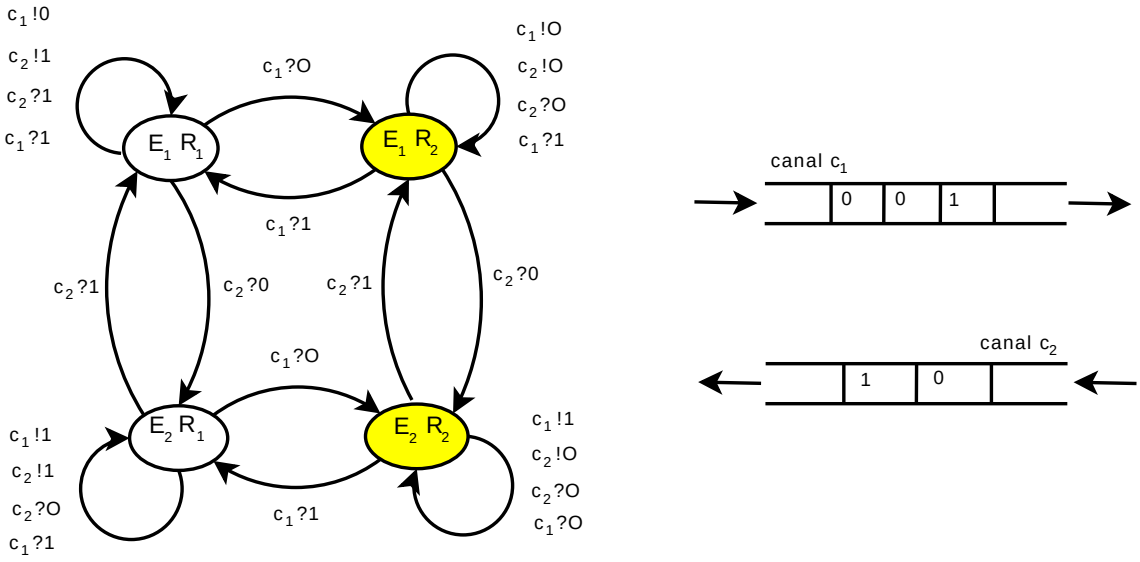

FIG. 3.3 – Le Bit Alterné sous forme d'automate communicant

Dans la figure 3.3, on réalise le produit des deux composants, pour obtenir un unique automate fini. De plus, pour simplifier le schéma, on a représenté plusieurs boucles en une seule en écrivant de multiples opérations sur une unique boucle.

Définition 1 (Automate communicant) : Un automate communicant (ou Channel System, CS *en abrégé*) est un n-uplet $\Gamma =< S, C, M, \Delta >$ composé des éléments suivants :

- $S = \{s, ...\}$ un ensemble fini d'états de contrôle,
- $C = \{c1, ...\}$ un ensemble fini de canaux,
- $M = \{m, ...\}$ un alphabet fini de messages, et
- $\Delta = \{\delta, ...\}$ un alphabet fini de règles de transition.

Les règles de transition forment un sous-ensemble de $S \times (\{C \times \{?, !\} \times M\} \cup \{\surd\}) \times S$. Si $\delta = (s, op, s') \in \Delta$ est une règle, op est appelé l'opération de δ et on note $\delta : s \xrightarrow{op} s'$. On écrira plus simplement $c!m$ (resp. $c?m$) pour $(c, !, m)$ (resp. $(c, ?, m)$). On détaille ci-dessous la signification des différentes opérations.

- $c!m$ envoyer le message m sur le canal c,
- $c?m$ lire le message m sur le canal c,
- $\surd$ action interne d'un processus.

Sémantique opérationnelle : Dans la suite on considère un automate communicant $\Gamma =< S, C, M, \Delta >$ fixé.

On appelle *configuration* de Γ un couple $\theta = (q, w)$ constitué d'un état de contrôle

$s \in S$ et d'une valuation des canaux : $w : C \to M^*$. Une configuration représente l'état du système, à la fois l'état de contrôle dans lequel se trouve l'automate fini, et les contenus des canaux. Dans la suite, on notera pour simplifier ϵ à la fois pour le mot vide et pour désigner la valuation ayant tous les canaux vides. L'ensemble de toutes les configurations est noté $Conf$. Dans l'exemple de la figure 3.3, la configuration dans laquelle se trouve l'automate communicant est $(E_2R_2, (100, 10))$ c'est-à-dire que l'état de contrôle est E_2R_2 (grisé), le contenu du canal c_1 est 100 et le contenu du canal c_2 est 10.

Définition 2 (Règle tirable) : Une règle de transition $\delta : (s, op, s')$ est tirable dans la configuration $\theta = (t, w)$ si

- premièrement $s = t$ et
- l'opération op est réalisable à partir de w ; formellement
  écriture ou action interne aucune contrainte sur w
  lecture si $op = c?m$, alors il existe $w \in M^*$ tel que $w(c) = mw$, c'est-à-dire qu'un message m se trouve en tête du canal c.

L'ensemble des règles de transition de Δ tirables dans une configuration θ donnée est noté $\Delta(\theta)$. Toujours sur l'exemple du modèle du bit alterné, et pour la configuration courante, $\Delta(E_2R_2, (100, 10)) = \{E_2R_2 \overset{c_1?1}{\to} E_2R_1, E_2R_2 \overset{c_1!1}{\to} E_2R_2, E_2R_2 \overset{c_2!0}{\to} E_2R_1, E_2R_2 \overset{c_2?1}{\to} E_1R_2\}$.

Remarque 3 : on fera souvent l'hypothèse qu'il n'y a aucune configuration bloquante, et donc que $\Delta(\theta) \neq 0$ pour toute configuration $\theta \in Conf$. Cela revient à supposer en particulier que pour chaque état de contrôle $s \in S$, il existe une règle de transition $\delta = (s, op, s')$ où op est une opération du type écriture ou action interne puisque ce sont les seules qui sont tirables lorsque les canaux sont vides.

A un automate communicant, on associe un système de transitions, dont les états sont les configurations de $Conf$, et les transitions sont décrites ci-dessous. On note $\to_{perf}$ la relation de transition. Soient θ et τ deux configurations. On note $\delta : \theta \to_{perf} \tau$ (ou encore $\theta \overset{\delta}{\to}_{perf} \tau$) si δ est tirable dans θ, et τ est le résultat de l'application de la règle de transition δ à la configuration θ. Détaillons ce dernier point. Supposons que $\delta = (s, op, t)$, $\theta = (s, w)$ et $\tau = (t, v)$ (l'état de contrôle de θ est nécessairement s pour que δ soit tirable). Les contenus des canaux v après transition sont obtenus à partir de w et en fonction de l'action op comme on s'y attend :

écrire $c!m$ **:** v est obtenu à partir de w en écrivant m à la fin du contenu du canal c :

$$v(d) \stackrel{def}{=} \begin{cases} w(d) & \text{si} \quad d \neq c \\ w(d)m & \text{sinon} \end{cases}$$

lecture $c?m$ **:** v est obtenu à partir de w en lisant m en tête du canal c :

$$v(d) \stackrel{def}{=} \begin{cases} w(d) & \text{si} \quad d \neq c \\ w & \text{sinon, et si} \quad w(d) = mw \end{cases}$$

action interne $\surd$ **:** le contenu des canaux est inchangé, $v = w$.

On note $ST_\Gamma = (Conf, \rightarrow_{perf})$ le système de transitions ainsi obtenu. Telle que nous l'avons définie, la relation de transition $\rightarrow_{perf}$ est un sous-ensemble de $Conf \times \Delta \times Conf$. Parfois, on la verra comme un sous-ensemble de $Conf \times Op \times Conf$ où Op est l'ensemble des opérations.

Prédécesseurs et successeurs : on définit les prédécesseurs et les successeurs d'une configuration et d'un ensemble de configurations comme habituellement. Tout d'abord, les prédécesseurs et successeurs d'une configuration θ par une règle de transition δ sont donnés par :

$$Pre[\delta](\theta) \stackrel{def}{=} \{\tau \in Conf | \tau \stackrel{\delta}{\rightarrow}_{perf} \theta\}$$

$$Post[\delta](\theta) \stackrel{def}{=} \{\tau \in Conf | \theta \stackrel{\delta}{\rightarrow}_{perf} \tau\}$$

Ensuite, pour $A \subseteq Conf$,

$$Pre[\delta](A) \stackrel{def}{=} \bigcup_{\theta \in A} Pre[\delta](A) \quad et \quad Post[\delta](A) \stackrel{def}{=} \bigcup_{\theta \in A} Post[\delta](A),$$

$$Pre(A) \stackrel{def}{=} \bigcup_{\delta \in \Delta} Pre[\delta](A) \quad et \quad Post(A) \stackrel{def}{=} \bigcup_{\delta \in \Delta} Post[\delta](A).$$

Pouvoir d'expression : Le modèle des automates communicants possède la puissance des machines de Turing [13]. Plus précisément, les automates communicants permettent de simuler des machines de Turing en temps quadratique. Un canal de communication est suffisant pour simuler un ruban de machine de Turing ; les écritures et lectures en milieu de ruban nécessitent de permuter circulairement les messages puisqu'on ne peut qu'écrire en queue et lire en tête. Ceci explique le coût quadratique du codage.

Le fait qu'on puisse simuler une machine de Turing par un automate communicant entraîne l'indécidabilité de tous les problèmes de vérification non triviaux, conformément au Théorème de Rice. Par exemple, l'accessibilité (savoir si une configuration donnée -

états des processus et contenu des canaux - est accessible à partir d'une configuration initiale) est un problème indécidable.

Des restrictions sur le modèle permettent de simplifier les problèmes et en particulier entraînent des résultats de décidabilité. C'est bien sûr le cas si l'on borne la taille maximale des canaux. On obtient alors un système ayant un nombre fini de configurations et l'accessibilité devient décidable. Le problème est alors PSPACE-complet. Néanmoins, il est commode de conserver des canaux possiblement infinis pour, par exemple, prendre en compte les délais arbitraires de transmission des messages. Dans le cadre de canaux à capacité infinie, une façon de restaurer la décidabilité de certaines propriétés est de se restreindre aux systèmes communicants pour lesquels l'ensemble d'accessibilité est régulier [62], c'est-à-dire dans le cas où, pour tout état de contrôle, l'ensemble des contenus de canaux possibles à partir d'une configuration initiale fixée constitue un langage régulier. Il se trouve que cette propriété de régularité est fréquente en pratique. C'est le cas par exemple pour le modèle du bit alterné de la figure 3.3, quelle que soit la configuration initiale. Cependant l'algorithme de décision associé n'est pas utilisable en pratique.

Une façon de « simplifier » le modèle est de considérer que les canaux ne sont pas totalement fiables, et peuvent perdre des messages. Cette hypothèse est de plus pertinente dans le cas de modélisation de protocoles de communication, car les pertes de messages sont tout à fait réalistes dans ce domaine. Dans le modèle des automates communicants par canaux à pertes, certains problèmes, dont l'accessibilité, sont décidables alors qu'ils sont indécidables dans le cas de canaux parfaits.

3.2 Automates temporisés

Les automates temporisés sont des automates munis d'un contrôle fini et un nombre fini d'horloges. La section 3.2.1 et la section 3.2.2 présentent le formalisme qui nous sera nécessaire pour comprendre les modèles de systèmes temporisés. La section 3.2.1 présente l'alphabet et la notion de temps utilisé dans les automates temporisés. La section 3.2.2 présente les horloges et les opérations sur les horloges. Les horloges sont des variables qui évoluent au cours du temps, toutes à la même vitesse, celle du *temps universel* qui pourrait être représenté par une horloge qui n'est jamais remise à zéro. Cette horloge universelle est souvent implicite.

3.2.1 Alphabet et notion de temps

Si Z est un ensemble, nous notons Z^* (respectivement Z^w) l'ensemble des suites finies (respec- tivement infinies) d'éléments de Z. Souvent, nous parlerons de mots finis sur l'alphabet Z pour les éléments de Z^* et de mots infinis sur l'alphabet Z pour les éléments de Z^w. Nous noterons de manière plus générale Z^∞ l'union de ces deux ensembles Z^* et Z^w.

Nous noterons de manière générique $\mathbb{T}$ un domaine de temps : ce pourra donc être $\mathbb{Q}^+$, l'ensemble des nombres rationnels positifs, ou bien $\mathbb{R}^+$, l'ensemble des nombres réels positifs.

Dans tout ce qui suit, Σ représente un alphabet fini d'*actions*. En outre, ε désigne une action qui n'est pas dans Σ. Cette action est dite *silencieuse*. L'ensemble $\Sigma \cup \{\varepsilon\}$ est noté Σ_ε. Une *suite de dates* sur le domaine $\mathbb{T}$ sera une suite finie ou infinie, croissante (strictement ou pas) $\tau = (t_i)_{i\geq 1} \in \mathbb{T}^\infty$. Un *mot temporisé* $u = (a_i, t_i)_{i\geq 0}$ est un élément de $(\Sigma \times \mathbb{T})^\infty$ tel que $(t_i)_{i\geq 0}$ est une suite de dates. Nous pourrons aussi écrire u comme une paire (σ, τ) où $\sigma = (a_i)_{i\geq 1}$ et $\tau = (t_i)_{i\geq 0}$ sont des suites de même longueur (par convention, deux suites infinies seront dites de même longueur). Un mot classique représente une mise en séquence d'actions. Un mot temporisé modélise aussi une mise en séquence d'actions en donnant en plus une information sur les dates auxquelles ces actions sont effectuées.

3.2.2 Horloges et opérations sur les horloges

Nous considérons un ensemble X de variables appelées *horloges*. Ces variables prennent leurs valeurs dans le domaine de temps $\mathbb{T}$ que nous considérons. Une *valuation* des horloges de X est ainsi une application $v : X \to \mathbb{T}$. L'ensemble de toutes les valuations des horloges de X est noté $\mathbb{T}^X$. Nous écrirons aussi parfois $\mathbb{T}^n$ à la place de $\mathbb{T}^X$ (quand $|X| = n$).

Si $d \in \mathbb{T}$ et si v est une valuation, la notation $v + d$ désigne la valuation définie par $(v + d)(x) = v(x) + d$ pour tout $x \in X$. Si $v \in \mathbb{T}^X$ est une valuation et si $C \subseteq X$, nous notons $[C \leftarrow 0]v$ la valuation obtenue à partir de v en remettant les horloges de C à zéro, c'est-à-dire la valuation définie par :

$$\begin{cases} ([C \leftarrow 0]v)(x) = 0 & \text{si} \quad x \in C \\ ([C \leftarrow 0]v)(x) = v(x) & \text{si} \quad x \notin C \end{cases}$$

Si v est une valuation et si $C \subseteq X$, alors nous notons $v \mid C$ la restriction de v aux horloges de C. Etant donnés deux ensembles d'horloges disjoints X_1 et X_2, ainsi que deux valuations v_1 et v_2 pour les ensembles d'horloges X_1, respectivement X_2, $v_1 : v_2$ représente

la valuation pour les horloges de $X_1 \cup X_2$ telle que

$$\begin{cases} (v_1 : v_2)(x) = v_1(x) & \text{si} \quad x \in X_1 \\ (v_1 : v_2)(x) = v_2(x) & \text{si} \quad x \in X_2 \end{cases}$$

3.2.3 Contraintes d'horloges

Si X est un ensemble d'horloges, nous définissons l'ensemble des *contraintes d'horloges* (ou tout simplement contraintes) sur X comme étant l'ensemble, noté $C(X)$, engendré par la grammaire suivante :

$$\varphi ::= x \sim c \mid x - y \sim c \mid \varphi \wedge \varphi$$

où $x, y \in X$, $c \in \mathbb{Q}$ et $\sim \in \{<, \leq, =, \geq, >\}$.

Nous considérons aussi un sous-ensemble propre de ces contraintes d'horloges. Cet ensemble conserve les contraintes qui comparent une horloge et une constante, mais n'autorise plus de comparaisons entre horloges. Ces contraintes sont appelées non diagonales. L'ensemble des contraintes d'horloges non diagonales est noté $C_{df}(X)$ et peut être décrit par la grammaire suivante :

$$\varphi ::= x \sim c \mid \varphi \wedge \varphi$$

où $x \in X$, $c \in \mathbb{Q}$ et $\sim \in \{<, \leq, =, \geq, >\}$.

Nous disons qu'une contrainte d'horloges est *k-bornée* (k étant un entier) si toutes les constantes qui apparaissent dans sa définition sont bornées par k.

La relation de satisfaction $\mid =$ pour les contraintes d'horloges est définie sur l'ensemble des valuations des horloges de X, inductivement, de la façon suivante (v est une valuation de $\mathbb{T}^X$) :

$$\begin{aligned} v \mid = x \sim c &\iff v(x) \sim c \\ v \mid = x - y \sim c &\iff v(x) - v(y) \sim c \\ v \mid = \varphi_1 \wedge \varphi_2 &\iff v \mid = \varphi_1 \; et \; v \mid = \varphi_2 \end{aligned}$$

Une contrainte d'horloges g représente un sous-ensemble (convexe) de $\mathbb{T}^X$. De façon classique, nous noterons parfois $v \in g$ (si $v \in \mathbb{T}^X$) au lieu de $v \mid = g$. Le fait que les sous-ensembles de $\mathbb{T}^X$ définis par une contrainte soient convexes facilite leur manipulation. Cependant, dans la définition des contraintes d'horloges, nous aurions pu ajouter l'opérateur de comparaison $\neq$ ainsi que la disjonction $\wedge$, mais les ensembles de $\mathbb{T}^X$ définis n'auraient alors pas été convexes. Grâce à des transformations semblables à celles que l'on

peut effectuer dans le cadre du calcul propositionnel, nous nous serions alors retrouvés au cas défini ici.

Nous allons maintenant présenter le modèle des automates temporisés défini par Rajeev Alur et David Dill dans le début des années 1990 [2, 3, 4].

3.2.4 Définition du modèle introduit par Alur et Dill

Un automate temporisé est un 7-uplet $A = (Q, X, \Sigma, Q_0, F, R, T)$ où

- Q est un ensemble fini d'états,
- X est un ensemble fini d'horloges,
- Σ est un alphabet fini d'actions,
- $Q_0 \subseteq Q, F \subseteq Q, R \subseteq Q$ sont respectivement les états initiaux, les états finals et les états répétés,
- $T \subseteq Q \times C(X) \times \Sigma \times 2^X \times Q$ est l'ensemble des transitions.

Un chemin dans l'automate temporisé A est une suite finie ou infinie de transitions consécutives

$$q_0 \xrightarrow{g_1,a_1,C_1} q_1 \xrightarrow{g_2,a_2,C_2} ... \xrightarrow{g_n,a_n,C_n} q_n...$$

Si $u = (a_1, t_1)...(a_n, t_n)...$ est un mot temporisé de $(\Sigma \times \mathbb{T})^\infty$, une exécution sur le chemin précédent pour le mot u est

$$(q_0, v_0) \xrightarrow[t_1]{g_1,a_1,C_1} (q_1, v_1) \xrightarrow[t_2]{g_2,a_2,C_2} ... \xrightarrow[t_n]{g_n,a_n,C_n} (q_n, v_n)...$$

où $(v_i)_{i\geq 0}$ est une suite de valuations d'horloges telle que pour tout $x \in X$, $v_0(x) = 0$ et pour tout $i \geq 0$,

$$\begin{cases} v_i + 1 = [C_{i+1} \leftarrow 0](v_i + t_{i+1} - t_i) \\ v_i + t_{i+1} - t_i \quad | = \quad g_i \end{cases}$$

[Par convention, la date t_0 correspond à la date de début d'exécution du mot temporisé, nous supposons donc que $t_0 = 0$.]

Une telle exécution est dite acceptante pour u si q_0 est un état initial et

- soit le chemin est fini et se termine dans un état final,
- soit le chemin est infini et passe infiniment souvent par au moins un état répété.

Un mot u est accepté par l'automate A s'il existe une exécution acceptante pour u dans A. Le langage accepté par A est l'ensemble des mots (finis ou infinis) acceptés par A et est noté $L(A)$.

Remarque : La condition d'acceptation des mots infinis est une condition de Büchi. Nous aurions pu considérer d'autres types de condition d'acceptation comme celles de Müller ou de Rabin [23, 66]. Fondamentalement, les résultats qui suivent seraient restés les mêmes.

Exemple 1 : Considérons le langage $L = \{((ab)^w, (t_i)_{i\geq 1}) \quad | \quad \exists i, \forall j \geq i, (t_{2j} \leq t_{2j-1} + 2)\}$. Il est accepté par l'automate temporisé de la figure 3.4.

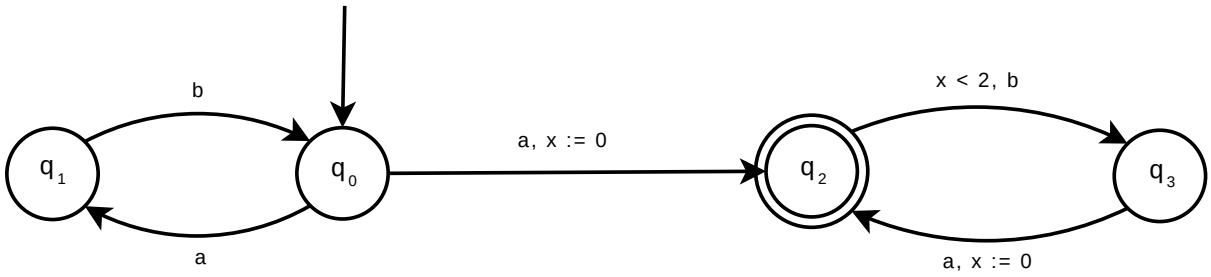

FIG. 3.4 – Le langage $L = \{((ab)^w, (t_i)_{i\geq 1}) \quad | \quad \exists i, \forall j \geq i, (t_{2j} \leq t_{2j-1} + 2)\}$

Automates temporisés avec ε-transitions. De la même manière, nous définissons un automate temporisé avec ε-transitions sur l'alphabet Σ comme étant un automate temporisé sur l'alphabet $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$. Les transitions de la forme $q \xrightarrow{g,\varepsilon,C} q'$ sont appelées des ε-transitions. Les mots acceptés par ces automates sont a priori des mots sur l'alphabet $(\Sigma_\varepsilon \times \mathbb{T})^\infty$, mais comme ε est une « action silencieuse », nous réduirons ces mots temporisés à des mots temporisés sur l'alphabet $(\Sigma \times \mathbb{T})^\infty$ en effaçant simplement les lettres de la forme (ε, t). Par exemple, le mot $(a, 0.2)(b, 1.7)(\varepsilon, 2)(\varepsilon, 2.3)(a, 2.5)(\varepsilon, 2.6)$ se réduit au mot $(a, 0.2)(b, 1.7)(a, 2.5)$. Le langage accepté par un automate temporisé avec ε-transitions est l'ensemble des mots temporisés de $(\Sigma \times \mathbb{T})^\infty$ qui sont obtenus de cette manière à partir d'un mot de $(\Sigma_\varepsilon \times \mathbb{T})^\infty$ lu dans l'automate.

3.2.5 Propriétés de clôture des automates temporisés

Les langages acceptés par les automates temporisés vérifient les propriétés de clôture suivantes :

Proposition 2 [3, 4] : L'ensemble des langages acceptés par des automates temporisés est clos par union et par intersection. Par contre, cet ensemble n'est pas clos par passage

au complémentaire.

Exemple 3 : Le langage $\{(a,t_1)...(a,t_n) \quad | \quad \exists i < j.t_j = t_i + 1\}$ sur l'alphabet $\Sigma = \{a\}$ est accepté par l'automate temporisé de la figure 3.5.

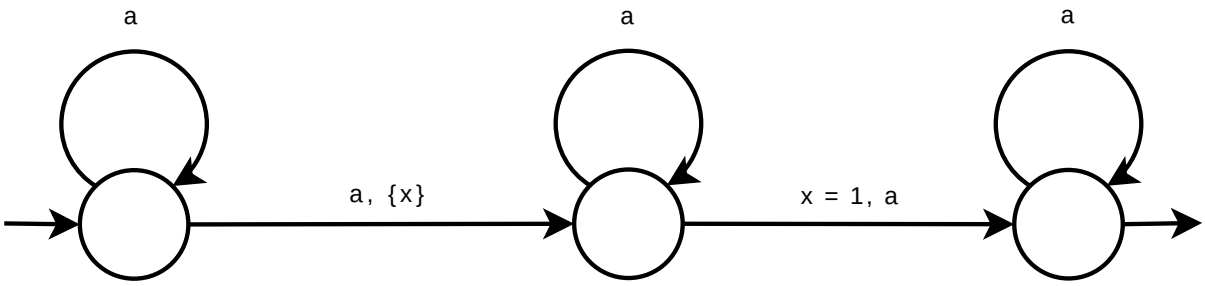

FIG. 3.5 – Le langage $\{(a,t_1)...(a,t_n) \quad | \quad \exists i < j.t_j = t_i + 1\}$ sur l'alphabet $\Sigma = \{a\}$

Par contre, son complémentaire L n'est reconnu par aucun automate temporisé. La preuve de ce fait n'est pas décrite dans [4], mais dans [12].

3.3 Conclusion du chapitre

Dans ce chapitre, nous avons introduit les fondements théoriques de notre langage d'injection de fautes.

Notre langage, fondé en partie sur le modèle des automates communicants, hérite ainsi de la puissance d'expréssivité des machines de Turing [13]. Le fait qu'on puisse simuler une machine de Turing par un automate communicant entraîne l'indécidabilité de tous les problèmes de vérification non triviaux. Des restrictions sur le modèle permettent de simplifier les problèmes et en particulier entraînent des résultats de décidabilité. C'est bien sûr le cas si l'on borne la taille maximale des canaux. On obtient alors un système ayant un nombre fini de configurations et l'accessibilité devient décidable. Ainsi, une façon de « simplifier » le modèle est de considérer que les canaux ne sont pas totalement fiables, et peuvent perdre des messages. Cette hypothèse est de plus pertinente dans le cas de modélisation de protocoles de communication, car les pertes de messages sont tout à fait réalistes dans ce domaine. Dans le modèle des automates communicants par canaux à pertes, certains problèmes, dont l'accessibilité, sont décidables alors qu'ils sont indécidables dans le cas de canaux parfaits.

Cependant, afin de limiter l'intrusion de l'injecteur de fautes et de respecter le scénario spécifié, nous allons conserver la modélisation avec les canaux totalement fiables. En effet,

le bon déroulement d'un scénario de fautes dépend de la réception des messages et si l'un des messages est perdu, le déroulement d'un test peut alors diverger du scénario prévu. Une solution serait d'utiliser des mécanismes de tolérance à la perte de messages, mais dans ce cas, deux problèmes fondamentaux sont soulevés : 1) ils ne régleraient pas forcément le problème du respect du scénario de fautes car ils entraîneraient alors des délais pouvant être très importants (dus à la détection et à la ré-émission des messages perdus) et faisant ainsi diverger le test du scénario prévu, et 2) ils entraîneraient une utilisation supplémentaire des ressources des machines où les processus de l'application instrumentée sont exécutés et impliqueraient alors une augmentation de l'intrusion de l'injecteur de fautes qu'il est préférable d'éviter.

Nous avons enrichi ce modèle avec des temporisateurs. Ces temporisateurs sont équivalents aux horloges du modèle des automates temporisés avec ε-transitions. L'ensemble Σ représentant l'alphabet fini d'actions est constitué des actions sur le processus de l'application instrumentée, *i.e.* l'injection de fautes. Les ε-transitions, dites transitions silencieuses, correspondent alors aux transitions dues à l'envoi ou à la réception d'un message, ou d'une manière plus générale, à une transition n'impliquant pas une injection de fautes.

Finalement, nous avons considéré les événements internes correspondant aux événements provenant de l'application instrumentée (comme l'appel par celle-ci à une fonction particulière). Ces événements internes sont intégrés au modèle en les considérant comme des réceptions de message et en associant aux automates modélisant le scénario de fautes des automates modélisant l'application instrumenté envoyant les messages correspondant aux événements qu'elle génère.

Ce modèle d'automate communicant temporisé nous sert ainsi de modèle pour notre langage de spécification de scénarios de fautes.

Chapitre 4

Le Langage FAIL (FAult Injection Language)

4.1 Principes du langage

Les scénarios de fautes se font par la description d'automates nommés pouvant être associés à une ou plusieurs machines du système. L'injection de fautes se fait à un moment précis, qu'il soit dépendant du temps ou d'un quelconque autre événement. Nous avons choisi d'utiliser des règles gardées : lorsqu'un événement se produit, une garde est alors valide (une garde est une expression à valeur booléenne) et les actions de la règle correspondante peuvent ainsi être exécutées.

L'injection de fautes doit pouvoir se faire suivant l'état local d'un processus ou suivant un état global du système. C'est pourquoi nous avons choisi de structurer ces règles par l'utilisation d'automates synchronisés. En effet, lorsque des événements locaux se produisent, l'automate se place dans un état particulier selon les règles valides, ce qui permet alors de limiter la portée des règles en évitant l'utilisation excessive de variables rendant le scénario rapidement incompréhensible. De plus, la synchronisation des automates entre eux par l'utilisation de messages permet d'injecter des fautes suivant un état global du système.

Le déclenchement des fautes se fait par rapport au temps (time-based trigger) ou par rapport à des événements relatifs au programme testé (event-based trigger). En effet, ce dernier est effectué par la levée d'exceptions dépendantes du temps, d'un appel à une fonction particulière par le programme testé, de l'arrivée du programme testé à une ligne particulière du code source (lorsque celui-ci est disponible) ou de l'arrivée de messages provenant d'autres automates (pour l'injection de fautes selon un état global du système

ou pour la synchronisation d'une injection de fautes).

De ce fait, l'utilisation d'automates permet de limiter l'intrusion de l'injecteur de fautes en évitant de nombreux tests inutiles car seules les règles du noeud où l'automate se trouve sont susceptibles de causer la levée d'une exception.

Les gardes peuvent dépendre de variables aléatoires pour permettre l'injection de fautes probabilistes. En effet, des fonctions aléatoires sont prédéfinies, mais l'utilisateur peut utiliser des fonctions d'une bibliothèque quelconque, s'il désire effectuer des tests probabilistes plus poussés.

Les actions possibles sont l'envoi de messages à certaines machines pour notifier, par exemple, l'arrivée du programme testé dans un état particulier, des actions sur le programme testé qui correspondent à l'injection de fautes et des affectations de variables.

Les différentes actions sur le programme testé sont :

- « stop » : interrompt l'exécution du programme ;
- « continue » : reprend l'exécution du programme là où il s'est arrêté ;
- « halt » : interrompt définitivement l'exécution du programme ;
- « restart » : relance le programme dans son état d'origine.

Le langage de programmation de l'application testée est arbitraire et a peu d'influence sur le scénario de fautes. La prise en charge des différents langages de programmation dépend essentiellement de l'implantation de la plateforme d'injection de fautes utilisant le langage FAIL. L'implantation actuelle de la plateforme prend en charge les langages de programmation générant du code natif ainsi que le Java. Elle sera détaillée dans le chapitre suivant.

4.2 Syntaxe et sémantique du langage

4.2.1 Structure globale

On décrit des automates nommés pouvant être associés à une ou plusieurs machines du système comme dans la figure 4.1.

Les applications à tester tourneront sur de très nombreuses machines. Il faut donc offrir la possibilité à l'utilisateur d'associer un ou plusieurs automates à un groupe de machine afin de faciliter l'écriture des scénarios. La définition de ce groupe doit être facile, elle est décrite dans la figure 4.2.

Dans cet exemple, 500 machines du système seront soumises au scénario de fautes décrit par l'automate Adv1 et feront partie du groupe nommé g1.

Remarque 1 : lorsqu'un groupe est spécifié, une constante de même nom et de type

```
Daemon Adv1 { [...] }
Daemon Adv2 { [...] }
Daemon Adv3 { [...] }
Computer p1, p2, p3 { daemon = Adv1 ;}
Computer p4 { daemon = Adv2 ; }
Computer null { daemon = Adv3 ; }
```

FIG. 4.1 – Association automates/machines en FAIL

```
Group g1
{
    program = "server -p 5420" ;
    size = 500 ;
    daemon = Adv1 ;
}
```

FIG. 4.2 – Définition d'un groupe en FAIL

« tableau de machines » est automatiquement créé. Cette constante est un tableau contenant tous les identifiants des machines du groupe (voir la Section 4.2.8).

Remarque 2 : le champ « server -p 5420 » est facultatif, il correspond à la commande qui sera excutée lors du lancement du test (ici lors du lancement du test, la machine p1 lancera la commande « server -p 5420 »). Lors d'un « restart » cette commande sera exécutée ou la dernière commande éxécutée par la machine sera réexécutée (cas des applications auto-déployantes).

4.2.2 Structure des automates

Ce sont des automates probabilistes qui peuvent éventuellement communiquer par événements. Ils s'expriment sous forme de règles gardées. Ces règles sont composées d'une garde suivie d'une flèche et d'actions comme indiqué dans la figure 4.3.

Remarque : dans cet exemple, « ?ok » est une garde et « continue » une action. La garde « ?ok » signifie « lorsque je reçois le message ok ». Les actions servent à injecter les fautes. Ici, l'action « halt » signifie « le processus exécuté sur la machine s'interrompt définitivement ». En fait, les règles gardées et les actions sont un peu plus complexes et

```
Daemon Adv1
{
   ?ok -> halt ;
}
```

FIG. 4.3 – Exemple d'une règle dans un automate

seront expliquées par la suite.

Dans une suite d'actions de règle, les actions sont séparées par des virgules. Le point-virgule est utilisé à la fin d'une règle comme indiqué dans la figure 4.4.

```
Daemon test1
{
   int a = 0 ; // Definition et initialisation d'une variable entiere.
   ?ok -> continue, a = a+1 ;
}
```

FIG. 4.4 – Exemple de séparation avec la virgule et le point-virgule

Dans un automate, on peut définir des noeuds permettant de le structurer afin de limiter la portée des règles gardées. Chaque suite d'actions de règle doit se terminer par le noeud cible où l'automate se positionnera lorsque toutes les actions seront effectuées(grâce au mot clef « goto »). Si, à la fin d'une suite d'actions, il n'y a pas de noeud cible, alors par défaut le noeud cible sera celui auquel la règle appartient. La figure 4.5 présente un automate structuré par des nœuds.

4.2.3 Les commentaires

Les commentaires peuvent être écrits en étant insérés entre les signes « /* » et « */ ». On peut insérer des commentaires grâce aux signes « // ». Le commentaire débute alors aux signes « // » et se termine à la fin de la ligne. Les commentaires dans FAIL peuvent être imbriqués.

```
Daemon adv1
{
  node 1 :
     ?hello -> continue, goto 2; // Ici on va au noeud 2 lorsque toutes
  node 2 :                       // les actions sont effectuees.
     ?bye -> stop, goto 1; // Ici on retourne au noeud 1.
}
```

Fig. 4.5 – Un automate structuré par des nœuds

4.2.4 La phase d'initialisation

La figure 4.6 introduit la phase d'initialisation permet la déclaration de variables, leur initialisation et la définition de règles initiales.

```
Daemon exemp
{   // Phase d'initialisation.
    int count = 0; // Declaration du compteur "count" et initialisation a 0.
    init true -> !allez_y; // Diffusion d'un message initial "allez_y".
    // Regles de l'automate.
    ?crash -> count = count + 1; // Ici lorsqu'on recoit le message "crash",
}                                // on incremente la variable "count".
```

Fig. 4.6 – La phase d'initialisation

Dans cet exemple, l'automate « exemp » initialise un compteur « count » à zéro et diffuse un message « allez_y ». Puis il compte le nombre de message « crash » (En incrémentant le compteur « count »).

Remarque : le mot clef « init » permet la définition de règles initiales. Toute règle précédée de ce mot clef est une règle initiale. (voir la section 4.2.9)

4.2.5 Les différentes actions

Les actions de chaque règle sont une suite d'affectations, d'émission d'événements ou d'actions sur un processus.

Actions sur un processus

Les différentes actions possibles sur le processus sont les suivantes :

- « stop » : interrompt l'exécution du programme ;
- « continue » : reprend l'exécution du programme là où il s'est arrêté ;
- « halt » : interrompt définitivement l'exécution du programme ;
- « restart » : relance le programme à l'origine.

Actions d'affectation

L'utilisateur peut déclarer des variables qu'il peut utiliser dans les gardes et les actions (pour compter par exemple). Lorsque le signe « = » est utilisé dans une action, il correspond à une affectation d'une valeur à une variable comme le montre la figure 4.7.

```
Daemon Adv1
{
    int a = 0 ;
    ?incremente -> a = a + 1 ;
}
```

FIG. 4.7 – L'affectation dans un automate

Remarque : lors de la déclaration des variables, celles-ci doivent obligatoirement être initialisées (à l'aide du signe « = »).

Actions d'émission d'événements

Les actions d'émission d'événements sont en fait des diffusions de messages à toutes les machines, à un groupe de machines ou à une machine particulière. L'émission d'événements se fait avec le caractère « ! ».

```
Daemon Adv1
{
    ?coucou -> !hello ;
}
```

FIG. 4.8 – Emission d'un événement

```
Daemon Adv1
{
   ?coucou -> !go(G1); // diffusion du message "go" au groupe de machine G1.
}

Group G1
{
   size = 20;
   daemon = Adv1;
}
```

Fig. 4.9 – Diffusion à un groupe de machines

Dans la figure 4.8, le message « hello » est envoyé lorsqu'on reçoit le message « coucou ».

Dans la figure 4.9, le message « go » est diffusé à toutes les machines du groupe G1 lorsqu'on reçoit le message « coucou ».

```
Daemon Adv2
{
   ?coucou -> !go(Comp1);
}

Computer Comp2
{
   daemon = Adv2;
}
```

Fig. 4.10 – Envoi d'un message à une machine particulière

Dans la figure 4.10, le message « go » est envoyé à la machine « Comp1 » lorsque le message « coucou » est reçu.

L'utilisateur a aussi la possibilité d'envoyer des messages à une ou plusieurs machines particulière(s) d'un tableau de machines. Les signes « [», «] » et « .. » sont utilisés pour effectuer ces opérations. Pour cela il doit procéder de la manière suivante indiqué dans la figure 4.11 et la figure 4.12.

```
Daemon Adv8
{
   ?coucou -> !go(G1[4]) ;
}

Group G1
{
   size = 50 ;
   daemon = Adv8 ;
}
```

FIG. 4.11 – Envoi à une machine d'un tableau de machines

Dans la figure 4.11, on envoie « go » à la 4ième machine du tableau des machines du group G1 lorsqu'on reçoit le message « coucou ».

```
Daemon Adv8
{
 ?coucou -> !go(G1[1..5]) ;
}

Group G1
{
   size = 50 ;
   daemon = Adv8 ;
}
```

FIG. 4.12 – Envoi à plusieurs machines d'un tableau de machines

Dans la figure 4.12, on envoie « go » aux 5 premières machines du tableau des machines du groupe G1 lorsqu'on reçoit le message « coucou ».

Remarque : l'envoi à plusieurs processus correspond à des envois par « tranches ». L'utilisateur peut spécifier l'envoi d'un message à tous les processus compris entre la X ième et la Y ième position dans le tableau grâce à la syntaxe « X..Y » où X et Y sont des variables (avec X strictement inférieur à Y). L'utilisateur peut spécifier l'envoi d'un message à

plusieurs tranches en les séparant par des virgules (par exemple, pour l'envoi du message « go » aux machines 1 à 5 et 9 à 15, on obtient la syntaxe suivante : !go(G1[1..5,9..15])).

On peut aussi attacher une valeur entière à un message en utilisant le symbole « : » après le nom du message comme le montre l'exemple de la figure 4.13.

```
Daemon Adv8
{
   ?coucou -> !go :4(G1[1..5]) ;
}

Group G1
{
   size = 50 ;
   daemon = Adv8 ;
}
```

Fig. 4.13 – Envoi d'un entier attaché à un message

Dans cet exemple, l'automate Adv8 envoie le message « go » aux 5 premières machines du tableau des machines du groupe G1 en attachant à ce message la valeur entière 4.

4.2.6 Les différentes gardes

Les gardes sont des conjonctions d'entités à valeur booléenne. La conjonction d'entités est réalisée avec les signes « && ». Différentes entités pouvant constituer une garde ont été définies.

Réception d'un message

La réception d'un message se fait à l'aide du caractère « ? ». La réception d'un message est ponctuel. Ce qui implique que lors d'une réception, les gardes sont testées et si l'une d'entre elles est valide, alors les actions correspondantes sont exécutées. A la fin de ces exécutions, toutes les gardes dépendantes de la réception de ce message ne sont plus valides (elles seront potentiellement valides lors d'une prochaine réception de ce message). La figure 4.14 présente un exemple de réception du message « coucou ».

Dans cet exemple, lorsque la machine Comp1 reçoit le message « coucou », elle fait crasher le processus qu'elle était en train d'exécuter (ici le programme « test »). Le

```
Daemon crasheur
{
   ?coucou -> halt ;
}

Computer Comp1
{
   program = "test" ;
   daemon = crasheur ;
}
```

FIG. 4.14 – Exemple de réception d'un message

message reçu peut contenir une valeur entière (cela dépend de l'envoi, voir la section 4.2.5), l'automate peut alors la stocker dans une variable grâce au caractère « : » placé après le nom du message, suivi de la variable à laquelle la valeur sera affectée.

La figure 4.15 présente un exemple de réception du message « coucou » avec affectation de la valeur qui lui est attachée à la variable « count ».

```
Daemon crasheur
{ int count = 0 ;
  ?coucou :count -> halt ; }

Computer Comp1
{ program = "test" ;
  daemon = crasheur ; }
```

FIG. 4.15 – Exemple d'affectation d'une valeur reçue

Dans cet exemple, lors de la réception du message « coucou », la valeur entière attachée est affectée à la variable « count » ; cette variable peut alors être utilisée dans les actions ou même dans la garde où il y a eu réception du message. La figure 4.16 illustre l'utilisation de cette variable.

Dans cet exemple, lors de la réception du message « coucou », la valeur attachée est affectée à la variable « count », puis un test d'égalité est effectué et suivant la valeur de

```
Daemon crasheur
{ int count = 0 ;
 ?coucou :count && count == 3 -> halt ; }

Computer Comp1
{ program = "test" ;
daemon = crasheur ; }
```

FIG. 4.16 – Exemple d'utilisation d'une valeur reçue

« count », la commande « halt » est effectuée ou pas.

Test d'une variable

Une garde peut être un test d'égalité, d'inégalité ou de différence d'une variable par rapport à une valeur. La figure 4.17 présente ces différents tests.

```
Daemon test
{ int test = 0 ;
 ?incr && test<>19 -> test = test + 1 ; // difference
 ?incr && test==19 -> halt ; // egalite }
```

FIG. 4.17 – Exemple de test d'une variable

Dans cet exemple, l'automate « test » fait crasher le processus lorsqu'il reçoit 20 messages « incr ».

- Remarque 1 IMPORTANTE : les gardes doivent être vraies à des instants ponctuels. C'est pour cela que FAIL définit des types d'entités et des restrictions sur les gardes (voir la section 4.2.11).
- Remarque 2 : pour les inégalités, les signes utilisés sont les signes suivants : « < » pour l'inférieur stricte,« > » pour le supérieur stricte, « <= » pour l'inférieur ou égal et « >= » pour le supérieur ou égal.

Test d'un minuteur

Une garde peut être un test sur un minuteur. La figure 4.18 présente un exemple montrant la déclaration d'un minuteur et l'utilisation de celui-ci dans une garde.

```
Daemon test
{  // Declaration du minuteur.
    time_g timer = 20;
    timer -> halt; }
```

FIG. 4.18 – Exemple de test d'un minuteur

Dans cet exemple, un minuteur de 20 secondes est déclaré. Lors de l'expiration de ce minuteur au bout de 20 secondes d'exécution de l'application testée, l'automate interrompt définitivement cette application.

Il existe deux types de minuteurs :

- les minuteurs globaux déclarés grâce au mot clé *time_g*,
- et les minuteurs locaux déclarés grâce au mot clé *time_l*.

Un minuteur global a pour unité de temps la seconde et un minuteur local a pour unité de temps la milliseconde. Dans un scénario, les deux types de minuteur peuvent être utilisés selon la précision nécessaire en utilisant la déclaration correspondante.

Test sur appel de fonction

Une garde peut être un test sur un appel d'une fonction du programme sous test. La figure 4.19 présente un exemple montrant la déclaration d'une fonction à tester et l'utilisation de cette fonction dans une garde.

```
// Declaration de la fonction "calcul" a espionner.
spyfunc calcul;

Daemon test
{ before(calcul) -> halt; }
```

FIG. 4.19 – Exemple de test sur appel de fonction

Dans cet exemple, lorsque le programme testé fait appel à la fonction « calcul », l'automate le fait crasher avant qu'il n'exécute cette fonction.

Test sur une ligne du code source particulière

Une garde peut être un test sur l'arrivée du programme testé à une ligne particulière du code source. L'exemple de la figure 4.20 nous montre la déclaration d'un lien vers une ligne particulière d'un fichier source du programme à tester et l'utilisation de ce lien dans une garde.

```
Daemon test
{ ln ligne_test = toto.c :25 ; // Declaration d'un lien vers
                               // la ligne 25 du fichier toto.c
  ligne_test -> halt ; } // Utilisation du lien dans une garde

Computer Comp1
{ program = "client -i console" ;
  daemon = test ; }
```

Fig. 4.20 – Exemple de point d'arrêt sur une ligne du code source

Dans cet exemple, lorsque le programme *client -i console* exécuté sur la machine Comp1 arrive à la ligne 25 du fichier toto.c, il s'interrompt définitivement (il crashe). Les informations de débogage sont, dans ce cas, utilisées pour repérer les fonctions ou les lignes de code particulières.

4.2.7 Les variables et fonctions prédéfinies

L'utilisateur a accès à quelques variables prédéfinies (Toutes les variables et fonctions prédéfinies sont préfixées par « FAIL_ » et sont composées de lettres en majuscule et du signe « _ ») :

- le tableau contenant les identifiants de toutes les machines « FAIL_COMPUTERS ».
- une fonction prenant en entrée un tableau d'identifiants de machines et retournant sa taille : « FAIL_SIZE ».
- une fonction retournant un entier aléatoire uniforme dans [min,max] « FAIL_RANDOM(min, max) ».
- une fonction prenant en entrée un tableau d'identifiant de machine « tab_comp » et un entier « nb_comp » et retournant un tableau d'identifiant de taille « nb_comp » dont les éléments ont été choisis aléatoirement parmi ceux de « tab_comp », la fonction est la suivante : « FAIL_RANDOM_TABC(tab_comp, nb_comp) ».

4.2.8 Les différents types de variables et constantes

La définition de variable ou de constante est implicite et donc gérée par le compilateur. L'utilisateur n'a pas à spécifier si tel ou tel nom est une variable ou une constante.

Les différents types existants sont les suivants :

- la variable de type *entier naturel*. Elle est définie comme indiqué dans la figure 4.21.

```
Daemon Adv1
{ int nb_sms_recu = 0 ; } // Declaration et initialisation
                          // d'une variable nommee "nb_sms_recu".
```

FIG. 4.21 – Déclaration d'une variable de type *entier*

- la constante de type *numéro de ligne*. C'est une constante faisant référence à une ligne du code source. Elle ne peut être utilisée que dans la partie garde d'une règle. La définition et l'utilisation d'une telle constante est décrite dans la figure 4.22.

```
Daemon Adv3
{ // Declaration d'une constante, nommee "ligne_25",
  // faisant reference a la ligne 25 du fichier "toto.c".
  ln ligne_25 = "toto.c" :25 ;
  ligne_25 -> halt ; } // lorsque le programme teste arrive a la ligne 25
                       // du fichier "toto.c", il crashe.
```

FIG. 4.22 – Déclaration d'une variable de type *numéro de ligne*

  Remarque : pour pouvoir définir et utiliser ce type de constante le programme émulé doit être compilé en mode débug.

- la constante de type *fonction à tester*. C'est une constante faisant référence à une fonction du code source. Elle ne peut être utilisée que dans la partie garde d'une règle. La définition et l'utilisation d'une telle constante se font de la façon indiquée dans la figure 4.23.

  Remarque 1 : la définition des fonctions à tester ne se fait pas à l'intérieur de la définition des automates.

  Remarque 2 : la connaissance du code source du programme à tester est nécessaire, mais la compilation en mode débug ne l'est pas.

  Remarque 3 : l'utilisation du mot clef *before* a été choisie pour des raisons de

```
// declaration d'une fonction nommee "calcul" a espionner.
spyfunc float calcul ( int, int) ;

Daemon toto
{ before(calcul) -> halt ; }
```

FIG. 4.23 – La constante de type *fonction à tester*.

sémantique. En effet, il signifie que les actions de la règle seront effectuées avant l'exécution de la fonction *calcul*.

– la variable de type *tableau de machines*. C'est une variable contenant un tableau d'identifiants de machines. Elle est initialisée en utilisant une fonction d'une bibliothèque dynamique puis utilisée pour l'envoi de messages comme indiqué dans la figure 4.24.

```
Daemon Adv1
{ tabc tableau1 = calcul_tab(FAIL_COMPUTERS) ;
  ?go -> !on_crash(tableau1) ; }
```

FIG. 4.24 – La variable de type *tableau de machines*.

Remarque 1 : dans cet exemple, la fonction *calcul_tab* est une fonction d'une bibliothèque dynamique (voir la section 4.2.9).

Remarque 2 : les tableaux d'identifiants de processus sont des tableaux de n éléments (avec n le nombre de processus contenus dans le tableau). Ce sont des tableaux dont les cases (numérotées de 1 à n) contiennent des identifiants de processus (des entiers).

Pour avoir la taille d'un tableau, il suffit d'utiliser la fonction « FAIL_SIZE » qui prend en entrée un tableau de processus et retourne un entier comme l'indique la figure 4.25.

4.2.9 Déclaration de variables et gestion de l'aléatoire

La gestion du hasard se fait à l'aide de variables aléatoires. Ces variables sont initialisées à l'aide de fonctions aléatoires de bibliothèques extérieures ou de fonctions aléatoires prédéfinies. Elles sont ensuite utilisées dans les gardes des règles. Il existe deux façons de déclarer et d'initialiser les variables (aléatoires ou non) :

```
Daemon test
{ // Ici nb_g1 est initialise avec la valeur 500
  int nb_g1 = FAIL_SIZE (G1); }

Group G1
{ size = 500;
  daemon = test; }
```

FIG. 4.25 – La taille d'un tableau de machines.

– déclaration globale dans un automate :
  Lorsqu'une variable est déclarée globalement dans un automate, elle est initialisée lors du lancement de l'automate. Cette variable est visible et utilisable par tous les noeuds de l'automate. La déclaration et l'utilisation de variable globale se fait comme dans l'exemple de la figure 4.26

```
Daemon Mefisto
{ int wait_msg = 0;
  node 1 :
    ?msg && wait_msg==25 -> wait_msg = 0; goto 2;
    ?msg && wait_msg<>25 -> wait_msg = wait_msg + 1;
  node 2 :
    ?msg -> !kill_random, goto 1; }
```

FIG. 4.26 – Les variables globales.

  Dans cet exemple, l'automate nommé *Mefisto* attend 25 messages *msg* et diffuse le message *kill_random* lorsqu'il reçoit de nouveau le message *msg* (donc au 26ième message *msg*) et il recommence cette procédure indéfiniment.
– déclaration locale dans un automate :
  Lorsqu'une variable est déclarée localement dans un automate, elle est initialisée quand l'automate se positionne pour la première fois dans ce noeud et qu'il passe d'un autre noeud à ce noeud (en fait, les variables locales ne sont pas réinitialisées lors de récursion sur un même noeud). Ces variables ne sont visibles que dans le noeud où elles sont déclarées. La déclaration et l'utilisation de variables locales se font comme dans l'exemple de la figure 4.27.
  Remarque 1 : cet exemple correspond au même exemple que précédemment mais

```
Daemon Mefisto
{ node 1 :
      int wait_msg = 0 ;
      ?msg && wait_msg==25 -> goto 2 ;
      ?msg && wait_msg<>25 -> wait_msg = wait_msg + 1 ;
  node 2 :
      ?msg -> !kill_random, goto 1 ; }
```

FIG. 4.27 – Les variables locales.

en utilisant des variables locales.
Remarque 2 : lorsque les noeuds ne sont pas spécifiés, toutes les variables sont des variables globales comme dans l'exemple de la figure 4.28.

```
Daemon Mefisto
{ int wait_msg = 0 ;
  ?msg && wait_msg<>26 -> wait_msg = wait_msg + 1 ;
  ?msg && wait_msg==26 -> !kill_random, wait_msg = 0 ; }
```

FIG. 4.28 – Les variables dans un automate sans nœuds.

Cet exemple correspond au même exemple que précédemment mais en n'utilisant pas de noeuds dans l'automate.

– déclaration et utilisation d'une variable aléatoire :
l'utilisateur peut utiliser dans ses actions, ou pour initialiser une variable, le résultat de la fonction prédéfinie FAIL_RANDOM(min, max) qui renvoie un entier aléatoire uniforme compris dans [min, max]. Il peut aussi fournir d'autres fonctions dont il peut vouloir se servir via une bibliothèque dynamique. La figure 4.29 présente un exemple d'utilisation d'une variable aléatoire.

```
Daemon random_halt
{ int rand = FAIL_RANDOM(0,10) ;
  ?fail_or_not && rand == 4 -> halt ; }
```

FIG. 4.29 – Les variables aléatoires.

A la réception d'un message « fail_or_not », l'automate « random_halt » fait crasher le programme testé suivant la valeur de la variable aléatoire « rand ».

Remarque : on ne peut pas faire appel à une fonction dans une garde. Il faut initialiser une variable avec cette fonction puis l'utiliser dans la garde. Cette mesure est prise pour éviter certains problèmes sémantiques liés à l'implantation.

Utilisation de bibliothèques dynamiques

L'utilisateur peut vouloir utiliser des fonctions d'une bibliothèque dynamique. Par exemple pour calculer une variable aléatoire en utilisant une fonction aléatoire d'une bibliothèque dynamique. Pour cela il doit donner la signature de cette fonction ainsi que le nom de la bibliothèque où elle se trouve comme dans l'exemple de la figure 4.30.

```
/* declaration d'une fonction nommee "mon_hazard" prennant en entree
   deux entiers et retournant un entier. Cette fonction fait partie
   de la bibliotheque "libutils.so" */
function int mon_hasard(int, int) in library "libutils.so";

/* si a la reception de "fail_or_not", mon_hasard(3, 400) renvoi 4,
   alors arret definitif de l'application testee. */
Daemon Adv1
{ int rand = mon_hasard(3, 400);
  ?fail_or_not && rand == 4 -> halt; }
```

Fig. 4.30 – Les bibliothèques dynamiques.

Utilisation de la commande *always*

La commande *always* permet de forcer l'initialisation d'une variable pendant la récursion sur un même noeud (après une suite d'actions) comme le montre la figure 4.31.

Dans cet exemple, au bout de x secondes (x étant une variable aléatoire uniforme comprise entre 10 et 20), l'automate « coucou » diffuse le message « faire_1 » ou « faire_2 » selon la valeur de la variable aléatoire *rand* (la valeur de cette variable étant obtenue grâce à la fonction *mon_hasard* de la bibliothèque externe *libutils.so*). Une fois le message envoyé, la valeur du minuteur est recalculée et le minuteur est réarmé.

```
function int mon_hasard(int, int) in library "libutils.so" ;
Daemon coucou
{ // le temps global est en millisecondes
   always time_g timer = FAIL_RANDOM(10,20) ;
   always rand = mon_hazard(3, 400) ;
   timer && rand >= 100 ->!faire_1 ;
   timer && rand < 100 ->!faire_2 ; }
```

Fig. 4.31 – Utilisation de la commande *always*.

Initialisations gardées

Il est possible de définir une règle gardée comme faisant partie de l'initialisation. La syntaxe pour la définition de telles règles est présentée dans l'exemple de la figure 4.32.

```
Daemon toto
{ int rand = FAIL_RANDOM(1,10) ;
   // Definition d'une regle gardee initiale :
   init rand == 4 -> halt ; }
```

Fig. 4.32 – Les initialisations gardées.

Dans cet exemple, à son initialisation, l'automate tire un nombre aléatoirement et crashe ou non suivant la valeur de ce nombre.
Remarque 1 : ce type de règle ne sera testé que lors de l'initialisation.
Remarque 2 IMPORTANTE : les restrictions sur les entités des gardes pour ce type de règle (règle initiale) sont différentes des restrictions pour les règles standards (voir la section 4.2.11 introduisant les restrictions sur les gardes).

4.2.10 Précisions sur la sémantique du langage

- Lorsque la garde d'une règle est valide, alors les actions correspondantes sont exécutées séquentiellement.
- Si plusieures gardes sont valides à un instant donné, alors les actions de l'une des règles correspondantes seront effectuées. Le choix de la règle à traiter est fait de façon aléatoire ou non suivant le mode d'exécution des automates. Lorsque le mode

d'exécution n'est pas aléatoire, la règle choisie sera celle qui apparaît la première dans la description de l'automate.

- Une garde (non initiale) ne peut contenir qu'une seule réception de messages (voir la section 4.2.11 présentant les restrictions sur les gardes).
- Sémantique d'un exemple d'automate (envoi et réception de messages). Soit l'exemple de la figure 4.33.

```
Daemon exemp1
{ // envoi de message initial.
   init true -> !hello(P2), !hello2(P2) ; }

Daemon exemp2
{ bool msg_hello_rcv = false ;
   int sortie = 0 ;
   ?hello -> msg_hello_rcv = true, sortie = 1 ;
   ?hello2 && msg_hello_rcv -> sortie = 2 ; }

Computer P1
{ program = "test" ;
   daemon = "exemp1" ; }

Computer P2
{ program = "test" ;
   daemon = "exemp2" ; }
```

FIG. 4.33 – Envoi et réception de messages.

Lors de l'exécution du programme « test », le processus P1 envoie au processus P2 le message « hello » puis le message « hello2 ». A la réception du message « hello », le processus P1 met sa variable « msg_hello_rcv » à « 1 ». Puis à la réception de « hello2 », la valeur de la variable « msg_hello_rcv » est « 1 », donc le processus P1 fait « sortie = 2 ». Maintenant si l'on inverse l'ordre d'envoi des messages, le processus P1 ne fera pas « sortie=2 » car à la réception de « hello2 » (le premier message reçu) la valeur de « bool msg_hello_rcv » est « false ». Puis à la réception de « hello » il fera « sortie = 1 ».

4.2.11 Restrictions sur les gardes

Les deux types d'entités de garde

Dans FAIL, il existe deux types d'entités de garde. Les entités « testables » et les entités « interruptibles ».

- Dans une garde, on a obligatoirement une et une seule entité « interruptible ».
- Dans une garde, il peut y avoir un nombre quelconque d'entités de type « testable ».
- Dans une garde initiale, il ne peut y avoir que des entités de type « testable ».

Voici les types des différentes entités des gardes :

- Réception d'un message : « interruptible ».
- Test sur appel de fonction : « interruptible ».
- Test sur une ligne du code source : « interruptible ».
- Test d'un minuteur : « interruptible ».
- Test d'égalité, d'inégalité, de différence de variables : « testable ».

Pourquoi des types de gardes ?

Le système d'injection de fautes utilise des exceptions pour le déclenchement des fautes, ce qui permet un minimum d'intrusion lors de l'exécution de l'application. En effet, le daemon d'injection de fautes est inactif jusqu'à la levée d'une exception qui correspond aux entités de type « interruptible ». Lorsqu'une interruption est levée, la validité de toutes les gardes dépendantes de cette interruption est testée. Cette étape correspond au test des entités de type « testable ». Puis les actions de l'une des règles dont les gardes sont valides seront alors exécutées. C'est pour cela qu'une et une seule entité de type « interruptible » est nécessaire dans chaque règle standard. La validité des gardes des règles initiales est testée à l'entrée dans le noeud et ne doit donc pas dépendre d'une interruption. Les gardes des règles initiales ne sont donc constituées que d'entités de type « testable ». Lorsque plusieurs gardes sont valides, le choix de la règle dont les actions seront exécutées se fait, soit de manière aléatoire, soit de manière déterministe en choisissant systématiquement la première règle valide. En effet, ce deuxième mode est nécessaire pour permettre la reproductibilité des tests bien qu'il soit peut être un peu moins intuitif.

4.3 Exemples de scénarios possibles

Dans cette section, nous allons expliquer le fonctionnement du langage FAIL à travers l'explication de deux scénarios de fautes. Ces scénarios ont été écrits pour une application de type client/serveur constitué d'un serveur et de 500 clients (chaque client étant exécuté sur une machine distincte).

4.3.1 Premier exemple

L'exemple de la figure 4.34 présente un scénario de fautes induisant le crash des processus de l'application distribuée sur une machine, déterminée de façon aléatoire, toutes les 2 minutes. Les clients seront exécutés sur 500 machines et seront soumises à ce scénario de fautes. Une machine non sousmise au scénario de fautes exécutera le serveur.

```
1   Daemon dem_root
2   {  // Definition du minuteur de 2 minutes
3      // (le temps global est en seconde)
4      time_g timer = 120 ;
5      // gestion de l'aleatoire
6      always int rand = FAIL_RANDOM (1, 500) ;
7      timer -> !crash(All_comp[rand]) ; }
8
9   Daemon dem_other
10  { ?crash -> halt ; }
11
12  Computer Comp_root
13  {  program = "server -p 3420 -i console"
14     daemon = dem_root ; }
15
16  Group All_comp
17  {  program = "client -p 3420 -i console"
18     size = 500 ;
19     daemon = dem_other ; }
```

FIG. 4.34 – Premier scénario de fautes

La ligne 12 correspond à la définition de la machine, nommée Comp_root, exécutant

le server. L'automate dem_root définit à la ligne 1 est associé à cette machine grâce à la ligne 14.

La ligne 16 correspond à la définition du groupe de machines, nommée All_comp, dont chaque machine exécute un client. L'automate dem_other définit à la ligne 9 est associé à ce groupe de machines grâce à la ligne 19. La taille de ce groupe est de 500 (ligne 18) comme l'exige le test.

Les lignes 13 et 17 correspondent, respectivement, à la commande permettant l'exécution du serveur et à la commande permettant l'exécution des clients.

L'automate dem_other est très simple, il est constitué d'une seule ligne, la ligne 10. Lors de l'exécution, cette automate se place en attente du message *crash*. Lorsque ce message est reçu, le processus client local à la machine exécutant cet automate est interrompu définitivement grâce à la commande *halt*.

L'automate dem_root est exécuté sur la machine exécutant le serveur. Cet automate détermine quelle machine sera victime d'une faute en envoyant un message crash à une machine du groupe All_comp toutes les 2 minutes. La ligne 4 permet de définir le minuteur fixé à 2 minutes. La ligne 6 permet d'affecter une valeur aléatoire comprise entre 1 et 500 à la variable entière *rand*. Cette variable sera recalculée à chaque expiration du minuteur grâce à l'utilisation de *always*. La ligne 7 est la règle gardée permettant l'envoi du message *crash* à la machine selectionnée grâce à l'action *!crash(All_ comp[rand])* lorsque le minuteur expire (ce minuteur est activé grâce à la garde *timer*).

Remarque : s'il n'y a pas de processus lancé sur une machine recevant le message « crash », alors la commande halt n'a aucun effet.

4.3.2 Deuxième exemple

L'exemple de la figure 4.35 présente un scénario de fautes où chaque machine a une probabilité de 0.5 de faire crasher son processus au bout d'un temps aléatoire compris entre 100 et 200 secondes. Cependant, le crash d'un client ne sera effectué que lorsque celui essaiera d'exécuter la fonction *execute* ou arrivera à la ligne 453 de son code source. Si au bout de ce temps aléatoire, un client n'est pas tué à cause de la probabilité, alors le scénario est répété. La grille est toujours constituée de 500 machines exécutant les clients et d'une machine exécutant le server.

Dans cet exemple, l'automate de la ligne 3, nommé *dem_ root*, est associé à la machine *Comp_ root* défini à la ligne 21 grâce à la ligne 23. Cet automate est vide car la machine *Comp_ root*, qui exécute le serveur, n'est soumise à aucune faute et l'automate n'est pas nécessaire au déroulement de l'automate correspondant aux autres machines.

```
1   spyfunc execute;
2
3   Daemon dem_root
4   { }
5
6   Daemon dem_all
7   {  ln ligne_test = calculate.c :453;
8      node 1 :
9           always int rand = FAIL_RANDOM (1, 10);
10          always time_g timer = FAIL_RANDOM (100, 200);
11          timer && rand <= 5 -> goto 2;
12          timer && rand > 5 -> goto 1;
13     node 2 :
14          before(execute) -> stop, goto 3;
15          ligne_test -> stop, goto 3;
16     node 3 :
17          time_g timer = 20;
18          timer -> continue, goto 1;
19  }
20
21  Computer Comp_root
22  {  program = "server -p 3420 -i console"
23     daemon = dem_root; }
24
25  Group All_comp
26  {  program = "client -p 3420 -i console"
27     size = 500;
28     daemon = dem_all; }
```

FIG. 4.35 – Deuxième scénario de fautes

La ligne 25 correspond à la définition du groupe de machines, nommé *All_ comp*, dont chaque machine exécute un client. L'automate *dem_ all* définit à la ligne 6 est associé à ce groupe de machines grâce à la ligne 28. La taille de ce groupe est de 500 (ligne 27) comme l'exige le test.

L'automate *dem_ all* est constitué de trois états : node 1, node 2 et node 3. Le premier état correspond à l'etat où le processus client est en cours d'exécution. Une variable aléatoire, *rand*, est défini à la ligne 9. Un minuteur, timer, est défini à la ligne 10 et sa valeur, comprise entre 100 et 200, est déterminée aléatoirement grâce à la fonction *FAIL_ RANDOM(100,200)*. Lors de l'expiration de ce minuteur et suivant la valeur de *rand*, soit l'automate se positionne dans l'état 2 (ligne 11), soit l'automate se repositionne dans le même état (ligne 12), dans ce deuxième cas, le même processus est relancé.

Lorsque l'automate se positionne dans l'état 2 (appel au *goto 2* de la ligne 11), il se place alors en attente d'un événement provenant du client local de l'application testée. Ces événements sont l'appel à la fonction *execute* (ligne 14) ou l'arrivée à la ligne 453 du code source du client (ligne 15). Dans les deux cas, l'exécution du client est momentanément interrompue (appel à la commande *stop*), puis l'automate se place dans l'état 3 (appel à *goto 3*).

Lorsque l'automate se place dans l'état 3, un minuteur de 20 secondes est défini (ligne 17). Quand ce minuteur expire (ligne 18), l'exécution du client est reprise et l'automate se positionne dans l'état 1 pour un nouveau cycle.

4.4 Conclusion du chapitre

Dans ce chapitre, nous avons présenté notre langage de spécification de scénarios de fautes, qui est fondé sur la notion d'automates communiquants et d'automates temporisés introduit dans le chapitre 3. Un scénario de fautes est ainsi spécifié par l'écriture d'un automate communiquant temporisé à l'aide du langage. En effet, un automate décrit en FAIL (Fault Injection Language) change d'état suite à la réception d'un message provenant d'un autre automate, comme dans le modèle des automates communiquants, ou suite à l'expiration d'un minuteur, comme dans le modèle des automates temporisés.

Cependant, puisque généralement un utilisateur peut désirer qu'un scénario de fautes prenne en compte le comportement et l'état de l'application qui sera instrumentée, un automate peut également changer d'état lorsqu'un événement particulier provenant de celle-ci se produit. Ce type d'événement est traité comme la réception d'un message ou l'expiration d'un minuteur. Ainsi, grâce à l'utilisation de variables aléatoires pouvant être affectées aux minuteurs, les fautes peuvent être injectées suivant un scénario de fautes probabiliste, mais également déterministe, grâce à l'utilisation des événements provenant de l'application instrumentée.

Pour plus de précision sur la syntaxe du langage FAIL, l'annexe A présente la gram-

maire sous forme BNF (Backus-Naur Form) correspondante.

Dans le chapitre suivant, nous allons présenter notre implantation d'une plateforme pour l'injection de fautes dans les systèmes distribués que nous avons nommé FCI (Fail Cluster Implementation). Cette plateforme est construite à partir du scénario spécifié dans le langage FAIL.

Chapitre 5

Implantation d'une Plate-forme pour l'Injection de Fautes

5.1 Introduction

Dans ce chapitre, nous allons présenter notre plate-forme d'injection de fautes FCI. Cette plate-forme est construite à partir du scénario de fautes écrit dans le langage présenté dans le chapitre précédent. En effet, un compilateur prend en entrée ce scénario et génère du code C++ qui sera combiné à une bibliothèque pour former des automates qui constitueront la plate-forme. Ces automates seront compilés sur les différentes machines participantes et exécutés sur celles-ci. Le comportement de ces automates suivra alors le scénario de fautes qui a été spécifié.

Dans la section 5.2, nous allons introduire le principe général de fonctionnement de la plate-forme. Dans la section 5.3, nous allons détailler la plate-forme FCI : la composition de la biliothèque FCI, le principe de la génération de code à partir du scénario de fautes écrit en FAIL et l'exécution d'une expérience de test.

5.2 Principe de fonctionnement

Notre solution est contituée de deux principaux composants :

– Le premier composant est FAIL (pour FAult Injection Language) qui est un langage qui permet de décrire « facilement » des scénarios de fautes (voir chapitre précédent). Le langage FAIL est un langage abstrait de haut niveau permettant de définir des scénarios de fautes grâce à des machines à états qui modélisent les occurences de fautes. Il décrit également l'association entre ces machines à états et

un ordinateur (ou un groupe d'ordinateur) du réseau.

- Le second composant est FCI (pour FAIL Cluster Implementation) qui est une plate-forme distribuée d'injection de fautes configurée à partir d'un scénario de fautes décrit dans le langage FAIL.

Ces deux composants sont développés dans le cadre du projet GriD eXplorer qui a pour but l'émulation de réseaux à grande échelle sur des clusters ou des grilles plus petites.

Décomposition de la plate-forme FCI. La plate-forme FCI (voir figure 5.1) est décomposée de la manière suivante :

- *Le compilateur de FCI :* Le scénario écrit en FAIL doit être enregistré dans un fichier portant l'extention « fl ». Il doit alors être précompilé par le compilateur de FCI qui génère alors du code C++ ainsi que des fichiers de configuration par défaut. Un fichier « .cpp » et un fichier « .h » sont générés pour chaque automate défini dans le scénario. Ces fichiers contiennent la définition d'une classe correspondant à chaque automate. Un fichier « .cpp » et un fichier « .h » sont générés pour chaque déclaration de machine et de groupe. Ces fichiers contiennent la définition d'une classe correspondant à l'injecteur de fautes.
- *La bibliothèque de FCI :* Toutes les classes générées lors de la compilation du scénario de fautes dérivent de classes principales contenues dans la bibliothèque FAILD. Cette bibliothèque contient également les classes modules qui sont des outils fournissant chacune des fonctions du langage FAIL.

Les fichiers générés et la bibliothèque FAILD sont alors copiés sur les différentes machines du réseau suivant les fichiers de configuration définis par l'utilisateur et ensuite compilés localement pour générer un programme exécutable « faild » (le code source des fichiers générés par le compilateur et de la bibliothèque FCI est compilé sur chaque machine pour permettre le support des clusters hétérogènes). C'est l'exécution de ce programme sur chacune des machines qui créera les démons FCI.

Les démons FCI. Quand l'expérience débute, l'application distribuée testée est exécutée à travers les démons FCI installés sur tous les ordinateurs pour leur permettre de l'instrumenter et de la contrôler suivant le scénario de fautes défini. Le démon FCI associé à un ordinateur particulier consiste en :

1. une machine à états implantant le scénario de fautes,
2. un module pour communiquer avec les autres démons (i.e. pour injecter des fautes basées sur l'état global du système),

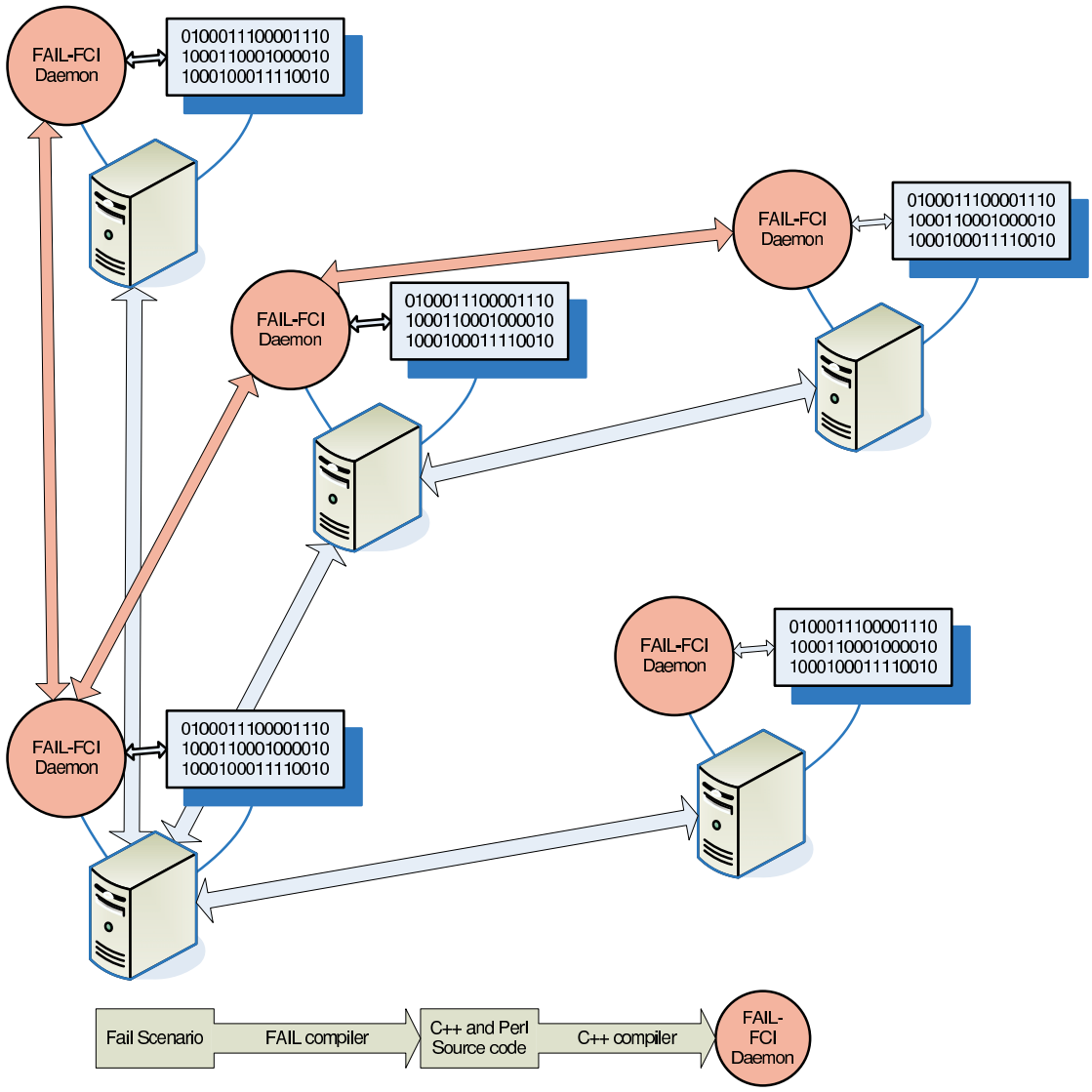

FIG. 5.1 – Structure de la plate-forme FCI

3. un module pour gérer le temps (i.e. pour permettre l'injection de fautes à un instant donné),
4. un module pour l'instrumentation de l'application testée (en contrôlant un débogueur),
5. et un module pour gérer les événements (pour déclencher les fautes).

L'utilisation d'un débogueur. FCI est un injecteur de fautes de type « Debugger-based » car l'injection de fautes et l'instrumentation de l'application testée sont faites grâce à l'utilisation d'un débogueur logiciel. Ceci permet de ne pas avoir à modifier le code source de l'application testée tout en ayant la possibilité d'injecter des fautes arbitraires (modification du « program counter » ou des variables locales pour simuler une attaque de type « buffer overflow », etc). Comme le débogueur parallèle Mantis [54], FCI communique avec le débogueur à travers des tubes unix. Mais contrairement à Mantis, les communications avec le débogueur doivent être minimisées pour permettre un faible surcoût de la plate-forme d'injection de fautes (dans notre approche, le débogueur est utilisé uniquement pour déclencher et injecter les fautes). L'application testée peut ainsi être interrompue quand elle appelle une fonction particulière ou avant d'exécuter une ligne particulière de son code source. Son exécution peut être reprise suivant le scénario de fautes considéré. L'implantation actuelle de FCI intègre le support de deux débogueurs différents : *gdb*, l'environment de debogage séquenciel de la fondation Free Software pour la prise en charge de programmes natifs et *jdb*, le débogueur Java pour la prise en charge de programmes Java.

Comparatif avec les autres approches. Du point de vue de l'utilisateur, il est suffisant d'écrire un scénario de fautes FAIL pour définir une expérience. Le code source des démons d'injection de fautes est automatiquement généré. Ces démons communiquent entre eux explicitement suivant le scénario de fautes défini par l'utilisateur. Ceci permet l'injection de fautes basée sur l'état global du système ou sur un mécanisme plus complexe impliquant plusieurs machines (i.e. une injection de fautes en cascade).

De plus, l'architecture complètement distribuée des démons FCI facilite le passage à l'échelle de l'outil, ce qui est nécessaire dans le contexte de l'émulation de systèmes distribués à grande échelle.

L'utilisation d'un débogueur pour le déclenchement des fautes permet également de limiter l'intrusion de l'injecteur de fautes pendant l'expérience. En effet, le débogueur place des points d'arrêt correspondant au scénario de fautes défini par l'utilisateur et exécute alors l'application testée. Tant qu'aucun point d'arrêt n'est atteint, l'application s'exécute normalement et le débogueur reste inactif.

Critères	Orchestra	NFTAPE	LOKI	FAIL-FCI
Grande expressivité	non	oui	non	oui
Langage de haut niveau	non	non	oui	oui
Pas de modification du code source	oui	non	non	oui
Passage à l'échelle	oui	non	oui	oui
Scénario probabiliste	oui	oui	non	oui
Injection selon un état global	non	oui	oui	oui

Fig. 5.2 – Comparatif des systèmes d'injection de fautes

Les différences essentielles entre notre approche et les travaux précédents sont résumées dans la figure 5.2.

5.3 La plate-forme FCI

La plate-forme FCI est notre implantation d'une plate-forme d'injection de fautes qui utilise FAIL comme langage de description de scénarios de fautes. Dans cette section, nous allons décrire l'implantation de la bibliothèque d'exécution, la génération de code, le déploiement de la plate-forme et enfin, nous allons étudier les tests de surcoût réalisés afin de déterminer l'impact de FCI sur l'application testée lors d'une expérience.

5.3.1 La bibliothèque FCI

La bibliothèque FCI fournit au code généré lors de la compilation du scénario de fautes des briques de base nécessaires pour l'exécution du démon FCI. Cette bibliothèque est constituée d'un ensemble de classes C++ utilisant la bibliothèque standard C++ et la bibliothèque ACE (Adaptative Communication Environment) [69, 70], qui est disponible pour un grand nombre de systèmes d'exploitation (incluant MS Windows et différents systèmes Unix).

La bibliothèque ACE. ACE (Adaptive Communication Environment) est un *framework* orienté objet libre et open-source qui implante plusieurs motifs pour des logiciels utilisant des communications concurrentes. ACE fournit un ensemble d'adaptateurs C++ réutilisables et des composants « framework » qui effectuent les tâches courantes de communications pour plusieurs systèmes d'exploitation différents. Les tâches de communication fournies par ACE incluent le démultiplexage d'événements, le dispatche d'événe-

ments vers les gestionnaires, la gestion des signaux, l'initialisation de services, l'exécution concurrente et la synchronisation. ACE vise à être utilisé par les développeurs d'applications haute-performance à communication temps réel. Il simplifie le développement d'applications réseau orientées objet qui utilisent des communications inter-processus, du démultiplexage d'événements et de la concurrence. ACE est soutenu commercialement par plusieurs compagnies utilisant un modèle de fonctionnement open-source.

Il existe plusieurs avantages à utiliser ACE :

- Améliorer la portabilité : les composant ACE facilitent l'écriture d'applications réseau concurrentes sur un certain système d'exploitation et permettent également de les porter rapidement sur d'autres systèmes d'exploitation.
- Améliorer la qualité d'une application : les composants de ACE utilisent plusieurs motifs clés qui améliore la flexibilité, l'extensibilité, la réutilisabilité et la modularité d'une application.
- Améliore l'efficacité et la prévisibilité : ACE est conçu pour fournir un grand nombre de qualités de service incluant une faible latence pour les applications sensibles à la latence, de hautes performances pour les applications sensibles à la bande passante et une bonne prévisibilité pour les applications en temps réel.

Les adaptateurs C++ simplifient le développement de l'application en fournissant des interfaces C++ qui encapsulent la concurrence, les communications, la gestion de la mémoire et le démultiplexage d'événements. Les applications peuvent ainsi combiner et composer ces adaptateurs en héritant, en agrégeant, ou en instanciant les composants suivants :

- Les composants de concurrence et de synchronisation : ACE abstrait les mécanismes natifs de multi-threading et de multi-processing du système d'exploitation pour créer des abstractions orientées objet de plus haut niveau.
- Les composants d'IPC et de système de fichiers : les adaptateurs C++ de ACE encapsulent les mécanismes d'IPC locaux et/ou distants, comme les sockets, les TLIs, les FIFOs UNIX, les tubes et les tubes nommés Win32. En plus, les adaptateurs C++ de ACE encapsulent les APIs du système de fichiers du système d'exploitation.
- Les composant de gestion de la mémoire : Les composants de gestion de la mémoire de ACE fournissent une abstraction flexible et extensible pour la gestion de l'allocation et de la désallocation dynamique de la mémoire partagée entre processus.

Les adaptateurs C++ fournissent ainsi plusieurs fonctionnalités qui sont structurées en classes C++ plutôt qu'en fonction C. L'utilisation du C++ améliore la robustesse car

les adaptateurs C++ sont fortement typés, les compilateurs peuvent ainsi détecter les erreurs de typage à la compilation plustôt que lors de l'exécution.

ACE contient également des composants « framework » de programmation réseau de plus haut niveau qui intègrent et améliorent les adaptateurs C++ de bas niveau. Deux de ces composants sont le `Reactor` et le `Proactor` qui sont des composants de démultiplexage d'événements. Ce sont des démultiplexeurs extensibles qui dispatchent les gestionnaires spécifiques de l'application en réponse à différents types d'événements tels que les I/Os, les timers, les signaux, etc.

Décomposition de la bibliothèque FCI. Les classes de la bibliothèque FCI peuvent être partitionnées en trois grandes catégories :

- **Gestion d'événements :** La bibliothèque FCI utilise la bibliothèque ACE pour la gestion des événements. Elle fournit un mécanisme pour le multiplexage d'événements. Une classe ACE appelée `Reactor` [68] est utilisée pour attendre l'occurence d'un événement particulier. Quand un objet est intéressé par un événement particulier (i.e. temps, connection reseau, etc.), il s'enregistre auprès de l'objet `Reactor` approprié qui commence alors à attendre cet événement. Quand un des événements attendus se produit, le `Reactor` active l'objet qui était intéressé par celui-ci. Ce mécanisme permet d'attendre plusieurs événements de types différents avec un impact minimum sur le fonctionnement de l'application testée. En effet, le mécanisme d'attente d'événements du `Reactor` est passif.
- **Contrôle de l'application :** La bibliothèque FCI utilise un débogueur logiciel (dans l'implantation actuelle : `gdb` et `jdb`) pour contrôler l'application testée. Ce mécanisme permet d'observer et de contrôler le comportement de l'application sans avoir à modifier son code source et avec une faible intrusion[1]. Il n'est donc pas nécessaire de disposer du code source de l'application testée. Pour avoir accès à toutes les fonctionnalités, l'application doit nécessairement être compilée en mode « debug ». Cependant, si uniquement une version « release » de l'application est disponible, des scénarios n'utilisant pas de connaissances liées au code source (i.e. utilisation de timer, messages, etc.) peuvent être implantés. De plus, `gdb` est compatible avec de nombreux langages de programmation différents, ce qui permet l'utilisation de notre outil pour de nombreuses applications existantes. Les applications écrites en

[1] L'application testée devrait cependant être écrite pour être « debugger friendly », *i.e.* reprendre correctement sont exécution aprés des opérations bloquantes suite à une interruption (les débogueurs utilisent généralement les signaux pour contrôler l'exécution du processus à déboguer).

Java sont également supportées grâce à l'utilisation de `jdb`.

- **Communication entre démons :** Les démons FCI constituent eux-même une application distribuée. Nous utilisons également la bibliothèque ACE pour la gestion des connexions et des communications entre les machines distantes.

Nous avons implanté une grande partie de la plate-forme directement dans la bibliothèque d'exécution afin de faciliter l'étape de génération de code et améliorer la robustesse de l'outil grâce à des tests unitaires qui ont pu ainsi être effectués. Nous allons maintenant étudier de façon plus détaillée cette bibliothèque.

Architecture du noyau de la blibliothèque FCI. Nous avons structuré le noyau de la bibliothèque pour le rendre modulaire, extensible, et pour faciliter le contrôle d'applications écrites dans des langages de programmation qui génèrent ou pas de code natif. La structure interne de la bibliothèque FCI peut être divisée en trois parties principales : le noyau, l'interface externe et la structure réseau.

L'architecture du noyau est décrite dans la figure 5.3. La classe principale `Environment` contrôle la classe `Trigger` et la classe `StateMachine`. La classe `Trigger` modélise les événements qui peuvent apparaître pendant l'exécution de l'application distribuée (un point d'arrêt atteint dans le programme testé, un message reçu provenant d'une autre machine, un timer écoulé etc.). La classe `StateMachine` représente l'automate qui décrit les actions qui seront effectuées par le démon FCI quand certains événements se produisent.

Lorsqu'un démon FCI est exécuté, tout d'abord, les différentes interfaces sont exécutées pour la prise en charge des événements, la prise en charge du réseau, le contrôle de l'application testée, la prise en charge des exécutions à distance de processus de l'application testée (pour les applications auto-déployantes). L'environnement permet la création des nœuds (méthode `InsertNode`) et leur affectation dans des groupes (méthode `InsertNodeInGroup`). La machine à états est également initialisée par l'environnement grâce à la méthode `InitStateMachine`. Dès lors, tous les modules du démon sont exécutés et il se place alors dans une boucle infinie d'attente d'événements grâce à la méthode `handle_events` du `reactor` de la bibliothèque ACE (non représenté sur le schéma).

Pendant l'exécution, l'environnement sert de lien entre la machine à états (le motif `SateMachine`) et les événements (le motif `Observer`). Lorsque la machine à états est initialisée, elle entre dans son premier état (qui correspond à `State1` sur le schéma). L'état est alors chargé : la méthode `RegisterAllRules` est exécutée, elle définit les événements nécessaires en utilisant les différentes interfaces externes (qui seront détaillées dans le

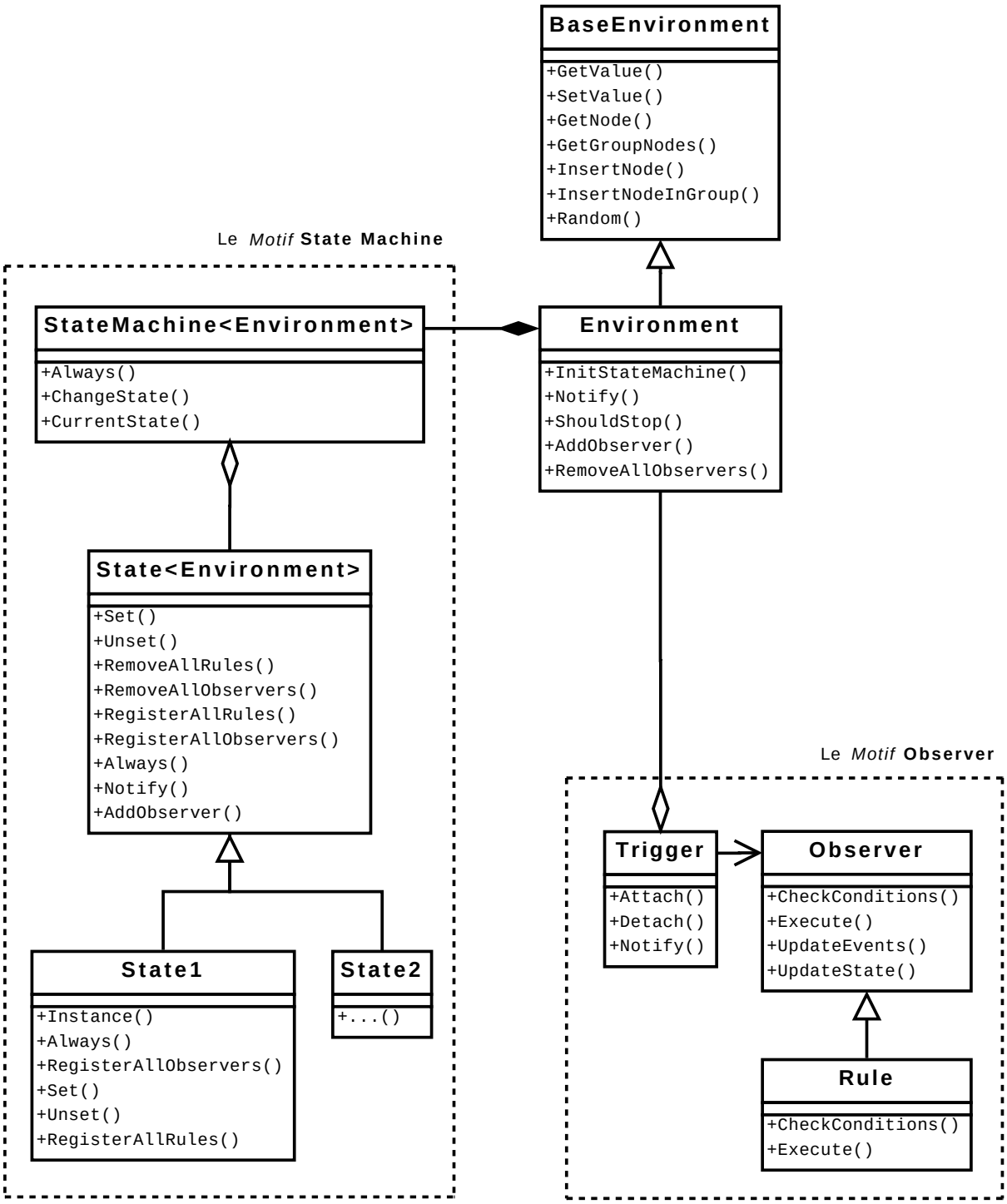

FIG. 5.3 – Architecture du noyau de la blibliothèque FCI

paragraphe suivant) et les attaches à chaque règle en utilisant la méthode `AddObserver` de l'environnement. Ainsi lorsque un événement est déclenché par une des interfaces externes, celle-ci alerte l'environnement en exécutant sa méthode `Notify`. Lorsque l'environnement est alerté de l'arrivée d'un événement, il déclenche le `trigger` correspondant à cet événement en exécutant la méthode `Notify` de celui-ci. Le `trigger` recherche alors dans la liste des `Observer` l'un d'entre eux qui est attaché à son événement et le met à jour en exécutant sa méthode `UpdateEvents`. Cette méthode permet l'exécution de la méthode `Execute` suivant le résultat de l'exécution de la méthode `CheckConditions`. En effet, la méthode `CheckConditions` permet d'effectuer différents tests (égalité sur des entiers, etc) après avoir reçu un événement pour déterminer si les actions de la règle doivent être exécutées ou non. Tous les `Observer` attachés à l'événement seront mis à jour jusqu'à en trouver un dont les conditions soient valides (i.e sa méthode `CheckConditions` renvoie vrai). Ainsi, deux cas sont possibles :

- il existe un `Observer` dont les conditions sont valides. Les actions correspondantes sont donc exécutées et l'automate change d'état grâce à la méthode `ChangeState` de la classe `StateMachine`. Les nouveaux événements correspondants au nouvel état sont alors chargés et un nouveau cycle commence ;
- il n'existe aucun `Observer` dont les conditions sont valides. Aucune exécution d'actions n'est alors faite et l'automate ne change pas d'état. Mais les événements de l'état courant sont rechargés et un nouveau cycle commence.

L'interface externe de la bibliothèque FCI. Les actions possibles déclenchées par l'`Environment` sont contrôlées par quatre classes différentes (voir figure 5.4). La classe `TimerController` permet de configurer des événements de timeout, La classe `NetworkController` permet de gérer les communications entre les démons FCI, la classe `ProgramController` permet de contrôler le programme testé à travers un débogueur, et la classe `LoaderController` permet le support des exécutions à distance des processus de l'application testée (i.e applications auto-déployantes). La classe `ProgramControllerGDB` permet la gestion des programmes natifs à travers le débogueur `gdb`, tandis que la classe `ProgramControllerJDB` permet la gestion des programmes Java à travers le débogueur `jdb`. D'autres extensions pour la prise en charge d'autre langages et d'autres systèmes peuvent ainsi être implantées grâce à ces classes d'interface.

Le contrôleur réseau (la classe `NetworkController`) permet notamment l'envoi d'un message à un nœud FCI distant grâce à la méthode `SendMessage2Node`, l'envoi d'un message à un groupe de nœuds FCI grâce à la méthode `SendMessage2Group` et l'envoi d'un message à un noeud pariculier d'un groupe grâce à la méthode `SendMessage2OneOfGroup`.

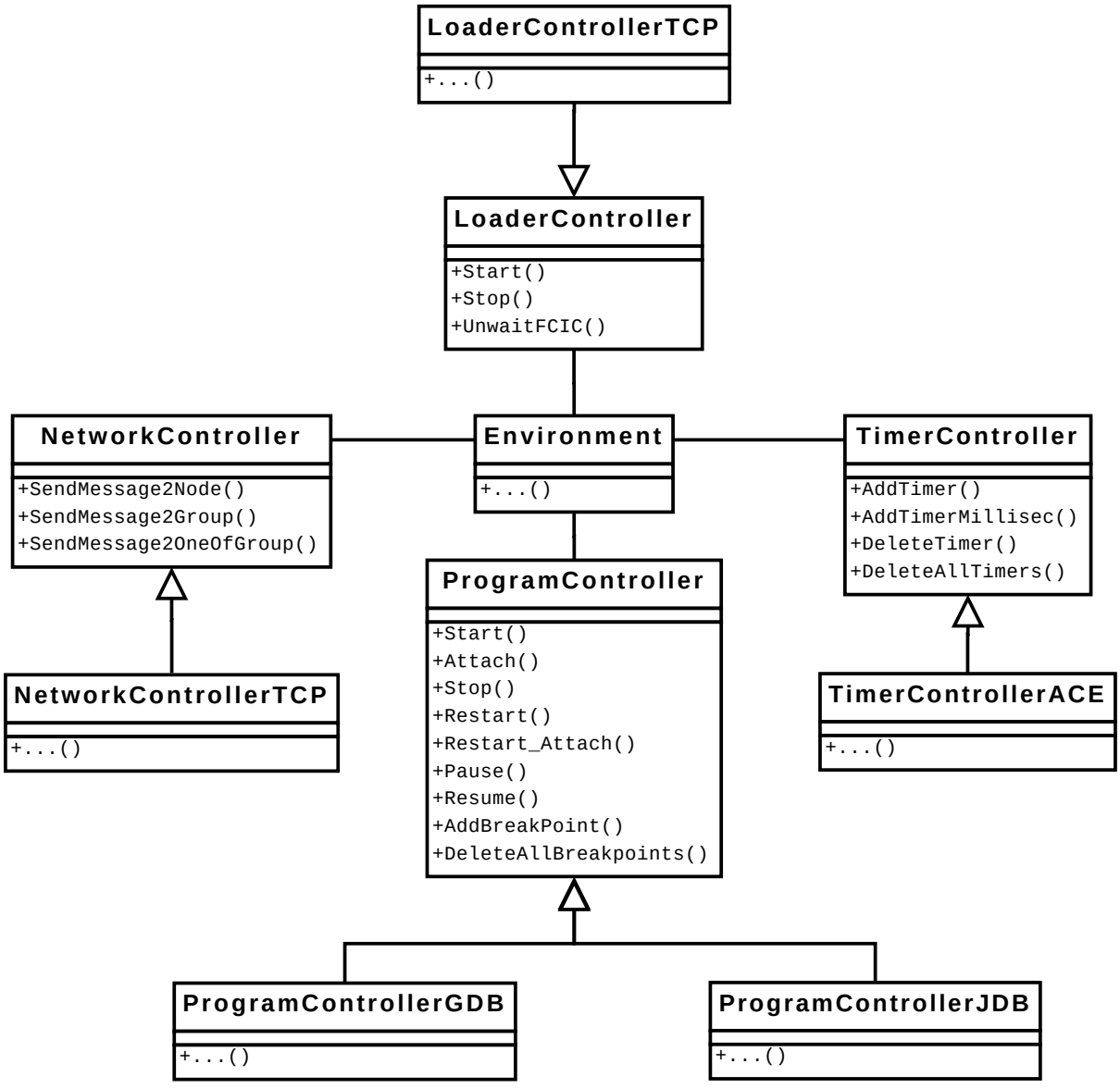

FIG. 5.4 – L'interface externe

Le contrôleur de programme (la classe `ProgramController`) permet notamment de contrôler l'application testée et d'injecter les fautes : arrêt (méthode `Pause`) puis reprise (méthode `Start`), relancement (méthode `Restart`) et arrêt définitif (méthode `Stop`) de l'application testée. Le contrôleur de programme peut exécuter lui même un processus de l'application testée ou s'attacher à un processus existant grâce à la méthode `Attach`, il peut alors définir des points d'arrêt sur un appel à une fonction spécifique ou sur l'arrivée à une ligne spécifique du code source par l'application testée grâce à la méthode `AddBreakPoint`.

Le contrôleur de temps (la classe `TimerContrôller`) permet de définir des minuteurs de l'ordre de la seconde ou de la milliseconde grâce aux méthodes `AddTimer` et `AddTimerMillisec` respectivement.

Le contrôleur de lancement (la classe `LoaderController`) permet au démon FCI d'exécuter une commande qu'il a reçue du client *fcic* pour créer un processus de l'application testée et le contrôler. En effet, lorsque l'application testée désire exécuter un processus à distance, elle ne doit pas le faire directement, mais doit exécuter sur la machine distante un client *fcic* en lui passant en paramêtre la commande à exécuter pour créer ce processus. Le processus de l'application ne sera donc pas directement exécuté, mais c'est le client *fcic* qui communiquera au contrôleur de lancement la commande à exécuter. Après avoir communiqué la commande au contrôleur de lancement, le client s'interrompt. Cependant, pour l'application testée, il peut être nécessaire de détecter la terminaison du client lorsque le processus de l'application est effectivement en cours d'exécution. La méthode `UnwaitFcic` du contrôleur de lancement permet ainsi d'alerter le client *fcic* que le processus a été lancé et qu'il peut donc se terminer. Le client *fcic* a donc deux modes de fonctionnement, l'un où il se termine après l'envoi de la commande et l'autre où il se termine après la réception de l'alerte provenant du contrôleur de lancement.

La gestion des événements est présentée dans la figure 5.5. La gestion des événements est fondée sur l'utilisation du `Reactor` de la bibliothèque ACE. Les différentes interfaces enregistrent auprès du `Reactor` les événements qu'il doit espionner. Lorsqu'un événement se produit, une fonction de « callback » est automatiquement exécutée par le `Reactor` pour effectuer les actions adéquates. Cette fonction de « callback » est `handle_timeout` pour le contrôleur de temps et `handle_input` pour les autres contrôleurs. Ce mécanisme de gestion centralisée des événements permet de rendre plus robuste l'implantation en évitant des problèmes liés aux mécanismes d'éxécutions concurrentes. Voici la description du fonctionnement pour les différentes interfaces :

– Le contrôleur de temps enregistre le minuteur auprès du `Reactor` grâce à la méthode

`schedule_timer` de celui-ci. La méthode `handle_timeout` du gestionnaire de temps (la classe `TimerACEHandler`) sera alors exécutée dès l'expiration de ce minuteur (correspondant à l'événement « expiration d'un minuteur »).

- Le gestionnaire de flot GDB (la classe `GDBStreamHandler`) est enregistré par le contrôleur de programme GDB au `Reactor` de ACE grâce à la méthode `register_handler` du `Reactor` (pour le cas de l'utilisation de GDB pour les programmes natifs). Plus précisément, cette méthode permet d'enregistrer la sortie standard du programme testé avec le `Reactor` et de lui associer le gestionnaire de flot. Ainsi, lorsqu'un flot de caractères est détecté par le `Reactor`, la méthode `handle_input` de la classe `GDBStreamHandler` est invoquée et si dans ce flot de caractères un point d'arrêt est détecté (ce qui correspond à l'événement « appel d'une fonction particulière ou arrivée à une ligne particulière du code source par l'application testée »), alors la méthode `handle_bp_function` est invoquée.
- Le gestionnaire de connexion TCP (la classe `TCPServerAcceptorHandler`) est enregistré au `Reactor` par le contrôleur de réseau (la classe `NetworkControllerTCP`) lorsque celui est démarré grâce à sa méthode `Start`. Ainsi, lorsqu'un démon FCI distant établit une connexion, la méthode `handle_input` est automatiquement invoquée. Lorsque cette méthode est invoquée, le `TCPServerAcceptorHandler` enregistre le gestionnaire de flux TCP (la classe `TCPServerInputHandler`) au Reactor. Ainsi lorsque le message suivant la connexion (correspondant à l'événement « réception d'un message ») est délivré au démon FCI, la méthode `handle_input` du gestionnaire de flux TCP est automatiquement invoquée.
- Le fonctionnement du gestionnaire de lancement de processus distants (la classe `LoaderControllerTCPHandler`) est identique à celui gestionnaire de connexion TCP puisqu'il est également fondé sur la couche TCP/IP. Il est enregistré au `Reactor` par le contrôleur de lancement de processus distant (la classe `LoaderControllerTCP`) lorsque celui est démarré grâce à sa méthode `Start`. Ainsi, lorsqu'un démon FCI distant établit une connexion pour l'envoi d'une commande à exécuter, la méthode `handle_input` est automatiquement invoquée. Lorsque cette méthode est invoquée, le `LoaderControllerTCPHandler` enregistre le gestionnaire de flux TCP de commandes distantes (la classe `TCPLoaderInputHandler`) au `Reactor`. Ainsi lorsque le message suivant la connexion (correspondant à l'événement « exécution d'un nouveau processus par l'application testée ») est délivré au démon FCI, la méthode `handle_input` du gestionnaire de flux TCP de commandes distantes est automatiquement invoquée.

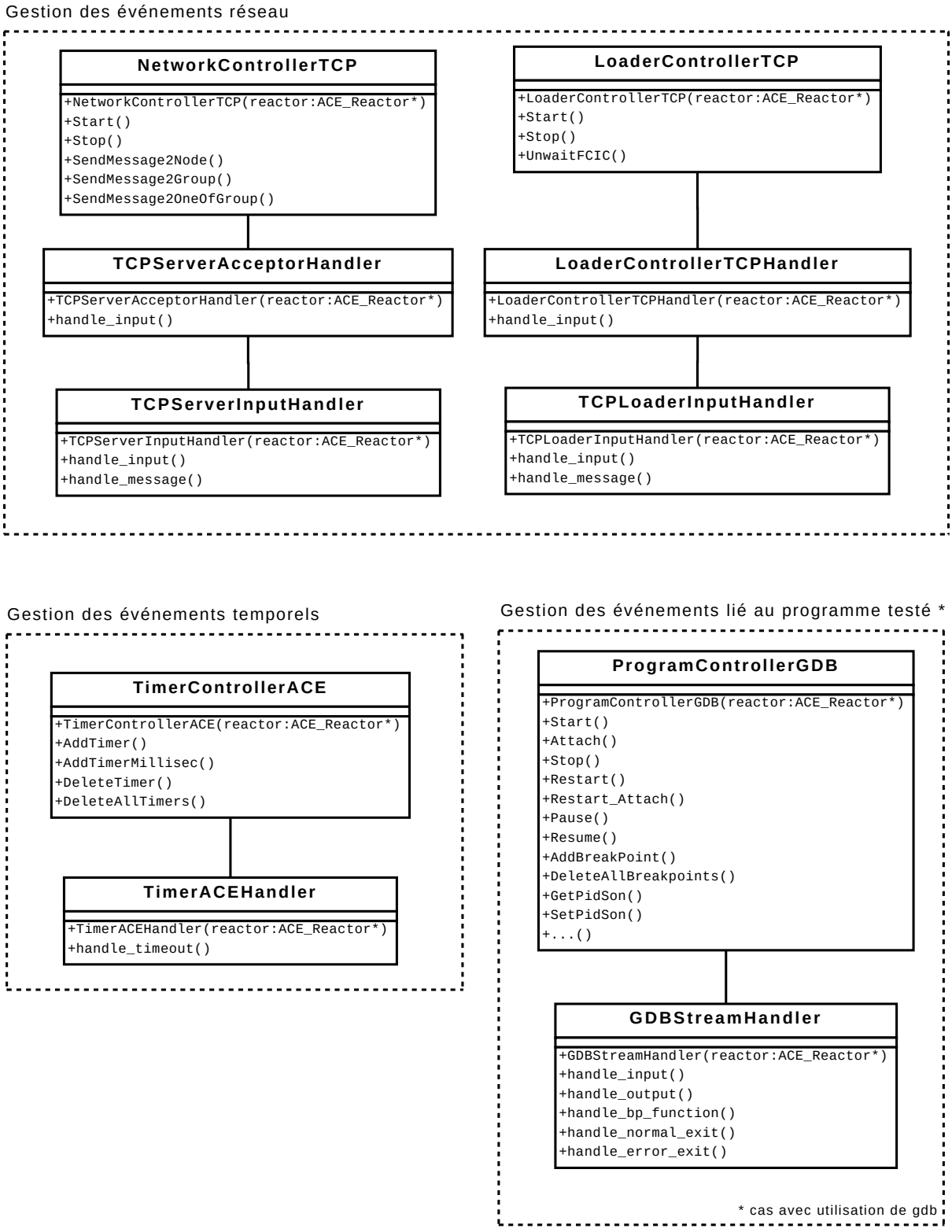

FIG. 5.5 – La gestion des événements

La structure réseau de la blibliothèque FCI. La structure du réseau logique induit par les différents noeuds FCI est générée par le compilateur FAIL et réside en un fichier XML comme celui de la figure 5.6. Dans ce fichier, chaque nœud FCI est défini par une balise <*computer*> et les nœuds sont associés à des groupes grâce à la balise <*group*>. Dans cet exemple, trois nœuds sont définis et indexés de 1 à 3, le premier nœud est associé au démon P1 et les autres nœuds au groupe G1 (l'association des nœuds aux démons est faite selon l'ordre d'apparition des déclarations dans le fichier XML, la balise <*!– computer P1 –*> correspond à un commentaire qui a été inséré pour faciliter la compréhention).

```
<fail>
    <network>
        <computer> <!– computer P1 –>
            <index>1</index>
            <ip>192.168.0.1 :5000</ip>
        </computer>
        <computer>
            <index>2</index>
            <ip>192.168.0.2 :5000</ip>
        </computer>
        <computer>
            <index>3</index>
            <ip>192.168.0.3 :5000</ip>
        </computer>
    </network>
    <groups>
        <group name="G1">
            <index>2</index>
            <index>3</index>
        </group>
    </groups>
</fail>
```

FIG. 5.6 – Exemple d'un fichier network.xml

La structure réseau est implantée par la classe `Node` pour la gestion des nœuds et

par la classe `GroupNodes` pour la gestion des groupes de nœuds comme le montre la figure 5.7. L'automate qui contrôle le composant distribué de l'application peut avoir à manipuler des variables entières, réelles ou booléennes. Afin de contrôler l'initialisation de ces variables, la hiérarchie de classe `Value` a été ajoutée.

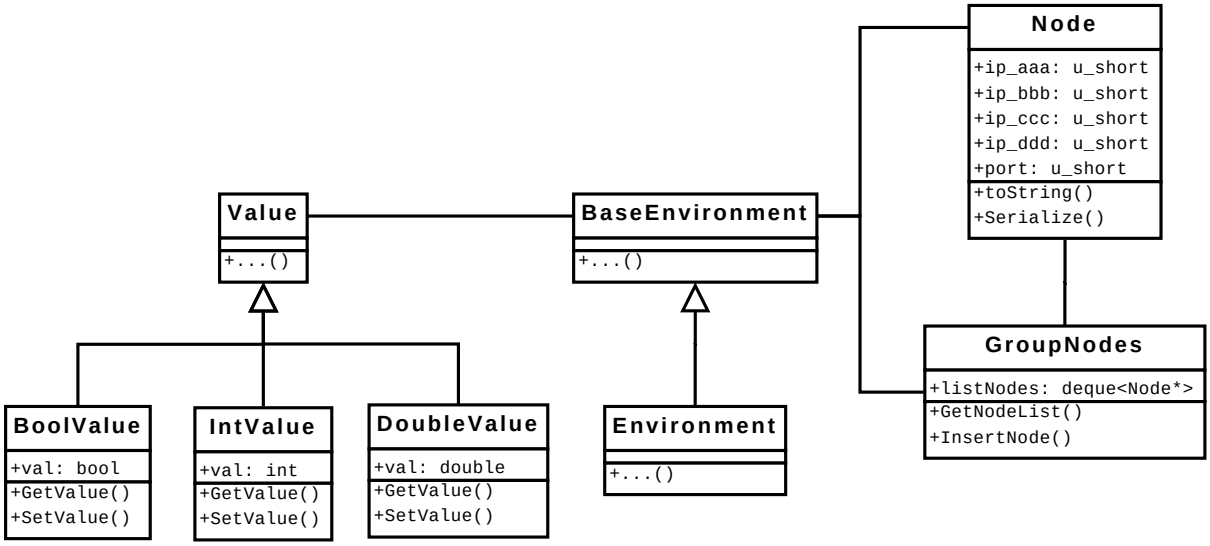

FIG. 5.7 – La structure réseau

5.3.2 Le compilateur de FCI

Le compilateur de FCI prend en entrée un scénario ecrit en FAIL et génère du code C++. Il a été ecrit en *Ocaml*. La première étape, l'analyse lexicale, est effectuée grâce à l'outil *ocamllex* qui est la version Ocaml de l'outil Unix standard *lex*. Cet outil construit automatiquement un analyseur lexical : un automate déterministe à actions prenant en entrée un flot de caractères et retounant une suite de « tokens ». La deuxième étape, l'analyse syntaxique, est effectuée grâce l'outil *ocamlyacc* qui est la version Ocaml de l'outil Unix standard *yacc*. Cet outil permet la création d'un analyseur syntaxique prenant en entrée une suite de « tokens » qui lui est fourni par l'analyseur lexical. Il reconnait si l'entrée appartient au langage FAIL et génère un arbre de syntaxe abstraite. Cet arbre représente la valeur sémantique du scénario FAIL et sera utilisé pour l'analyse sémantique et la génération de code. L'analyse sémantique consiste à analyser la structure du scénario. Lors de cette analyse, on vérifie si le scénario est structuré avec différents « node » ou pas et si les « goto » pointent effectivement vers des « node » existants. Puis, on vérifie la portée des variables et enfin on vérifier si leurs types sont cohérents. Une fois les différentes vérifications effectuées, le compilateur effectue la dernière étape qui consiste à générer le code C++.

La génération de code. Le compilateur FAIL permet de générer le code C++ correspondant au scénario de fautes écrit en FAIL. Les classes générées lors de cette compilation sont présentées dans la figure 5.8. Les classes générées ne sont autre que les classes correspondant à la machine à états : les classes correspondant aux états (`ADV1_state1` dans l'exemple) et les classes correspondant aux règles gardées (`ADV1_state1_rule0` dans l'exemple).

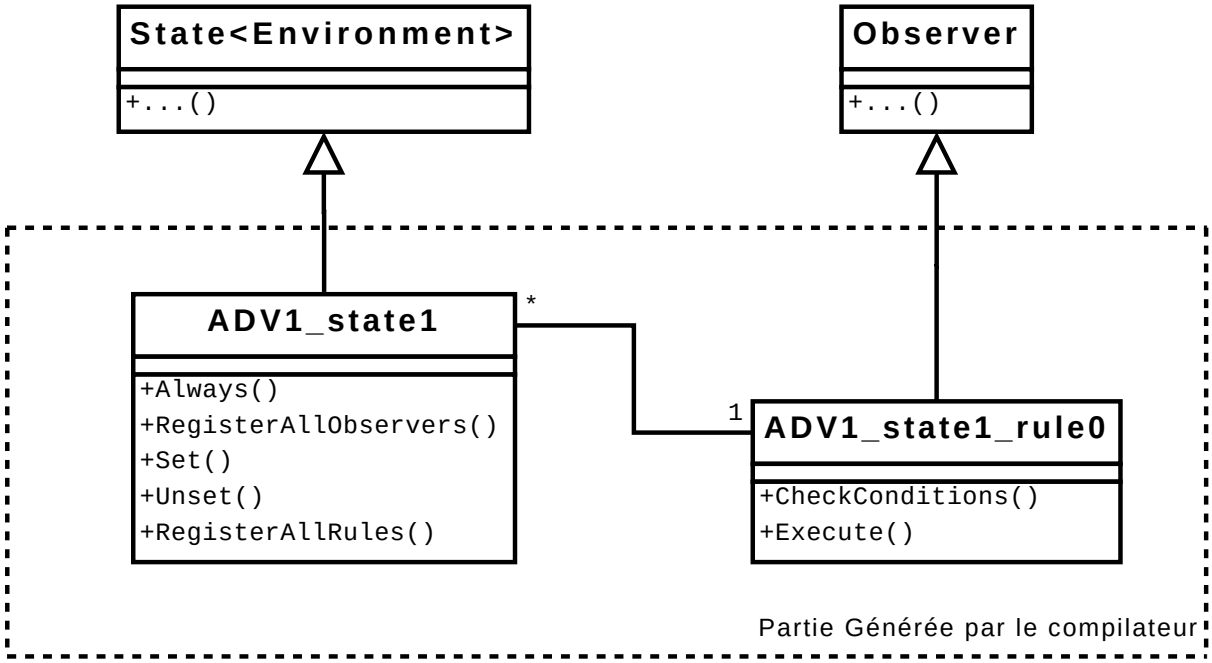

FIG. 5.8 – Le code généré par le compilateur

Comme nous l'avons déjà vu dans le paragraphe présentant l'architecture du noyau de la blibliothèque FCI, lorsqu'un état est chargé, la méthode `RegisterAllRules` est exécutée. Cette méthode permet de définir les événements nécessaires en utilisant les différentes interfaces externes et en les attachant à chaque règle. Dans l'exemple de la figure 5.8, un événement particulier (non représenté sur le schéma) est associé à la règle `ADV1_state1_rule0` qui est une classe héritant de la classe `Observer`. Lorsque cet événement se produit, la méthode `CheckConditions` est exécutée et suivant le résultat, la méthode `Execute` sera invoquée ou non (cette méthode correspond aux actions de la règle).

La figure 5.9 présente un scénario de fautes très simple.

Ce scénario est écrit pour être utilisé sur 6 machines. Une machine particulière appelée *P1* exécutera le serveur de l'application testée et les 5 autres machines exécuteront les clients et seront regroupées dans le groupe appelé *G1*. Les machines de ce groupe seront soumises au scénario décrit par l'automate ADV2 qui est un automate « vide », ce qui

```
spyfunc main;

Daemon ADV1
{   node 1 : before(main) -> continue, goto 2;
    node 2 : time_g timer = 30;
             always int test = FAIL_RANDOM(1,10);
             timer && test <= 2 -> stop, goto 3;
    node 3 : time_g timer = 20;
             timer -> continue, goto 2;   }

Daemon ADV2
{   node 1 : before(main) -> continue, goto 2;
    node 2 :   }

Computer P1
{   program = "server";
    daemon = ADV1;   }

Group G1
{   program = "client";
    size = 5;
    daemon = ADV2;   }
```

FIG. 5.9 – Exemple d'un fichier de scénario de fautes « .fl »

signifie qu'elles ne seront soumises à aucune faute. Le serveur sera lui soumis au scénario de fautes décrit par l'automate ADV1 suivant : toutes les 30 secondes, l'exécution du serveur sera interrompue avec une probabilité de 0,2 et une fois interrompu, le server continuera son exécution là où il s'était interrompu après un délai de 20 secondes. Ce mécanisme est ainsi répété indéfiniment.

La figure 5.10 présente le fichier *ADV1_state2.cpp*, généré par le compilateur, correspondant à l'état 2 de l'automate ADV1 du scénario de fautes.

La figure 5.11 présente le fichier *ADV1_state2_rule0.cpp*, généré par le compilateur, correspondant à la première règle de l'état 2 de l'automate ADV1 du scénario de fautes.

5.3.3 L'exécution d'une expérience de test

La compilation d'un scénario de fautes FAIL (*i.e* fichier portant l'extention *.fl*), en utilisant le compilateur FCI, produit un répertoire dédié portant le même nom sans l'extention *.fl*. Ce répertoire contient les fichiers C++ générés correspondant au scénario, le code source de la bibliothèque d'exécution FCI et les fichiers de configuration du test.

La configuration, la compilation et le déploiement de démons (i.e l'exécution du test) sont réalisés grâce à l'outil *fcideploy*. Nous allons maintenant détailler chacune de ces étapes. Dans la suite de cette section, pour toutes les commandes exécutées, nous allons supposer que le prompteur de la console est positionné dans le répertoire de test généré par le compilateur FAIL.

La configuration est faite en exécutant la commande *fcideploy configure*. L'utilisateur doit au préalable vérifier/modifier le fichier de configuration *config/deploy.conf* selon ses besoins et fournir un fichier *config/machines* contenant le nom (ou l'IP) de toutes les machines participantes au test (un nom de machine par ligne).

La figure 5.12 présente le fichier de configuration *config/deploy.conf*.

Voici la signication des différentes déclarations :

- <**staticcomputer**> : ne doit pas être modifié.
- <**staticgroup**> : ne doit pas être modifié.
- <**computer**> : correspond à une déclaration « Computer » du scénario FAIL.
- <**group**> : correspond à une déclaration « Group » du scénario FAIL.
- <**other**> : cette déclaration est utile si des processus doivent être exécutés sur des machines sans passer par des démons FCI. Dans l'exemple, la déclaration *other* en commentaire sert à exécuter un processus sur la machine exécutant le démon FCI *P1* sans passer par ce démon.

Voici la signification des différentes balises xml utilsées dans les différentes déclarations :

- <**exec**> : cette balise définit le programme qui sera exécuté par le démon FCI (ou simplement exécuté pour une déclaration *other*).
- <**focus**> : cette fonction sert à attendre que la chaîne de caractères définie apparaisse sur la sortie standard du programme exécuté pour terminer l'expérience. Cette déclaration est donc primordiale. Il est obligatoire de définir au moins une balise *focus* car sinon l'expérience se termine dès son lancement !
- <**preexec**> : définit un programme qui sera exécuté directement avant l'exécution de l'<*exec*> sans passer par un démon FCI.

- <**postexec**> : définit un programme qui sera exécuté directement à la fin de l'exécution de l'<*exec*> sans passer par un démon FCI.
- <**wait**> : définit une attente en seconde avant l'exécution du démon FCI ou l'exécution directe d'un processus dans le cas d'une déclaration *other*.

Précisions sur la balise focus : Il est obligatoire de définir au moins une balise *focus* car sinon l'expérience se termine dès son lancement ! Lorsqu'un *focus* est utilisé sur un programme, l'arrêt de l'expérience dépend de ce programme et deux cas sont alors possibles :

- le programme imprime effectivement la chaîne de caractères attendue sur sa sortie standard, l'expérience est alors terminée.
- le programme se termine sans avoir imprimé la chaîne de caractères attendue sur sa sortie standard, l'expérience est tout de même terminée.

Cette balise sert donc à déterminer quel programme « espionner » afin de détecter l'arrêt d'une expérience. La possibilité de définir une chaîne de caractères à détecter est fournie pour prendre en compte les applications utilisant des serveurs « persistants ». Par exemple, si on considère le cas d'un dispatcher de tâches : lorsque l'ensemble des tâches est terminé, l'expérience doit être détectée comme étant terminée alors que le dispatcher est toujours en cours d'exécution en attente de nouvelles tâches. Pour ce type d'applications, l'utilisation de la chaîne de caractères pour la détection de l'arrêt d'une expérience est nécessaire.

Lors de la configuration (commande *fcideploy configure*), un fichier *config/network.xml* est généré (ce fichier a déjà été présenté dans la figure 5.6). Ce fichier fait la correspondance entre les machines réelles du réseau et les *Computer* et *Group* définis dans le scénario FAIL. L'utilisateur peut modifier ce fichier si nécessaire pour réorganiser le déploiement des démons avant de passer aux autres étapes.

La compilation. Il y a deux méthodes de compilation :

1. la méthode locale
2. la méthode globale

Pour la méthode locale, il faut lancer la commande suivante :
fcideploy compil
Cette commande compile tous les démons FCI dans le répertoire courant.

Pour la méthode globale, il faut lancer la commande suivante :
fcideploy globalcompil
Cette commande copie dans un premier temps les fichiers nécessaires vers toutes les machines définies dans le fichier *machines* et ensuite lance la compilation sur toutes ces machines en parallèle.

L'application de la configuration est nécessaire pour mettre à jour toutes les machines participantes, il y a deux cas possibles :

1. La compilation a été faite localement. Dans ce cas, il faut lancer la commande :
   fcideploy copyall
   Cette commande copie tout le contenu du répertoire de test vers toutes les machines.

2. La compilation a été faite globalement. Dans ce cas, il faut lancer la commande :
   fcideploy copyconfig
   Cette commande copie uniquement les fichiers de configuration vers toutes les machines.

Le lancement de l'expérience (le déploiement) est fait par l'utilisation la commande suivante :
fcideploy run

Cette commande permet l'exécution des démons FCI sur toutes les machines participantes, ce qui correspond au début du test. Une fois tous les démons initialisés, soit ils exécutent une commande (si une commande a été fournie par l'utilisateur), soit ils attendent la connection d'un client *fcic* pour obtenir une commande à exécuter (pour le cas des applications auto-déployantes). Il est conseillé d'utiliser un démon particulier, le *point d'entrée*, qui s'occupe d'attendre que tous les autres démons se soient bien initialisés avant de lancer réellement le test. Dans ce cas, les autres démons s'initialisent et se placent ensuite en attente d'un message du *point d'entrée* avant de continuer leur exécution. Le démon *point d'entrée* n'a pas besoin (mais peut) participer à l'expérience (i.e exécuter un processus de l'application testée). Ce mécanisme peut être écrit directement dans le scénario de fautes grâce au langage FAIL.

5.4 Conclusion du chapitre

Dans ce chapitre, nous avons présenté notre plate-forme d'injection de fautes. Cette plateforme, libre et open-source, est disponible sous licence CeCILL à l'adresse suivante : *http ://fail.gforge.inria.fr*.

Actuellement, uniquement une version pour Linux est disponible, mais une version Windows et une version Mac sont envisageables...

Nous avons d'ores et déjà effectué plusieurs tests grâce à notre plateforme sur différentes applications distribuées. Ces tests sont présentés dans la partie suivante.

```
#include "ADV1_state2.h"
#include "ADV1_state2_rule0.h"

ADV1_state2* ADV1_state2 : :Instance()
   { /*...*/ return &instance; }

unsigned int ADV1_state2 : :fail_random(int min, int max)
   {  if( has_srand == false)
         {  struct timeval time; struct timezone timez; char buffer[32]; int rc;
            rc=gettimeofday(&time, &timez);
            srand( (unsigned)time.tv_usec ); has_srand = true; }
      return min+(int)((max-min+1) * (float)rand()/(RAND_MAX+1.0)); }

void ADV1_state2 : :Always( fail : :Environment* env )
   {  struct timeval time; struct timezone timez; char buffer[32]; int rc;
      rc=gettimeofday(&time, &timez);
      cout « "FAIL TIME : STATE \"ADV1_state2\" (Always) : "
      « (unsigned)time.tv_sec « "," « (unsigned)time.tv_usec « endl;
      env->SetValue( "test_2", new failValues : :IntValue( fail_random (1, 10) ) ); }

void ADV1_state2 : :RegisterAllObservers( fail : :Environment* env ) { }

void ADV1_state2 : :Set( fail : :Environment* env )
    { env->SetValue( "timer_1", new failValues : :IntValue( 30 ) ); }

void ADV1_state2 : :Unset( fail : :Environment* env ) { }

void ADV1_state2 : :RegisterAllRules( fail : :Environment* env )
   {   env->AddObserver( new failEvents : :TimeoutEvent( "timer_1_TIMER" ),
            ADV1_state2_rule0 : :Instance() );
      env->GetTimerController()->AddTimer( "timer_1_TIMER",
            ((failValues : :IntValue*) env->GetValue("timer_1")) ->GetValue() ); }
```

FIG. 5.10 – Le fichier ADV1_state2.cpp

```
#include "ADV1_state2_rule0.h"
#include "ADV1_state1.h"
#include "ADV1_state2.h"
#include "ADV1_state3.h"
#include <src/tools.h>

ADV1_state2_rule0* ADV1_state2_rule0 : :Instance()
   {   struct timeval time; struct timezone timez; char buffer[32];
       int rc; rc=gettimeofday(&time, &timez);
       static ADV1_state2_rule0 instance;
       cout « "DAEMON RULE \"ADV1_state2_rule0\" loaded... : "
           « (unsigned)time.tv_sec « "," « (unsigned)time.tv_usec « endl;
       return &instance; }

bool ADV1_state2_rule0 : :CheckConditions( fail : :Environment* env )
   {   struct timeval time; struct timezone timez; char buffer[32];
       int rc; rc=gettimeofday(&time, &timez);
       cout « "FAIL TIME : RULE \"ADV1_state2_rule0\" : CheckConditions : "
           « (unsigned)time.tv_sec « "," « (unsigned)time.tv_usec « endl;
       failValues : :IntValue* val_1 = (failValues : :IntValue*) env->GetValue( "test_2" );
       if ( val_1 == NULL ) return false;
       return ( val_1->GetValue() <= 2 && true ); }

void ADV1_state2_rule0 : :Execute( fail : :Environment* env )
   {   env-> GetStateMachine()-> ChangeState( ADV1_state3 : :Instance() );
       env-> GetProgramController()-> Pause(); }
```

FIG. 5.11 – Le fichier ADV1_state2_rule0.cpp

```
<!– CONFIGURATION FILE FOR FCIDEPLOY. –>
<!– This file is used to configure fcideploy. –>
<!– (compilation, configuration and deployment of the fci daemons) –>

<deploy>

    <!– DO NOT MODIFY STATIC DECLARATIONS –>
    <staticcomputer name="P1"></staticcomputer>
    <staticgroup name="G1">5</staticgroup>
    <!– END OF STATIC DECLARATIONS –>

    <computer name="P1">
        <!– <wait>x</wait> –> <!– wait x second befor exec –>
        <exec>server</exec>
    </computer>

    <group name="G1">
        <!– <wait>x</wait> –> <!– wait x second befor exec –>
        <size>5</size>
        <exec>client</exec>
    </group>

    <!– <other on="P1"> –>
        <!– <exec>prog <refcomputer>P1</refcomputer></exec> –>
        <!– <focus>Experience finished</focus> –>
    <!– </other> –>

</deploy>
```

FIG. 5.12 – Le fichier de configuration *config/deploy.conf*

Deuxième partie

Expérimentations

Chapitre 6

Surcoût Induit par l'Utilisation de FCI

6.1 Introduction

Lors de l'utilisation d'un injecteur de fautes logicielles, l'application instrumentée partage les ressources matérielles avec celui-ci, ce qui entraîne un sûrcout lors de l'exécution de cette application par rapport à une exécution « normale », *i.e.* sans l'utilisation d'un injecteur de fautes.

Dans ce chapitre, nous allons présenter les tests sur le surcoût induit par l'utilisation de FCI. Ces tests ont pour but de déterminer l'impact de notre solution sur le temps d'exécution de l'application distribuée. Ils ont fait l'objet d'une publication dans le workshop IEEE/ACM Grid 2005 qui c'est déroulé à Seatle [41].

Tout d'abord, nous allons présenter l'application qui a été utilisée lors des tests, puis la configuration matérielle utilisée, et finalement, nous allons présenter les scénarios de fautes que nous avons créés pour ces tests et expliquer les résultats obtenus.

6.2 Application utilisée

L'application utilisée pour les tests est spécifique à ces tests et consiste en un serveur et plusieurs clients. Nous avons considéré deux types de clients possibles (qui ont été exécutés chacun sur 60 machines) :

1. le client « dormant » exécute 60 fois une boucle où il dort pendant 10 secondes et exécute alors une connexion TCP avec le serveur puis attend une réponse,
2. le client « bouclant » exécute 60 fois une boucle où il exécute un calcul (une addition en boucle) et exécute alors une connexion TCP avec le serveur puis attend une réponse.

Cette opération est alors répétée plusieurs fois et une moyenne des temps d'exécution de tous les clients est effectuée.

6.3 Configuration matérielle

Ces tests ont été éffectués sur 61 machines exécutant Linux (kernel 2.6.7). Trente machines étaient équipées chacune d'un processeur de 2083 MHz et de 885 Mb de RAM (le serveur était exécuté sur une de ces machines). Six machines étaient équipées chacune d'un bi-processeur de 1533 Mhz et de 885 Mb RAM. Dix huit machines étaient équipées chacune d'un processeur 1533 Mhz et de 885 Mb de RAM. Sept machines étaient équipées chacune d'un processeur de 1800 Mhz et de 504 Mb de RAM. Toutes les machines étaient connectées grâce à un réseau Ethernet de 100 Mbps.

6.4 Scénarios de fautes utilisés et résultats obtenus

La description des tests que nous avons faits et les résultats que nous avons obtenus sont les suivants :

Premièrement, nous avons exécuté l'application sans utiliser FCI, pour obtenir un temps d'exécution de référence de l'application. Le temps d'exécution que nous avons mesuré était de 600.37 secondes avec le client « dormant », et de 554.02 secondes avec le client « bouclant ».

Ensuite, nous avons exécuté l'application en utilisant FCI avec un scénario de fautes vide. Tous les programmes ont été alors exécutés à travers un démon FCI ; l'objectif était de mesurer le surcoût induit par notre architecture. Le temps d'exécution de l'application que nous avons mesuré était de 600.42 secondes avec le client « dormant », et de 554.03 secondes avec le client « bouclant ».

Puis, nous avons lancé l'application en utilisant FCI et un scénario générant des événements de type « time-based » (le démon FCI prend le control de l'application toutes les secondes). Lors des tests que nous avons effectués, chaque démon a été activé 556 fois en moyenne. Cependant, aucune faute n'a été injectée et aucune garde n'a été définie. Ceci nous a permis de vérifier l'impact de ce type d'événement déclencheur sur les performances. Le temps d'exécution de l'application que nous avons mesuré était de 600.47 secondes avec les clients « dormants », et de 555.75 secondes avec les clients « bouclants ».

Puis, nous avons exécuté l'application en utilisant FCI et des déclenchements lors

Client	Ref.	Empty	Time	Func.	1->all	all->1
Dormant	0	0.01%	0.02%	0.03%	0.08%	0.10%
Bouclant	0	0.00%	0.31%	0.08%	0.18%	0.17%

Fig. 6.1 – Surcoût de FCI

d'appels à une fonction spécifique par l'application testée (dans notre cas, quand l'application testée essayait d'exécuter la fonction `connect_and_read` lors de la réception d'un message, le démon FCI prenait le contrôle de cette application). Cette fonction est appelée 60 fois lors des tests. Dans ces tests, il n'y a toujours aucune faute injectée et aucune action exécutée. Le temps d'exécution mesuré était de 600.58 secondes avec les clients « dormants » et de 554.48 secondes avec les clients « bouclants ».

Puis, nous avons exécuté l'application en utilisant FCI et en utilisant des déclenchements sur appels d'une fonction particulière (60 appels à cette fonction ont été fait par chaque processus), mais cette fois nous avons exécuté des actions à chaque appel à cette fonction par l'application testée. Nous avons considéré deux types d'actions :

1. `one to all` : un démon envoie un message à tous les autres démons du système.
2. `all to one` : tous les démons envoient un message à un même démon préalablement désigné.

Le temps d'exécution que nous avons obtenu était de 600.83 secondes pour le cas `one to all` et de 600.95 secondes pour le cas `all to one` avec les clients « dormants », et 555.01 secondes pour le cas `one to all` et de 554.97 secondes pour le cas `all to one` avec les clients « bouclants » (tous ces résultats sont résumés dans la figure 6.1).

6.5 Conclusions sur le surcoût induit par FCI

Finalement, les tests préliminaires que nous avons exécutés montrent que le modèle « event-driven » de notre injecteur de fautes limite son impact (i.e surcoût calculé) sur le temps d'exécution de l'application à moins de 0.31% pour les cas considérés (voir la figure 6.1). Le plus grand impact est atteint avec les clients « bouclants » car le déroulement normal du programme est alors interrompu. L'impact plus important des déclenchements à un moment donné (déclenchement de type « time-based ») peut être expliqué par le nombre d'interruptions plus important de ces déclenchements lors des tests (556 occurences en moyenne contre 60).

Chapitre 7

Injection de Fautes

7.1 Introduction

Dans le chapitre précédent, les expérimentations ont été faites sur une application distribuée, programmée dans le but de tester FAIL-FCI, dont le code source était disponible et très simple (quelques dizaines de ligne de code C). De plus, les tests portaient uniquement sur le surcoût de la plate-forme FAIL-FCI et ont simplement montré que ce surcoût était, en pratique, négligeable.

Dans ce chapitre, nous allons étudier la tolérance aux fautes de différentes applications distribuées telles que XtremWeb, MPICH-V et FreePastry en injectant des fautes effectives grâce à FAIL-FCI. Ces expériences d'injection de fautes ont fait l'objet de plusieurs pubications : dans le *Second CoreGRID Workshop* en janvier 2006 [42], dans le *International Workshop on Java for Parallel and Distributed Computing (joint with IPDPS 2006)* en avril 2006 [44], dans la *conférence IEEE Cluster 2006* en septembre 2006 [39] et dans le journal *Future Generation Computer Systems* en 2007 [45].

Les tests que nous avons effectués ont permis de montrer que FAIL-FCI est adapté à l'étude d'applications distribuées et qu'il comble les nombreux inconvénients des autres approches présentées dans le chapitre 2. En effet, dans [39], des scénarios de fautes complexes ont pu être spécifiés et dans [39, 42, 44, 45], l'aspect aléatoire de l'injection des fautes a été utilisé. De plus, l'élaboration d'une expérience ne demande pas un travail trop important pour l'utilisateur (il doit principalement scpécifier le scénario de fautes en FAIL), l'injection de fautes est adaptée aux systèmes distribués à grande échelle du fait de l'architecture distribuée de FCI présentée dans le chapitre 5, l'injecteur de fautes est très peu intrusif comme le montre les résultats présentés dans la chapitre 6 et ne demande pas la modification du code source de l'application testée grâce à son principe

de fonctionnement présenté dans la section 5.2.

7.2 XtremWeb

Dans cette section, nous avons utilisé FAIL-FCI pour injecter des fautes et stresser XtremWeb [31]. La suite de la section est organisée de la façon suivante : la section 7.2.1 introduit la plate-forme XtremWeb, la section 7.2.2 décrit les conditions expérimentales, la section 7.2.3 décrit les tests préliminaires qui ont été exécutés, les sections 7.2.4 et 7.2.5 décrivent respectivement comment utiliser FAIL-FCI pour des tests d'injection de fautes quantitatives et pour des tests d'injection de fautes qualitatives.

7.2.1 Vue d'ensemble de XtremWeb

A la différence de systèmes comme Gnutella ou Freenet pour lesquels toutes les ressources ont un rôle identique et se comportent à la fois comme serveur, c'est à dire fournisseur de ressouces, et client, c'est à dire demandeur de ressources, XtremWeb distingue trois rôles essentiels.

Une architecture client/coordinateur/worker. Le *coordinateur* assure la mise en relation entre les demandes de ressources émanant des *clients* et les ressources mises à disposition nommées *worker* par la suite. Le coordinateur organise les calculs sur des ressources distantes géographiquement distribuées ainsi que les échanges d'informations entre ces ressources. Le coordinateur s'exécute à priori sur des ressources stables dans le système, alors que les clients et workers peuvent eux s'exécuter sur des noeuds volatiles. Cette distinction entre ressources volatiles et stables simplifie la gestion de la tolérance aux fautes et de la volatilité en centralisant la coordination globale du système sur des ressources stables. Précisons que ressource stable signifie que des défaillances peuvent survenir mais avec une faible fréquence. Le principe général étant qu'une faute survenant sur un élément quelconque du système ne provoque pas l'interruption de l'ensemble du système.

L'organisation première de XtremWeb, illustrée dans la figure 7.1, est une organisation totalement centralisée représentative des systèmes de calcul global tel que SETI@Home.

L'organisation centralisée qui semble être pertinente dans le cas du calcul global avec un fonctionnement maître-esclaves ; puisque c'est la solution adoptée par tous les grands projets du domaine (SETI@home, Entropia, United Devices, Parabon, etc.) ; peut sembler inappropriée dans le cas du mode d'interaction pair-à-pair (Peer to Peer). En effet, les

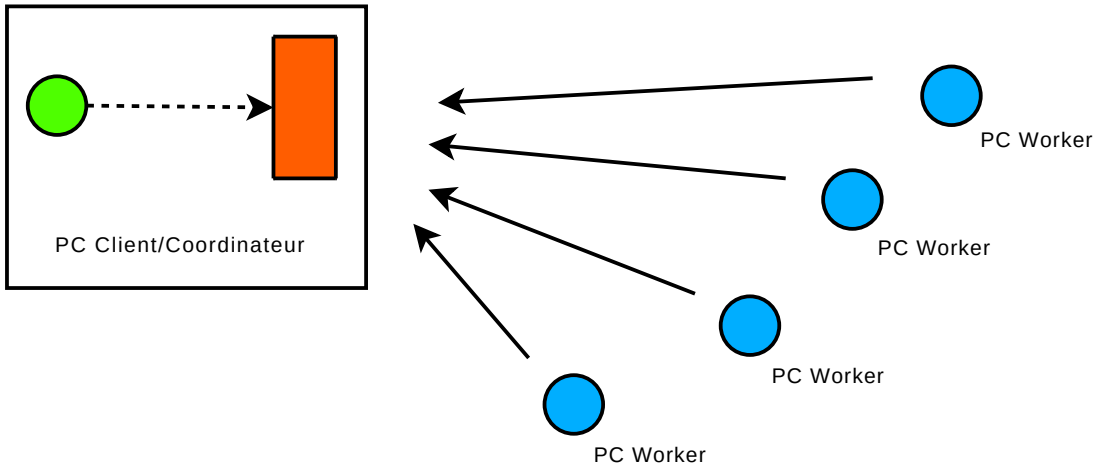

FIG. 7.1 – L'architecture *calcul global* de la plate-forme XtremWeb

ressources étant sensées remplir tour à tour le rôle de serveur et de ressources, il semblerait plus efficace de distribuer totalement le système pour éviter les goulots d'étranglement. Cependant, l'architecture centralisée n'est pas forcement inadaptée pour les systèmes pair-à-pair comme l'indique la figure 7.2. En effet, les systèmes totalement distribuées doivent construire et maintenir dynamiquement un mécanisme de repérage et d'allocation de ressources en n'utilisant pour ce faire que les ressources du système. Ceci constitue un problème complexe de système distribué.

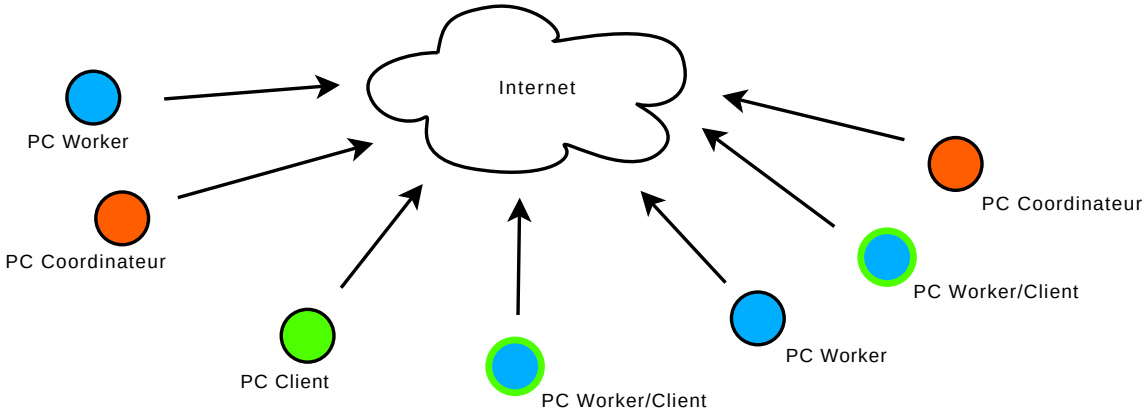

FIG. 7.2 – L'architecture *calcul pair-à-pair* de la plate-forme XtremWeb

Clairement, la topologie induite par la concentration des fonctions de cohérence sur des noeuds stables fait appartenir XtremWeb à la famille des systèmes pair-à-pair gras. Un système P2P gras défini dans [46] est un système où une faible fraction des noeuds assurent les services P2P (maintien de la topologie, publication, recherche) et où le plus grand nombre offre des services qui peuvent être la mise à disposition de leurs ressources.

Un système gras peut être construit par une hiérarchie de serveurs, comme l'illustre la figure 7.3. Par opposition, dans les systèmes P2P fins ou ultra-fins, la plupart des noeuds ou tous les noeuds assurent les services P2P comme c'est le cas dans les topologies ad-hoc par exemple.

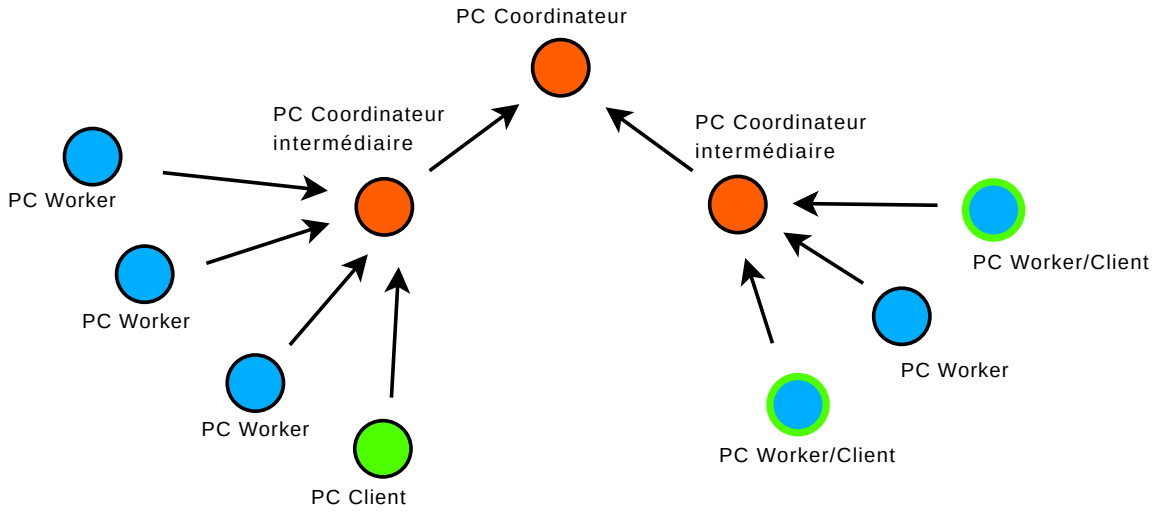

FIG. 7.3 – L'architecture *hiérarchique* de la plate-forme XtremWeb

Propriétés et architecture du worker. L'architecture du worker s'articule autour de quatre composants : le *WorkPool*, l'*exécuteur*, le *gestionnaire de communications* et le *moniteur d'activité*. La figure 7.4 présente cette architecture. Le WorkPool est une file d'attente des travaux à traiter localement par le worker. Cette file d'attente est utilisée selon le schéma producteur/consommateur par le gestionnaire de communication et l'exécuteur. Chaque événement sur la file d'attente déclenche une action de la part du lanceur d'exécution ou du gestionnaire de communication. Tant que la file n'est pas pleine, des demandes de travaux sont générées par le gestionnaire de communications vers le coordinateur. La file d'attente donne deux statuts aux travaux : «en exécution » et «en attente d'exécution ». Par défaut, il y a autant de tâches en exécution que de processeurs sur la machine. Cette file d'attente est sérialisée sur le disque, ce qui permet, en cas d'arrêt puis de redémarrage de la machine du worker, de reprendre les exécutions des travaux en attente d'exécution sans avoir besoin de connexion résau. L'exécuteur sélectionne dans cette file la prochaine tâche à exécuter. Lorsque cette tâche est terminée, la file est remise à jour pour tenir compte du statut de la tâche, et le résultat renvoyé au coordinateur.

La responsabilité d'obtenir ou de libérer les ressources de la machine du participant est assurée par le moniteur d'activité. la politique d'activation définit la disponibilité

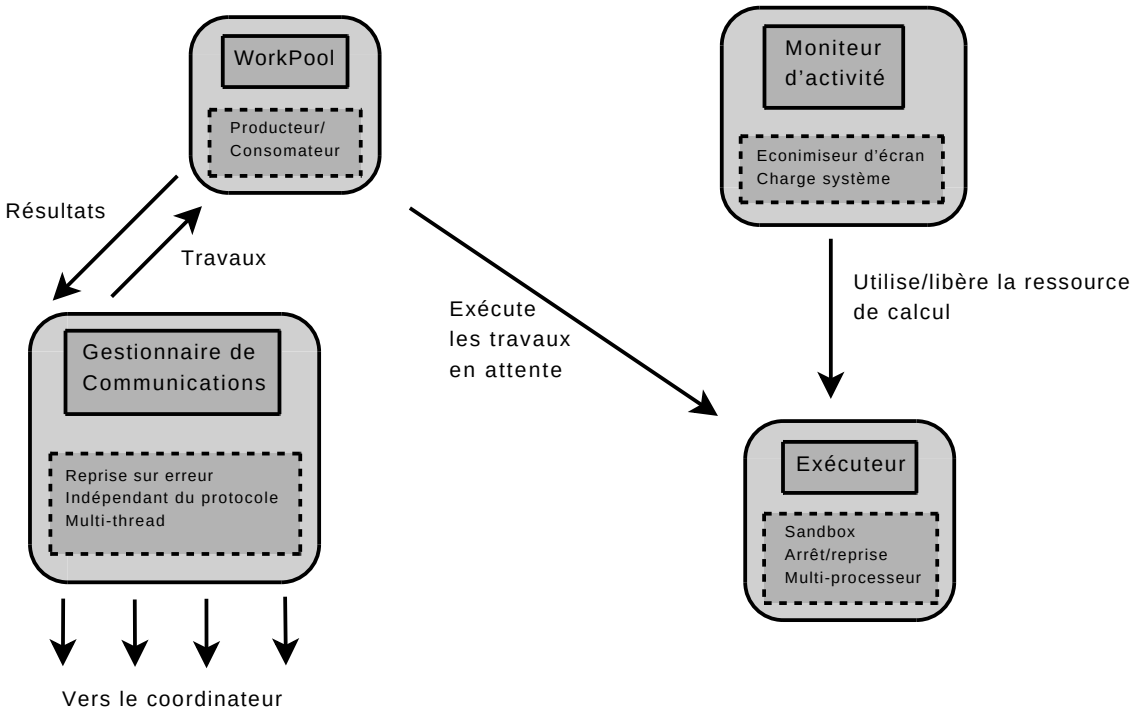

FIG. 7.4 – L'architecture du worker

d'un *worker* ; un *worker* est disponible quand il peut exécuter un *job* sans intrusion pour l'utilisateur de l'ordinateur. Le *activator.class* définit la politique d'activation (Quand le *worker* peut-il exécuter son *job* ?). Les politiques disponibles sont :

- *xtremweb.worker.AlwaysActive* : Cette politique permet au worker de toujours être autorisé à exécuter un job ; c'est la politique par défaut.
- *xtremweb.worker.DateActivator* : La disponibilité d'un worker est définie par la date and l'heure.
- *xtremweb.worker.CpuActivator* : La disponibilité d'un worker est définie par l'activité du processeur central.
- *xtremweb.worker.MouseKbdActivator* : La disponibilité d'un worker est définie par l'activité de la souris et du clavier. Si l'utilisateur n'utilise pas sa souris ou son clavier pendant un certain temps, le worker peut alors exécuter un job jusqu'a se que l'utilisateur utilise sa souris ou son clavier.
- *xtremweb.worker.WinSaverActivator* : La disponibilité d'un worker est définie par l'activité de l'économiseur d'écran : si l'économiseur d'écran est en cours d'exécution, le worker peut exécuter un job. Ceci est défini pour Win32 uniquement.
- *xtremweb.worker.MacSaverActivator* : La disponibilité d'un worker est définie par l'activité de l'économiseur d'écran : si l'économiseur d'écran est en cours d'exécu-

tion, le worker peut exécuter un job. Ceci est défini pour Mac OS X uniquement.

Les workers peuvent exécuter du code natif et du code intermédiaire Java. La protection de l'ordinateur d'un utilisateur suggère que le calcul d'une tâche s'effectue dans un environnement virtuel sécurisé, typiquement la machine virtuelle Java. Mais l'inconvénient majeur de cette approche réside dans les performances plus faibles d'exécution de l'application en comparaison d'un code binaire, même si les performances des machines virtuelles tendent à se rapprocher des performances d'exécution du code natif comme le montre certaines études comparatives. La deuxième approche consiste donc à exécuter du code natif. Les performances sont ainsi bien meilleures puisque le code exécutable est directement celui généré par un compilateur C ou Fortran. L'autre raison qui incite au support de l'exécution de code natif, est que beaucoup d'applications scientifiques utilisent des codes matures écrits en C ou Fortran. La portabilité est assurée en préparant plusieur versions de la même application compilée pour des machines et des sysèmes d'exploitation différents. Ces binaires sont stockés dans la base de données du coordinateur et fournis, à la demande, au worker. Le worker maintient un cache de codes binaires de façon à ce que chaque exécution de tâches n'entraîne pas un transfert du code binaire.

Le terme de calcul off-line définit la propriété suivante : l'utilisation des resources réseau est indépendante de l'utilisation des ressources de calcul. Cette propriété présente deux avantages. Le premier est de cibler les machines qui ont un fort taux de connexions/déconnexions alors que la ressource de calcul reste disponible. Ce sont principalement des ressources mobiles et des utilisateurs d'Internet disposant d'une connection tarifiée dans le temps dial-up. Le second est la possibilité d'arrêter ou de déconnecter le coordinateur sans perturber le fonctionnement des workers. XtremWeb implante cette propriété grâce à trois mécanismes :

1. indépendance du gestionnaire de communication et de l'exécuteur,
2. réserve de travail sur le worker,
3. et algorithme de répartition adapté sur le coordinateur.

En cas de déconnexion du worker, l'exécution peut puiser dans la réserve de travail une suite de travaux à effectuer. Lorsque le worker se reconnecte au coordinateur, il est possible que ces travaux aient déjà été accomplis par d'autres. Dans ce cas l'algorithme de répartition doit prévenir le worker de l'obsolescence de certains travaux. Le coordinateur donne un retour au message *workAlive* qui renseigne sur le statut du travail en cours d'exécution. Le statut peut être *continue*, auquel cas le travail se produit normalement, *stop*, auquel cas l'exécution est suspendue, *cancel*, auquel cas le travail est détruit de la reserve de travail ce qui déclenchera une nouvelle demande de travail vers le coordinateur.

L'utilisation de XtremWeb dans un environnement de production implique que le logiciel puisse utiliser les machines multiprocesseurs qui sont les composants de base des grappes de PC. Le support des machines multiprocesseurs est implanté en utilisant un exécuteur multi-thread. L'exécuteur est composé de plusieurs threads **ThreadWork** chargés de lancer l'exécution des processus. Par défaut, le worker utilise tous les processeurs de la machine hôte, donc lance un thread par processeur.

Propriétés et architecture du coordinateur. Le coordinateur centralise la gestion de la mutualisation des ressources sur les nœuds du réseau. Il assure :

- **la mise en relation.** Le coordinateur réalise la mise en relation entre les demandes de ressources issues des clients et les mises à disposition des ressources par les workers. Les workers et les clients s'enregistrent auprès du coordinateur, ce qui constitue l'étape préalable de recherche des ressources. Le coordinateur enregistre les travaux créés par les clients et les distribue aux workers lorsqu'ils en font la demande.
- **la coordination des exécutions.** Durant l'exécution, via la plate-forme XtremWeb, d'une application distribuée soumise par un client, certaines ressources quitteront, volontairement ou non, le réseau. La tâche de détecter et de pallier ces défections incombe au coordinateur. Les resources envoient vers le coordinateur un message signalant leur activité. Sur ce signal se greffe un mécanisme de timeout qui permet de détecter la disparition d'une ressource. Certaines applications parallèles telles que les application maître-esclaves tolèrent une variation du nombre de nœuds impliqués dans l'exécution. La disparition d'une ressource n'entraîne pas l'arrêt de l'application parallèle. Pour d'autres applications parallèles, il faut prévoir une solution de checkpoint puis de reprise d'exécution sur une ressource vacante du système. Le coordinateur fournit l'environnement adapté pour l'exécution de l'application parallèle.
- **la maintenance de l'infrastructure.** L'infrastructure est composée des ressources et des services mis à disposition par ces ressources. Le coordinateur maintient cette infrastructure via le registre des services.
- **la persistance.** Comme nous l'avons vu précédemment, les coordinateurs sont destinés à être exécutés sur des ressources stables. Le coordinateur assure la persistance de l'état du système en stockant dans une base de données les travaux, les résultats et diverses informations sur le comportement du système.

La figure 7.5 présente l'architecture du coordinateur qui s'articule autour d'une base de données, d'un gestionnaire de communications et d'un serveur de tâches et de ré-

sultats. XtremWeb utilise deux notions différentes pour représenter les délocalisations d'exécution : les *travaux* et les *tâches*. les travaux et leur résultats sont la représentation

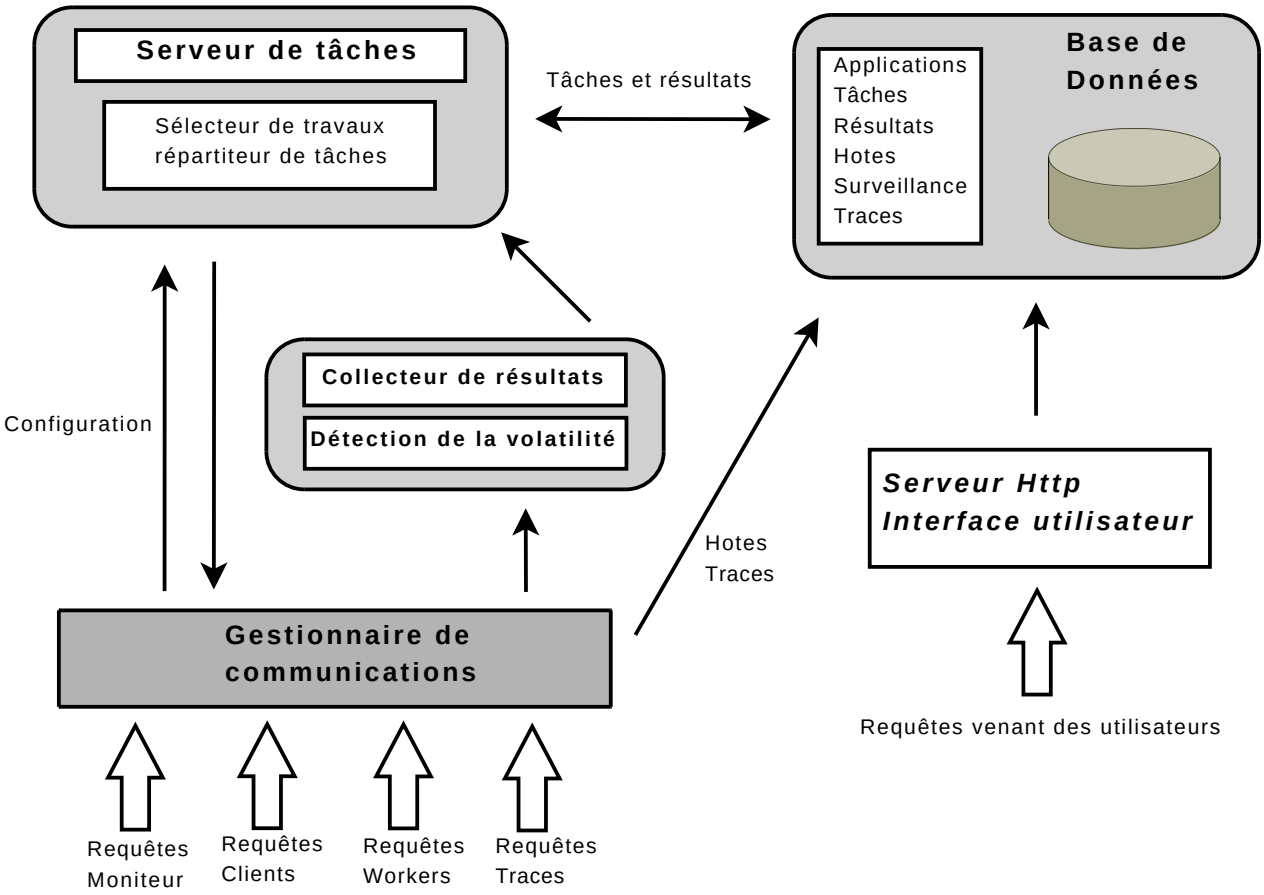

FIG. 7.5 – L'architecture du coordinateur

« statique » des exécutions. Les travaux sont définis par les paramètres de l'exécution : fichiers d'entrée, arguments de la ligne de commande, et sont stockés de manière persistante dans la base de données. Les tâches sont la représentation dynamique des exécutions. les algorithmes de placement/ordonnancement du coordinateur manipulent des structures de tâches, et non pas directement les travaux. Une tâche est créée en association avec les informations de description d'un travail, qui sont nécessairement transmises au worker pour qu'il puisse lancer l'exécution. A une tâche correspond une exécution en attente ou en cours sur un worker. Une exécution terminée sur un worker entraîne la destruction de la tâche. Si la terminaison a été détectée comme une faute du worker, alors une nouvelle tâche est créé qui réfère au même travail que la tâche interrompue. C'est cette nouvelle tâche qui sera attribuée à un worker. A un travail correspond donc une chronologie (les tâches sont chaînées par ordre de re-création) de une ou plusieurs tâche selon les interruptions d'exécution sur les workers.

Le coordinateur attribue aux requêtes de demande de travail issues des workers, des travaux sélectionnés dans la base de données. Lorsque les travaux ont été exécutés par les workers, ils rangent les résultats correspondants dans la base de données. Les clients peuvent interroger le coordinateur sur l'état d'avancement des travaux fournis. Lorsque le résultat est prêt, ils peuvent le retirer et détruire le travail sur le coordinateur. Lorsqu'un travail est créé par un client et lorsqu'une tâche est créée à partir de ce travail, ils sont enregistrés dans la base de données avec un unique identifiant de travail ou identifiant de tâche. Ces identifiants permettent les opérations d'ordonnancement et de récupération des résultats.

La base de données enregistre les informations persistantes du système. Ces données sont de deux natures : d'une part les fichiers qui peuvent être les fichiers de résultats ou de paramêtres des applications et d'autre part des « méta-données » qui sont des informations et des descriptions sur les travaux, les utilisateurs, les nœuds du réseau, etc. Cest pourquoi la base de données utilise à la fois une interface base de données, accesible via un langage SQL pour les méta-données, et un système de fichiers.

Propriétés et architecture du client. L'API Client est une interface pour programmer des applications qui peuvent s'exécuter sur la plate-forme XtremWeb. La figure 7.6 présente cette API qui permet à l'utilisateur du système de dialoguer avec le coordinateur pour spécifier un environnement d'exécution souhaité (nombre de ressources Workers), envoyer des tâches vers le serveur et récupérer les résultats des tâches soumises. Comme pour le Worker, les communications sont à l'initiative du client. Client et Worker peuvent fonctionner sur les mêmes machines.

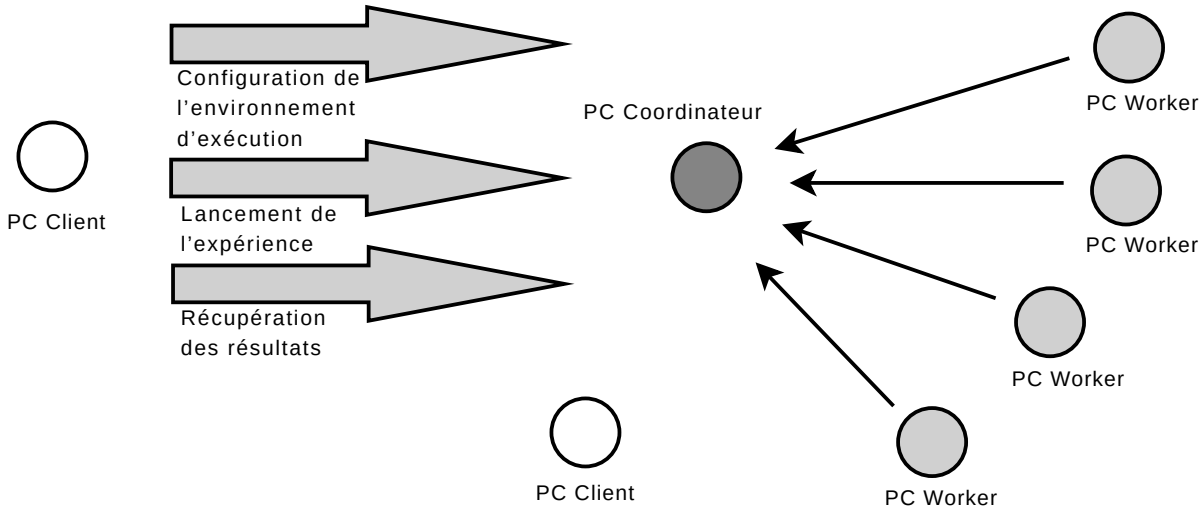

Fig. 7.6 – Fonctions de l'API Client

Les principales fonctionnalitées fournies par cette API sont :

- **la gestion de l'environnement d'exécution.** Elle consiste à demander au serveur un nombre de Workers pour exécuter l'application. On peut l'étendre à la prise en compte d'autres requêtes : types de machines, durée d'exécution totale, durée d'exécution par machines, etc.
- **la gestion de l'exécution.** Elle consiste en quelques fonctions de base permettant de : soumettre une tâche au serveur, demander l'état d'avancement d'une tâche, récupérer le résultat d'exécution d'une tâche, arrêter l'exécution d'une tâche.

7.2.2 Conditions Expérimentales

1. **Configuration Matérielle :**

   Les expériences faites sur XtremWeb ont été exécutées sur deux configurations matérielles différentes, pour montrer la polyvalence de notre mécanisme d'injection de fautes :

   - **Cluster :** Dans ce cas, les tests ont été effectués sur 61 machines exécutant Linux (kernel 2.6.7). Toutes les machines étaient équipées d'un processeur de 32 bits dont la fréquence variait entre 1533 MHz et 2083 MHz et étaient équipées d'une mémoire vive dont la capacité variait entre 504Mb et 885 Mb. Toutes les machines étaient connectées grâce à un réseau Ethernet de 100 Mbps. Les tests préliminaires ont été exécutés en utilisant les 61 machines du cluster (soit 60 workers), les tests d'injection de fautes quantitatives, qualitatives, et les expériences de stress de l'application ont été exécutés en utilisant 31 machines du cluster (soit 30 workers).
   - **GrideXplorer :** Dans ce cas, les tests ont été effectués sur 160 machines exécutant Linux (kernel 2.6.14.3). Toutes les machines étaient équipées d'un bi-processeur de 64 bits d'une fréquence de 1994 Mhz chacun, d'une mémoire vive d'une capacité de 2 Gb, et étaient connectées grâce à un réseau Ethernet Gigabit. Tous les tests effectués avec cette configuration ont été exécutés en utilisant les 160 machines. Le coordinateur était exécuté sur une machine particulière extérieure à GriDeXplorer (lri7-209) et un worker a été exécuté sur chacune des 160 machines (soit 160 workers).

2. **Configuration de XtremWeb :**

   L'application XtremWeb qui était exécutée sur la plate-forme était POV-Ray. C'est une application de rendu graphique qui crée des images en trois dimensions réalistes en utilisant une technique appelée `ray-tracing`. Nous avons considéré deux types de tâche lors de nos expériences :

- **type 1 :** la tâche consiste à calculer la même image deux fois. Le calcul d'une image POV-Ray est exécuté de la manière suivante : premièrement, le dispatcher divise l'image à calculer en plusieurs parties, chaque partie sera calculée par un worker, ensuite, après que chaque worker a fini de calculer sa portion d'image et de l'envoyer au dispatcher, le dispatcher selectionne un seul `worker` pour collecter toutes les parties de l'image et pour calculer l'image finale à partir d'elles. Cette image sera finalement envoyée au dispatcher.
- **type 2 :** la tâche consiste à calculer une image particulière. Cette opération est requise un nombre suffisant de fois pour que les mesures aient un sens. Quand le dispatcher reçoit une requête d'un worker, il lui envoie toutes les informations nécessaires pour exécuter le calcul d'une image complète.

Pour toutes les expériences réalisées, le dispatcher et le client XtremWeb étaient placés sur une machine particulière qui n'était pas soumise aux différents scénarios de fautes (*i.e* aucun démon FCI n'était exécuté sur cette machine). La charge du client n'influence pas le dispatcher. En effet, le client et le dispatcher s'exécutent quasiment de façon séquentielle ; le client envoie une liste de tâches au dispatcher au début de l'expérience, et le dispatcher avertit le client quand les tâches ont été complétées et les résultats disponibles. Un worker est exécuté sur chacune des autres machines du cluster ou de GrideXplorer suivant la configuration matérielle (30 et 160 respectivement, à l'exception des tests préliminaires où les 60 machines du cluster ont été utilisées pour exécuter un worker chacune). Avant le début d'un test, le dispatcher est exécuté ainsi que tous les workers. Ensuite, le client est exécuté et le moment correspondant au début d'exécution du client est référé comme le temps de début de l'expérience. Quand le client se termine, ce moment est référé comme temps de terminaison de l'expérience.

7.2.3 Tests préliminaires

Le premier ensemble de tests que nous avons effectué était destiné à valider notre approche en injectant des fautes dans une application distribuée « réelle » (XtremWeb). Pour cela, nous avons créé un scénario de fautes FAIL. Le `dispatcher` est exécuté sur la machine 1 et n'est pas sujet aux fautes (il attend simplement que des `workers` se connectent et leur envoie alors des tâches). Ensuite, les `workers` XtremWeb sont exécutés sur les 35 ordinateurs restants et sont sujets aux fautes. Le scénario de fautes qui est exécuté sur toutes les machines exécutant un `worker` est le suivant :

- Toutes les 5 secondes, un worker XtremWeb peut s'interrompre avec une probabilité `x`,
- Après une interruption, toutes les 5 secondes, un worker peut être relancé avec une probabilité 0.3,
- Un `worker` peut être interrompu seulement une fois.

Le code FAIL qui implante le scénario précédent (avec `x` égale à 0.1) est présenté dans la figure 7.7.

```
Daemon ADV1
{}

Daemon ADV2
{ time_g timer = 5;
  node 1 : always int rand = FAIL_RANDOM (1,10);
           timer && rand < 2 -> halt, goto 2;
  node 2 : always int rand = FAIL_RANDOM (1,10);
           timer && rand > 7 -> restart, goto 3;
  node 3 : }

Computer P1
{ program = "/users/parall/hoarau/XW++/Dispatcher";
  daemon = ADV1; }

Group G1
{ size = 34;
  program = "/users/parall/hoarau/XW++/RealWorker";
  daemon = ADV2; }
```

FIG. 7.7 – Premier scénario pour une injection de fautes dans XtremWeb.

Nous avons exécuté ces tests en utilisant différentes valeurs pour x (0.1, 0.3, 0.5, 0.7 et 0.9), et les résultats des expériences sont résumées dans la figure 7.8. Dans cette figure, *Added 1* et *Added 2* correspondent au temps où le dispatcher a collecté toutes les parties des images à partir de tous les workers concernés pour la première image et la seconde image respectivement. *Completed 1* et *Completed 2* correspondent au temps où

le dispatcher reçoit la première image complète et la deuxième respectivement. Le test avec une probabilité 0 sert de test référence.

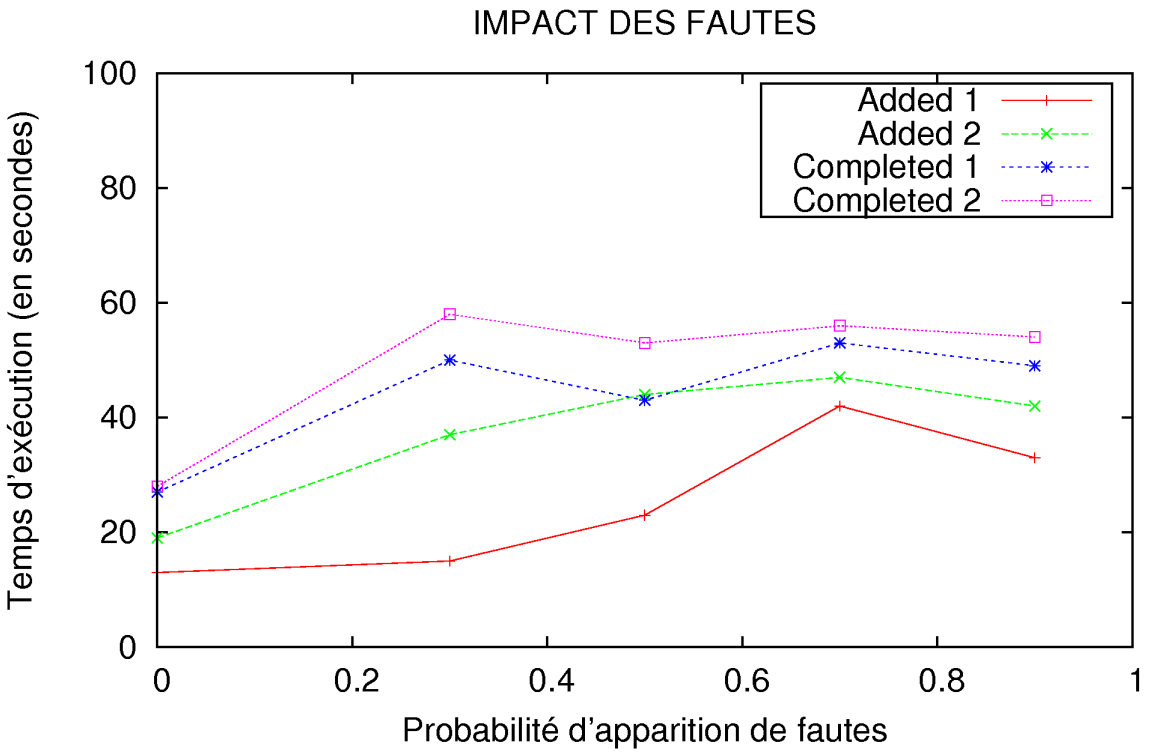

FIG. 7.8 – Resultats d'injection de fautes FCI

Les résultats de ces tests d'injection de fautes ne sont pas surprenants. Premièrement, pour toutes les probabilités d'apparition de fautes, le temps nécessaire pour compléter la seconde image est inférieur au double du temps nécessaire pour compléter la première image. Ceci est dû au scénario dans lequel une machine ne peut pas être victime d'une faute deux fois, ainsi si un worker s'interrompt pendant le calcul de la première image et se rétablit (ceci arrive avec une probabilité de 0.3), il ne s'interrompra pas durant le calcul de la seconde image. Quand la probabilité x d'apparition de fautes est faible (en dessous de 0.5), le temps de calcul des parties d'images est faible, car les workers sont seulement responsables d'une petite partie du calcul global, ainsi le travail d'un worker interrompu peut être réexécuté par un autre worker rapidement. Cependant, le temps de reconstitution de l'image complète est important, car si un worker s'interrompt lors de cette phase, il est le seul responsable du calcul de l'image complète, donc son interruption a un impact important.

Quand la probabilité d'apparition de fautes est importante (plus de 0.5), le temps de calcul des parties d'images est très important car un nombre important de workers peuvent s'interrompre lors de cette phase. Le temps de reconstitution de l'image final est aussi élevé, mais le coût en temps de cette phase n'est pas aussi importante que lorsque la probabilité d'apparition de fautes est faible, simplement parce que la plupart des fautes interviennent pendant le calcul des parties d'image, ainsi la plupart des workers ne s'interrompront pas lors de la phase de reconstitution des images.

7.2.4 Tests d'injection de fautes quantitatives

Nous avons défini un scénario de fautes probabiliste pour avoir une vue quantitative sur la tolérance aux fautes de XtremWeb. Nous nous sommes assurés que le dispatcher et le client ne sont pas sujets aux fautes (*i.e* toutes les tâches peuvent être soumises et tous les résultats peuvent être retournés). Les workers XtremWeb sont exécutés sur les machines restantes qui sont sujettes aux fautes. Le temps d'exécution est le temps correspondant aux temps où les résultats sont collectés moins le temps correspondant au départ du client. Le modèle de fautes est le suivant : toutes les x secondes, chaque worker XtremWeb peut s'interrompre (cesser leur exécution) avec la probabilté y. Cependant, nous voulons nous assurer qu'il existe un worker particulier qui ne peut pas s'interrompre pour garantir que le temps d'exécution est toujours fini. Ce scénario peut être exprimé d'une façon singulièrement compacte en utilisant le langage FAIL comme le montre la figure 7.9 (avec x=5, y=10% et 30 workers).

Nous allons maintenant décrire, de manière informelle, le code FAIL présenté dans cette figure. Deux machines à états sont définies, `ADV1` et `ADV2`. La machine à états `ADV1` est associée à la machine P1 (qui exécutera le programme `dummy`) et la machine à états `ADV2` est associée à 30 machines (qui forme le groupe G1), chacune exécutant le programme `WorkerStatic` avec les mêmes paramètres.

`ADV2` fonctionne de la façon suivante : le démon attend, dans un premier temps, que le programme s'est chargé, mais avant d'exécuter la fonction `main`, le programme est interrompu. L'exécution du programme continue quand le démon `ADV1` envoie soit le message `« ok »`, soit le message `« go »`. Le démon `ADV1` envoie simplement le message `« ok »` à un démon particulier (déterminé de façon aléatoire) du groupe `G1` et le message `« go »` à tous les démons du groupe `G1`. Ainsi, un démon du groupe `G1` reçoit un message `« ok »` en premier et se place alors dans un nouvel état (`node 4`) depuis lequel il continue simplement l'exécution du programme ignorant tous les autres messages ou événements. Le worker correspondant s'exécutera alors jusqu'à terminaison normale (*i.e* sans interven-

```
spyfunc main;

Daemon ADV1
{ node 1 : before(main) -> continue, !ok(G1[1]), !go(G1), goto 2;
  node 2 : }

Daemon ADV2
{ node 1 : before(main) -> stop, goto 2;
  node 2 : ?ok -> continue, goto 4;
           ?go -> continue, goto 3;
  node 3 : always int x = FAIL_RANDOM(1,100); always time_g timer = 5;
           timer && x <= 10 -> halt, goto 4;
           timer && x > 10 -> continue, goto 3;
  node 4 : }

Computer P1
{ program = "dummy";
  daemon = ADV1; }

Group G1
{ size = 30;
  program = "WorkerStatic -i lri7-209";
  daemon = ADV2; }
```

FIG. 7.9 – Scénario d'injection de fautes quantitatives.

tion du démon FCI). Au contraire, les autres démons du groupe `G1` reçoivent le message « `go` ». Ces démons se placent donc dans un état différent (`node 3`) et reçoivent alors des événements temporels (toutes les 5 secondes). Quand ce temps expire, avec une probabilité de 0.1, le processus testé est tué et avec une probabilité 0.9, le processus continue son exécution pendant encore 5 secondes. Ce mécanisme est répété jusqu'a la fin du test.

Nous avons effectué ce test avec deux valeurs pour x (5 et 10 secondes) et y variant de 10% à 90% avec une incrémentation de 10%. Les résultats obtenus concernant les temps d'exécution de l'ensemble des tâches sont résumés dans la figure 7.10 et la figure 7.11.

Comme nous pouvons le voir dans ces figures, dans certains cas lors des tests sur le

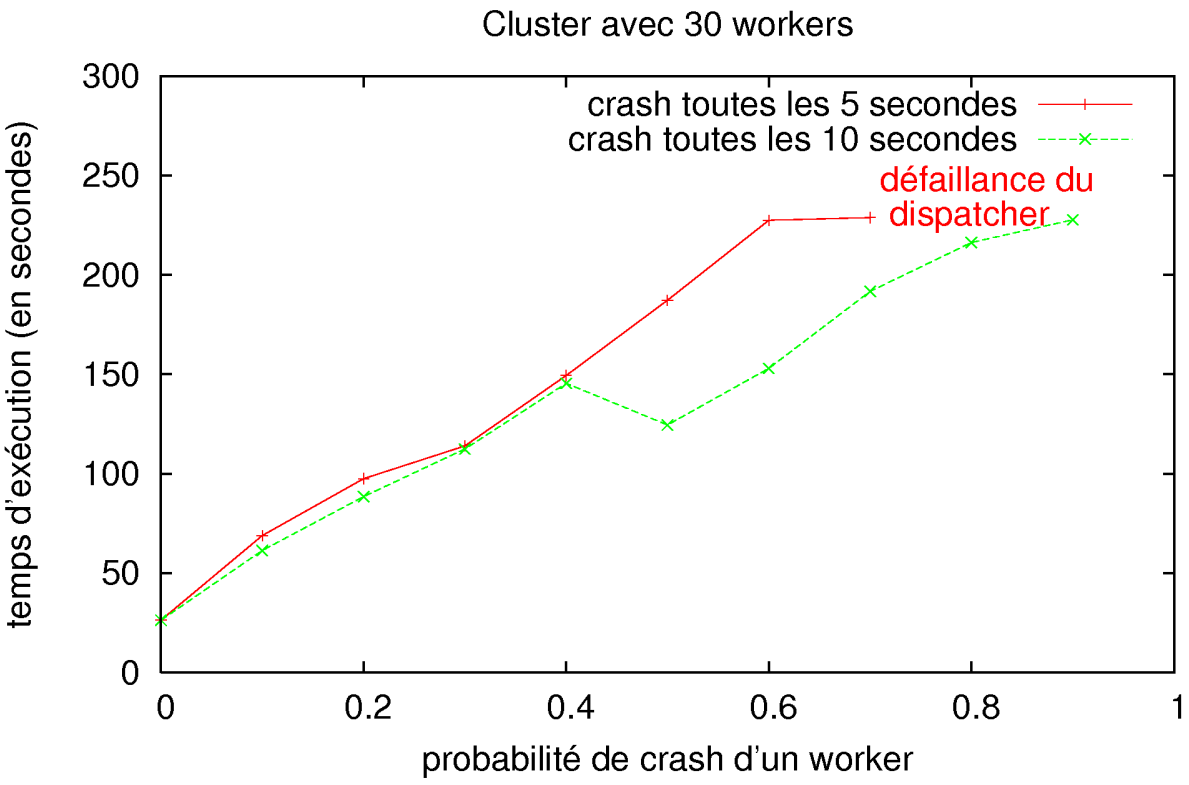

FIG. 7.10 – Impact du crash des workers pour le cluster avec 30 workers

cluster, les calculs ne se sont pas terminés (c'est pourquoi il n'y a pas de résultats pour le cas toutes les 5 secondes avec la probabilité de 0.8 et 0.9) à cause d'un dysfonctionnement du dispatcher de XtremWeb (il faut se rappeler que le dispatcher de XtremWeb n'était soumis à aucune injection de fautes par la plate-forme FCI). Nous avons donc collecté des informations sur les dysfonctionnements du dispatcher pendant les tests, ces informations sont présentées dans la figure 7.12.

Avant d'exécuter les tests, on aurait pu s'attendre à ce que les deux courbes augmentent avec un écart entre elles. Dans la configuration `Cluster`, quand il n'y a aucune injection de fautes, le temps nécessaire pour compléter toutes les tâches est approximativement 25 secondes. A partir d'une probabilité d'apparition de fautes de 0.4, les resultats correspondent à ce qui était prévu, mais pour une probabilité d'apparition de fautes plus faible, la fréquence d'apparition de fautes ne change pas significativement le temps d'exécution. Aussi, quand des fautes apparaissent seulement toutes les dix secondes, il y a une sorte d'équilibre (entre une probabilité de 0.4 et de 0.6) où le temps d'exécution ne change pas beaucoup. Cet équilibre reflète le fait que si plus de fautes apparaissent, cela

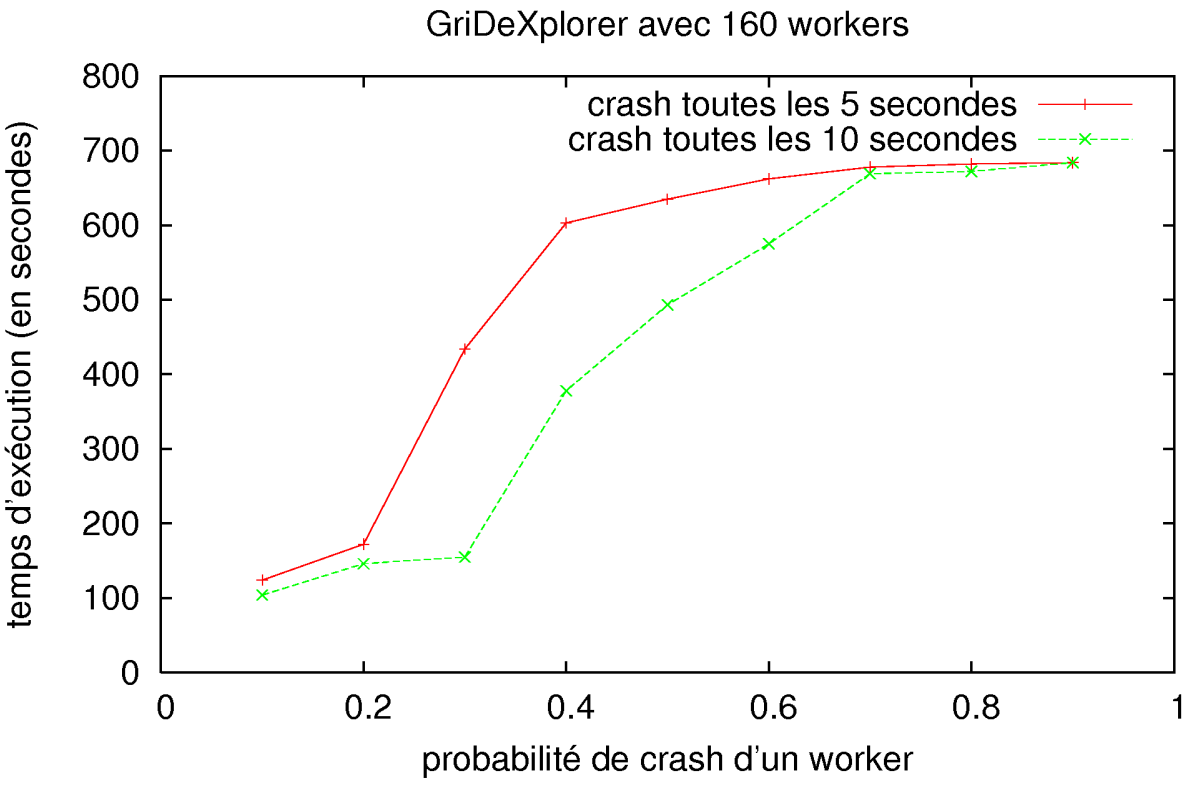

FIG. 7.11 – Impact du crash des workers pour GriDeXplorer avec 160 workers

veut dire que moins de fautes pourront apparaître (car il y a moins de machines) dans le futur. Nous pouvons voir ce même phénomène dans la configuration GriDeXplorer. Mais, pour cette configuration, ce phénomène apparaît lorsque la probabilité d'apparition de fautes est plus faible (entre une probabilité de 0.2 et 0.3) et il est moins marqué. Après ce phénomène, les deux courbes augmentent avec un écart comme pour la configuration `Cluster`, mais cette augmentation est plus importante. En effet, pour le cas toutes les 10 secondes, le temps d'exécution pour une probabilité de 0.3 est de 2.4 fois le temps d'exécution pour une probabilité de 0.4. A partir d'une probabilité d'apparition de fautes de 0.3, l'échelle de la configuration `GriDeXplorer` a un impact réel sur les performances comparé à la configuration `Cluster`. Avec une probabilité d'apparition de fautes plus faible, cet impact est négligeable.

Quand certains tests ne se sont pas terminés, nous avont détecté que dans ces cas le dispatcher était toujours en cours d'exécution mais n'était plus disponible (*i.e* les workers ne pouvaient pas communiquer avec le dispatcher pour lui notifier qu'ils avaient complété leur tâche). La figure 7.12 montre que, à partir d'une probabilité d'apparition

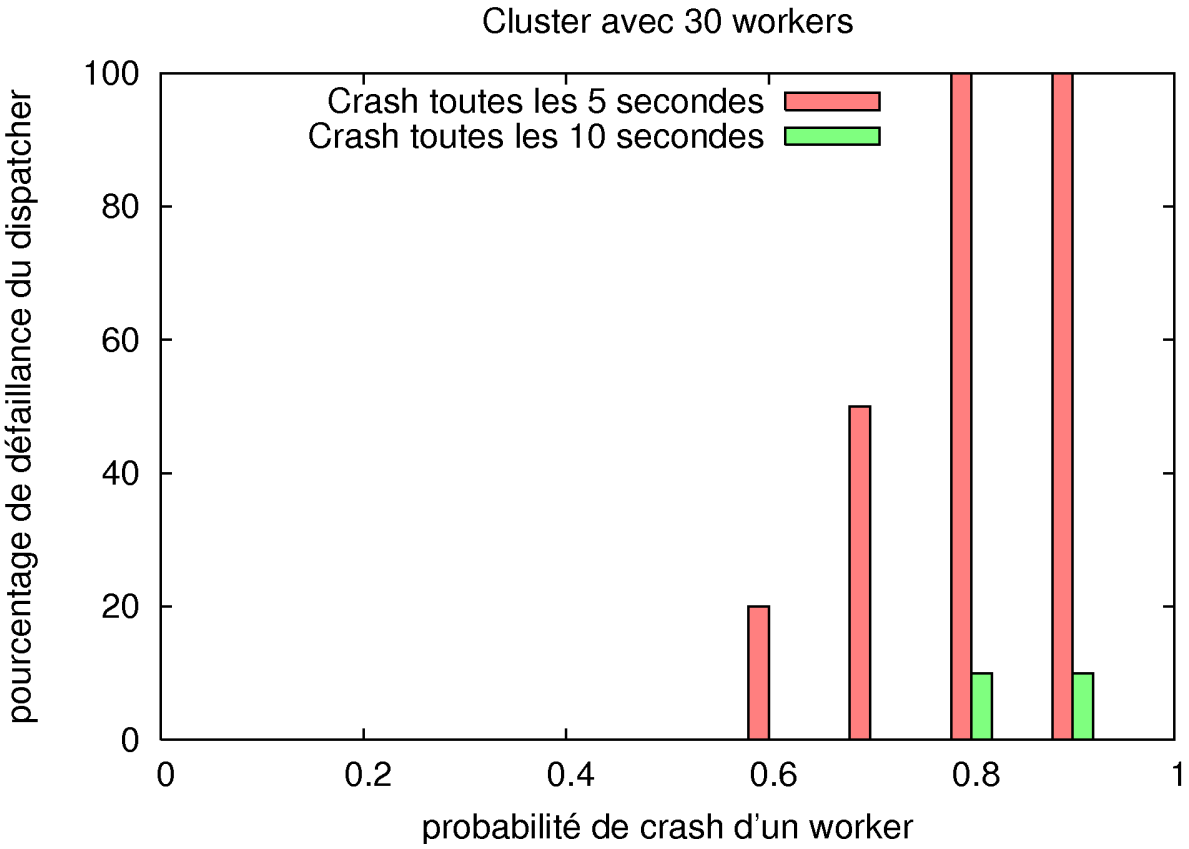

FIG. 7.12 – Impact du crash des workers sur la défaillance du dispatcher

de fautes de 0.7 toutes les 5 secondes, 50% des exécutions ne se sont pas terminées à cause du dysfonctionnement du dispatcher. Pour une probabilité d'apparition de fautes de 0.8 toutes les 5 secondes, les expériences ne se terminent jamais à cause du dysfonctionnement du dispatcher. Ce dysfonctionnement du dispatcher révèle probablement un bogue qui se produirait extrèmement rarement dans un cluster en fonctionnement normal, car ces taux de fautes sont extrêmes : toutes les 5 secondes, 80% des noeuds du reseaux s'interrompent ! Une observation surprenante est que, dans la configuration `GriDeXplorer`, tous les tests se sont terminés correctement malgré le nombre plus important de workers. L'échelle n'est donc probablement pas ce qui explique le dysfonctionnement du dispatcher dans la configuration `Cluster`. La différence matérielle des machines (un bi-processeur et 2 Gb de mémoire vive contre un seul processeur et moins de 1 Gb de mémoire vive) porte à croire que la différence est due probablement à un épuisement des ressources ou à un problème de synchronisation des threads.

7.2.5 Tests d'injection de fautes qualitatives

L'évaluation qualitative présentée dans la section 7.2.4 pouvait aussi être faite en utilisant des scripts appropriés même si cela requiert un travail beaucoup plus lourd et fastidieux. Dans cette section, nous allons introduire des tests d'un degré plus complexe correspondant à une évaluation qualitative des fautes qui peuvent potentiellement toucher les systèmes. Nous nous intéressons ici à la partie du code exécuté par les workers au moment où la faute apparaît. En particulier, nous considérons les quatre états logiques possibles pour un worker XtremWeb suivants :

1. *job received :* le worker XtremWeb a reçu une tâche à exécuter de la part du dispatcher.
2. *job comput. :* le worker XtremWeb a fini d'exécuter sa tâche.
3. *job finished :* le worker XtremWeb a notifié au dispatcher qu'il avait fini sa tâche.
4. *job completed :* le worker XtremWeb a envoyé au dispatcher les résultats correspondant à la tâche finie.

Notre but dans cette série de tests est de fixer le nombre de workers (30 et 160 respectivement) et la probabilité d'apparition de fautes (40%), mais en considérant qu'un worker ne peut seulement défaillir à un moment précis de son exécution qui correspond à l'entrée dans l'un des états mentionnés précédemment. Le code FAIL corespondant (*i.e.* scénario de fautes) est présenté dans la figure 7.13 (dans cet exemple on considère que les fautes ne peuvent se produire uniquement lorsque les workers sont dans l'état *job completed*).

Comme dans la section 7.2.4, il y a deux machines à états `ADV1` et `ADV2` qui sont dispatchées de la même manière que précédemment. Le même mécanisme d'envoi de messages « ok » et « go » est utilisé pour avoir au moins un worker non soumis aux fautes. La différence principale est l'utilisation de points d'arrêt pour stopper l'application testée lorsqu'une fonction particulière est appelée par celle-ci. Dans ce scénario, les méthodes *DataSaved* et *release* de la classe *Protocol* sont positionnées comme points d'arrêt. L'état *job completed* est atteint après que l'appel à la méthode *DataSaved* est fini et juste avant l'appel à la méthode *release*. On peut noter que la méthode *release* est souvent appelée dans différents contextes de l'exécution du worker XtremWeb, mais le worker n'entre dans l'état *job completed* que lorsque cette méthode est appelée après que la méthode *DataSaved* a été exécutée.

Les résultats obtenus sont résumés dans la figure 7.14 et la figure 7.15. Dans ces figures, la catégorie *without fault* correspond aux tests sans injection de fautes (sert de

```
spyfunc main ; spyfunc Protocol : :DataSaved ; spyfunc Protocol : :release ;

Daemon ADV1 {
  node 1 : before(main) -> continue, !ok(G1[1]), !go(G1), goto 2 ;
  node 2 :
}

Daemon ADV2 {
  node 1 : before(main) -> stop, goto 2 ;
  node 2 : ?ok -> continue, goto 5 ;
           ?go -> continue, goto 3 ;
  node 3 : always int x = FAIL_RANDOM(1,100) ;
           before(Protocol : :DataSaved) && x <= 40 -> continue, goto 4 ;
           before(Protocol : :DataSaved) && x > 40 -> continue, goto 3 ;
  node 4 : before(Protocol : :release) -> stop, goto 5 ;
  node 5 :
}
```

FIG. 7.13 – Scénario d'injection de fautes qualitatives.

base de comparaison). Pour chacun des quatre états possibles des workers mentionnés précédemment, deux types de fautes sont considérés :

1. suspension du processus (en utilisant `stop` dans le langage FAIL) pour simuler une machine surchargée,
2. interruption du processus (en utilisant `halt` dans le langage FAIL).

Nous n'avons pas collecté d'informations sur les défaillances du dispatcher, car aucun crash du dispatcher n'a été observé dans les deux configurations (ce qui était prévisible même dans la configuration `Cluster`, car la probabilité d'apparition de fautes était de 40%).

On pouvait s'attendre à ce que le fait d'injecter des fautes de type `stop` induise des performances plus mauvaises que d'injecter des fautes de type `halt` (car dans le premier cas, l'autre bout de la connexion TCP, *i.e.* le dispatcher, n'est pas alerté par la couche réseau d'une erreur de connexion, alors que dans le second cas, c'est généralement le cas). Ceci a été confirmé par les résultats obtenus sur le cluster et sur GriDeXplorer. Nous

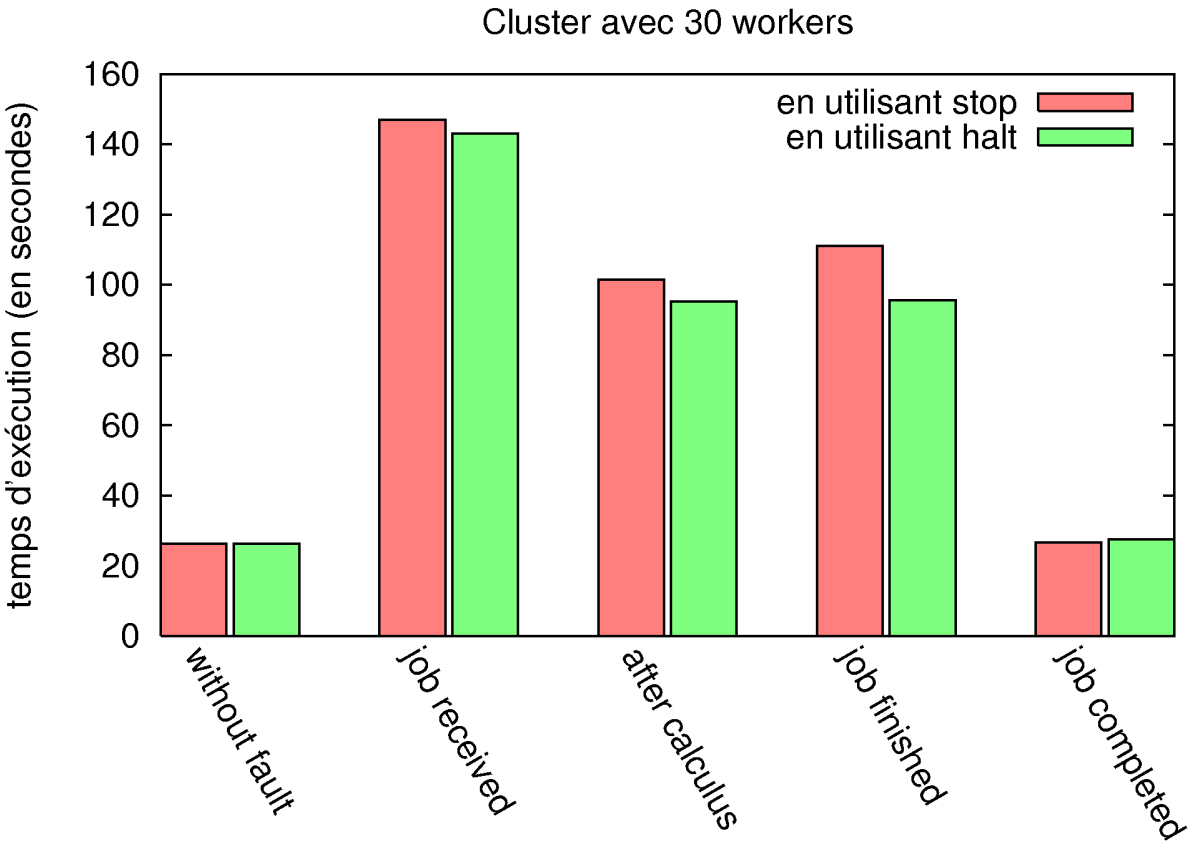

Fig. 7.14 – Impact de l'état du worker lors d'un crash (Cluster avec 30 workers)

avions également prévu que plus l'injection de fautes serait tardive (mais avant que les resultats soient renvoyés au dispatcher), plus le temps nécéssaire à la completion de toutes les tâche serait important. Cependant, et étonnament, si les workers s'interrompent avant même d'avoir lancé un calcul, les performances sont plus mauvaises que s'ils se seraient interrompus après un calcul. Ce comportement, qui apparait dans les deux configurations, est probablement dû à une mauvaise conception du dispatcher XtremWeb, qui ne doit pas s'attendre à la défaillance d'un worker juste après qu'une tâche a été envoyée (à ce moment, il n'est probablement pas en train de regarder la connexion TCP avec le worker, tandis qu'il l'est quand le job est presque complété). Nous avons également remarqué que si un worker s'interrompt après qu'une tâche a été completée (le worker avertit le dispatcher que les resultats sont disponibles), alors les performances sont pratiquement les mêmes que si aucune faute n'avait été injectée. L'échelle n'a pas un impact significatif sur les variations de performances quand un taux de fautes fixé apparaît dans différentes parties specifiques de l'exécution de l'application, car le comportement général est presque le même sur les deux configurations.

FIG. 7.15 – Impact de l'état du worker lors d'un crash (GriDeXplorer avec 160 workers)

7.3 MPICH-V

7.3.1 MPICH-V : MPI tolérant aux fautes

MPI, dans sa spécification [75] et dans ses implantations les plus déployées (*i.e* MPICH [35]) suit la sémantique *fail stop* (les specifications et implantations ne fournissent pas de mécanismes pour la détection de fautes et la reprise après apparition de fautes). Ainsi, les applications MPI exécutées sur un cluster de grande échelle peuvent être stoppées à tout instant pendant leur exécution à cause d'une défaillance du système.

Le besoin d'implantation de MPI tolérant aux fautes a récemment réactivé la recherche dans ce domaine. Plusieurs projets de recherche explore la tolérance aux fautes à différents niveaux : réseau [67], système [11], applicatif [30]. Différentes stratégies ont été proposées pour implanter la tolérance aux fautes dans MPI : a) détection et gestion par l'utilisateur/programmeur, b) pseudo-automatique guidée par le programmeur et c) complètement automatique et transparente. Pour cette dernière catégorie, plusieurs protocoles ont été étudiés dans la litérature. En conséquence, pour l'utilisateur et l'administrateur

système, il y a un choix, non seulement sur le type d'approche pour la tolérance aux fautes, mais aussi sur plusieurs protocoles de tolérance aux fautes pour chacun des types d'approche.

Choisir le meilleur protocole de tolérance aux fautes dépend grandement du nombre de composants, du schéma de communication pour l'application parallèle et du comportement du système en présence de fautes. Dans ces travaux, nous avons mis sous tension l'implantation du protocole de Chandy-Lamport du projet MPICH-V [38] avec un taux de fautes très important pour pouvoir définir son niveau de tolérance aux fautes.

L'algorithme de Chandy-Lamport [21] propose d'implanter la tolérance aux fautes à travers un mécanisme de reprise à partir d'un snapshot cohérent. Pendant l'exécution, les composants peuvent déclencher des vagues de *checkpoint*, qui construisent une vue cohérente de l'application distribuée. Chaque processus enregistre son image sur un média fiable lors de la prise d'une vue cohérente. Quand un processus est sujet à une défaillance, l'application distribuée est interrompue, des ressources de calcul sont allouées pour remplacer les processus défaillants et tous les processus *rollback* (*i.e* tous les processus chargent leur dernier *checkpoint* qui est la partie d'une vue complète cohérente). Comme une vue construit une configuration distribuée possible de l'application, l'exécution de l'application peut continuer à partir de ce point comme si aucune défaillance ne s'était produite.

Le projet MPICH-V [38] a pour but de comparer les performances de différents protocoles de tolérance aux fautes dans l'implantation de MPI : MPICH-1 [35]. Un des protocoles implanté est l'algorithme de Chandy-Lamport non bloquant optimisé. Il existe deux implantations possibles de l'algorithme de Chandy-Lamport : bloquante ou non bloquante. L'implantation bloquante utilise des marqueurs pour purger les canaux de communication et stopper les communications pendant une vague de checkpoint. Au contraire, l'implantation non bloquante laisse l'application continuer ses communications pendant une vague de checkpoint et enregistre les messages émis dans l'image de checkpoint. Nous allons, par la suite, décrire plus précisément le protocole et son implantation, mais il faut noter qu'un processus MPI est implanté par deux composants séparés pour permettre l'enregistrement des messsages en transit. Ces deux composants sont exécutés dans des processus unix séparés : un processus de calcul (MPI) et un processus de communication (daemon). Le processus de communication est utilisé pour enregistrer les messages en transit et pour rejouer ces messages quand un relancement du processus est exécuté.

Le fonctionnement du protocole est décrit dans la figure 7.16. Le processus MPI 1

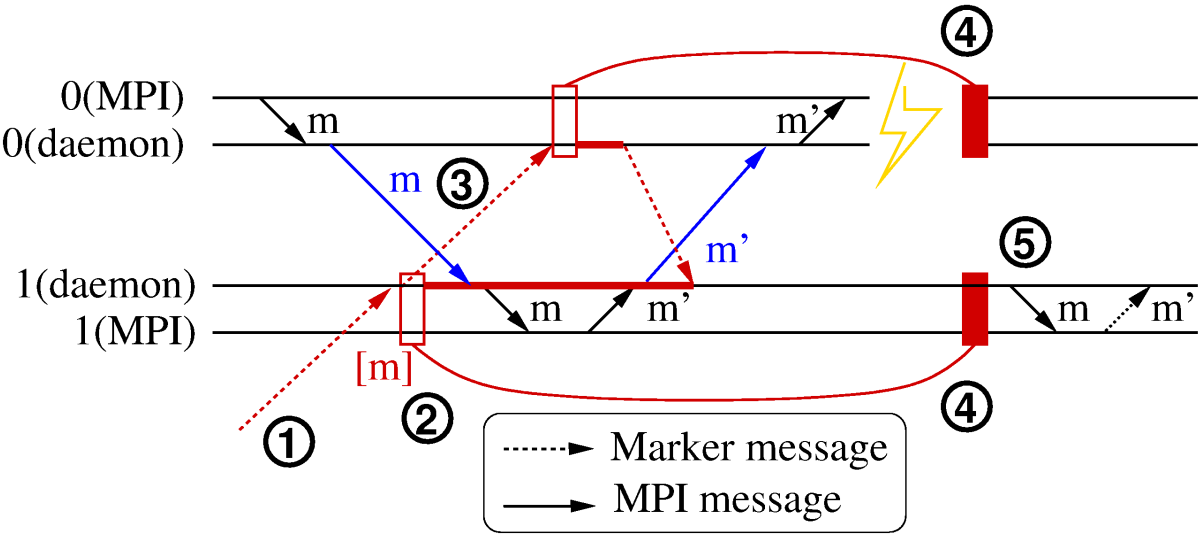

FIG. 7.16 – Exécution d'une application MPI sur MPICH-Vcl avec une faute

reçoit initialement le marqueur du *checkpoint scheduler* (1), enregistre son état local (2) et envoie un marqueur à tous les autres processus (2). A partir de cet instant, tous les messages (représentés par un m dans la figure) reçus après le checkpoint local mais avant d'avoir reçu le marker de leur expéditeur, sont enregistrés par le processus *daemon*. Quand le processus MPI 0 reçoit le marqueur, il commence son checkpoint local et envoie un marqueur à tous les autres processus (3). La réception de ce marqueur par le processus 1 marque la complétion de son checkpoint local. Si une défaillance se produit, tous les processus sont relancés à partir de leur checkpoint le plus récent (4) et le processus *daemon* rejoue l'envoie des messages enregistrés (5). On peut noter que le message m' peut ne pas être réenvoyé dans la nouvelle exécution.

MPICH implante une bibliothèque complète MPI à partir d'un canal. Un tel canal implante les routines de communication de base pour un matériel specifique ou pour un nouveau protocole de communication. MPICH-V est un framework générique pour comparer différents protocoles de tolérance aux fautes pour des applications MPI. Ce framework implante un canal pour la bibliothèque MPICH 1.2.7, basé sur le canal par défaut ch_p4.

MPICH-V est composé d'un ensemble de composants et d'un canal appelé ch_v. Ce canal s'appuie sur une séparation entre l'application MPI et le système de communication effectif. Les démons de communication (Vdaemon) fournissent toutes les routines de communication entre les différents composants de MPICH-V. La tolérance aux fautes est assurée en implantant des *hooks* dans les routines de communication. Cet ensemble de

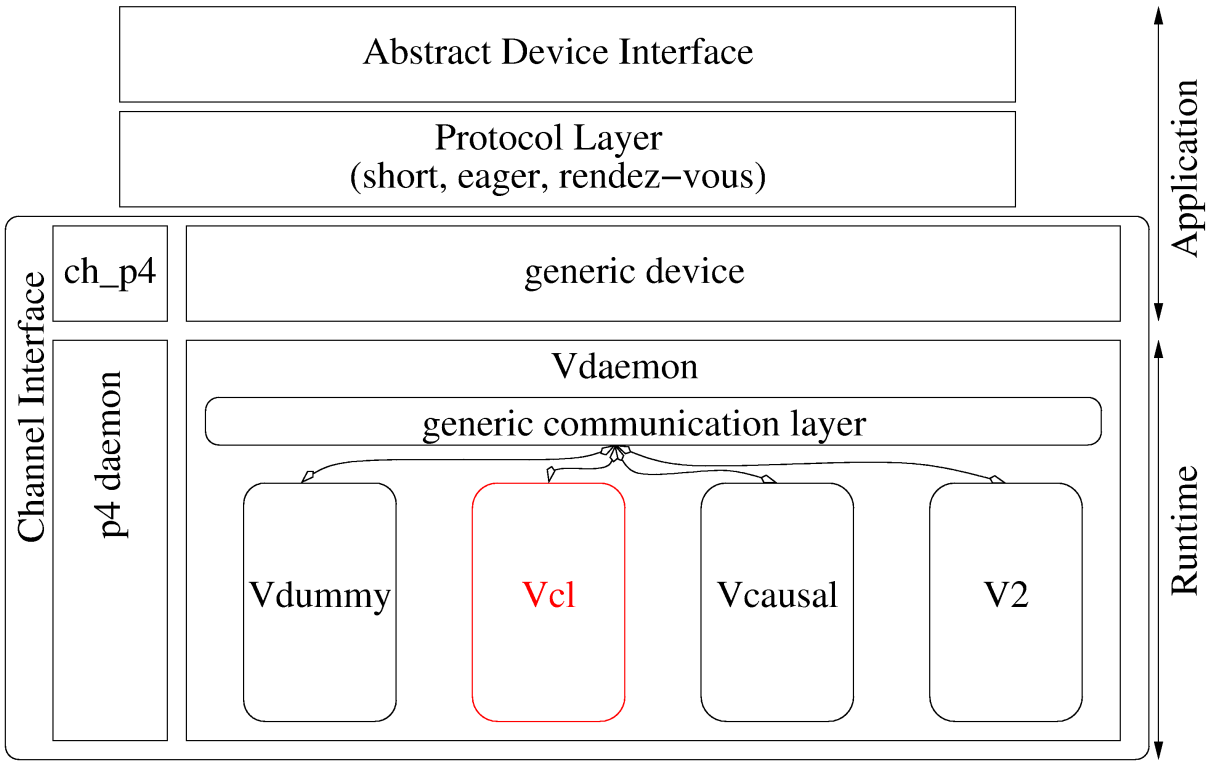

FIG. 7.17 – Architecture de MPICH-V

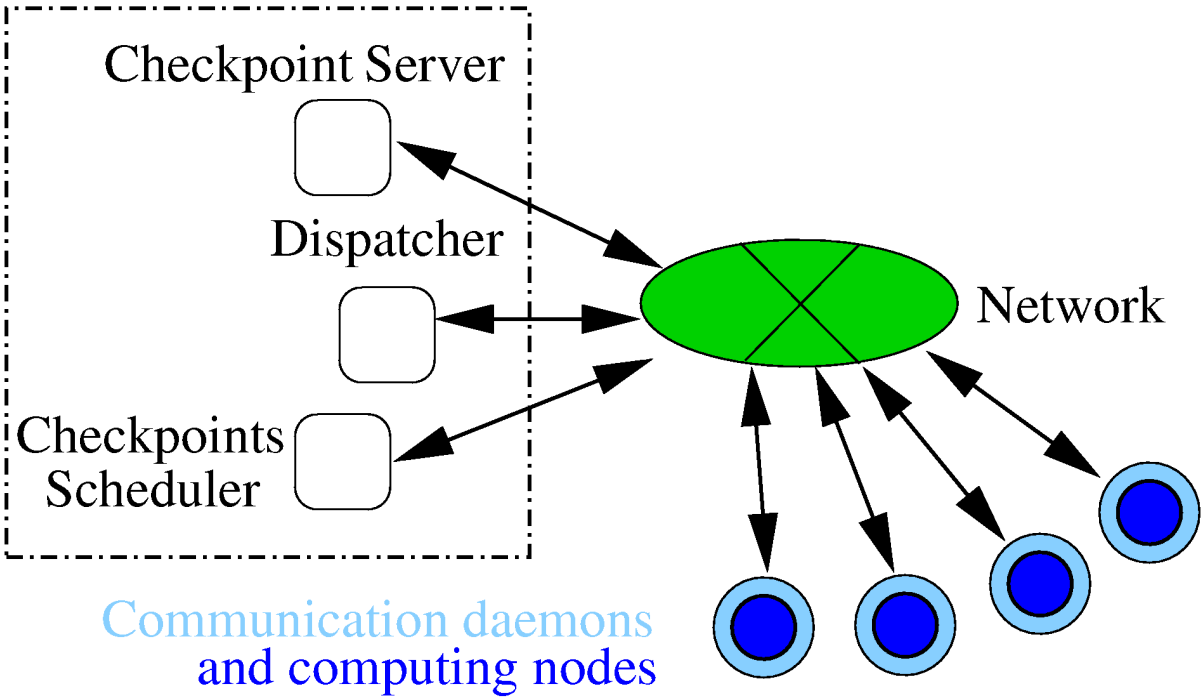

FIG. 7.18 – Déploiement typique de l'environnement MPICH-Vcl

hooks est appelé un `V-protocol`. Le `V-protocol` nous intéressant est Vcl, qui est une implantation non bloquante de l'algorithme de Chandy-Lamport.

Daemon Un démon gère les communications entre les noeuds (*i.e* envoi, réception, réordonnancement de messages et établissement des connexions). Il ouvre une socket TCP par processus MPI et une socket par type de serveur (le dispatcher et un serveur de checkpoint pour l'implantation Vcl). Il est implanté en tant que processus mono-thread qui multiplexe les communications à travers des appels à *select*. Pour limiter le nombre d'appels systèmes, toutes les communications sont empaquetées en utilisant des techniques *iovec*. La communication avec le processus MPI local est faite en utilisant des envois bloquants et des réceptions sur une socket Unix.

Dispatcher Le dispatcher est responsable du lancement de l'application MPI. Il lance les différents processus et les serveurs en premier, ensuite les processus MPI, en utilisant *ssh* pour lancer les processus distants. Le dispatcher est aussi responsable de la détection de défaillances et du relancement des noeuds. Une défaillance est supposée s'être produite après toute fermeture de socket non prévue.

La détection de défaillances s'appuie sur le paramètre *TCP keep-live* du système d'exploitation. Les configurations Linux typiques définissent la détection d'une défaillance après 9 pertes consécutives de sondes keep-alive, où les sondes keep-alive sont prévues toutes les 75 secondes. Ces paramètres peuvent être changés pour fournir plus de réactivité lorsque des défaillances du système se produisent. Dans les expériences, les fautes sont émulées en interrompant une tâche, et non pas le système d'exploitation, ce qui implique que la détection de défaillances est déclenchée par la fermeture d'une connexion, qui est effectuée dès que la tâche est interrompue par le système d'exploitation.

Serveur de checkpoint et mécanisme de checkpoint Les deux implantations utilisent le même mécanisme abstrait de checkpointing. Ce mécanisme fournit une API unifiée pour adresser trois biliothèques de checkpointing de tâches système : Condor Standalone Checkpointing Library [53], libckpt [86] et the Berkeley Linux Checkpoint/Restart [47, 67]. Toutes ces bibliothèques peuvent donc être utilisées pour la prise d'une image d'un processus unix, son enregistrement sur un disque et le relancement de ce processus sur une même architecture. Par défault, BLCR, qui est la bibliothèque la plus à jour, est utilisée.

Les serveurs de checkpoint sont responsables de la collecte des checkpoints locaux de tous les processus MPI. Quand un processus MPI démarre un checkpoint, il duplique

son état en faisant un appel à la primitive système *fork*. Le processus dupliqué utilise la bibliothèque de checkpoint pour créer le fichier de checkpoint pendant que le processus MPI initial peut continuer son exécution. Le démon associé avec le processus MPI se connecte au serveur de checkpoint et lui transmet, à travers un algorithme producteur/consommateur, le fichier de checkpoint généré par le processus fils. Quand le fichier de checkpoint a été complètement envoyé, le fils du processus MPI se termine et le démon envoie alors tous les messages, devant être enregistrés suivant l'algorithme de Chandy Lamport, qui ont été temporairement stockés sur une memoire volatile du démon. En utilisant cette technique, le calcul global n'est jamais interrompu pendant une phase de checkpoint.

Quand un checkpoint global est complété, il n'est pas nécessaire de maintenir le stockage des anciens checkpoints globaux. Ainsi, les serveurs de checkpoint ne stockent seulement qu'un checkpoint global complet à la fois, stockant alors au plus deux images de checkpoint par noeud de calcul.

Si une défaillance intervient, tous les processus MPI se relancent à partir du checkpoint local stocker sur le disque s'il existe, sinon ils l'obtiennent grâce au serveur de checkpoint.

Ordonnanceur de Checkpoint L'ordonnanceur de checkpoint gère les différentes vagues de checkpoint. Il envoie regulièrement des marqueurs à tous les processus MPI. La fréquence des checkpoints est un paramètre défini par l'utilisateur. Il attend alors un accusé de réception de fin de checkpoint de tous les processus MPI avant de confirmer la fin du checkpoint global aux serveurs de checkpoint. L'ordonnanceur de checkpoint commence une nouvelle vague de checkpoint seulement après la fin de la précédente.

7.3.2 FAIL-MPI : les applications auto-déployantes

Dans les versions de FAIL-FCI [41] qui précédaient les tests que nous allons présenter, on faisait plusieurs suppositions sur l'application distribuée qui ne correspondent pas au mécanisme auto-déployant de MPICH-V. En particulier, il était supposé qu'un mécanisme de type *ssh* pouvait être systématiquement utilisé pour lancer un processus de l'application distribuée et que les processus n'arrêtaient jamais leur exécution jusqu'à ce qu'une faute soit injectée. Alors, le middleware FAIL-FCI était capable de contrôler le lancement, l'arrêt, la reprise et l'interruption définitive des processus en s'interfaçant avec GDB. Comme MPICH-V est aussi un middleware qui s'auto-déploie sur plusieurs noeuds, il n'était pas possible de déployer MPICH-V à travers FAIL-FCI.

Pour contourner ce problème, nous avons développé un nouveau schéma d'intégration

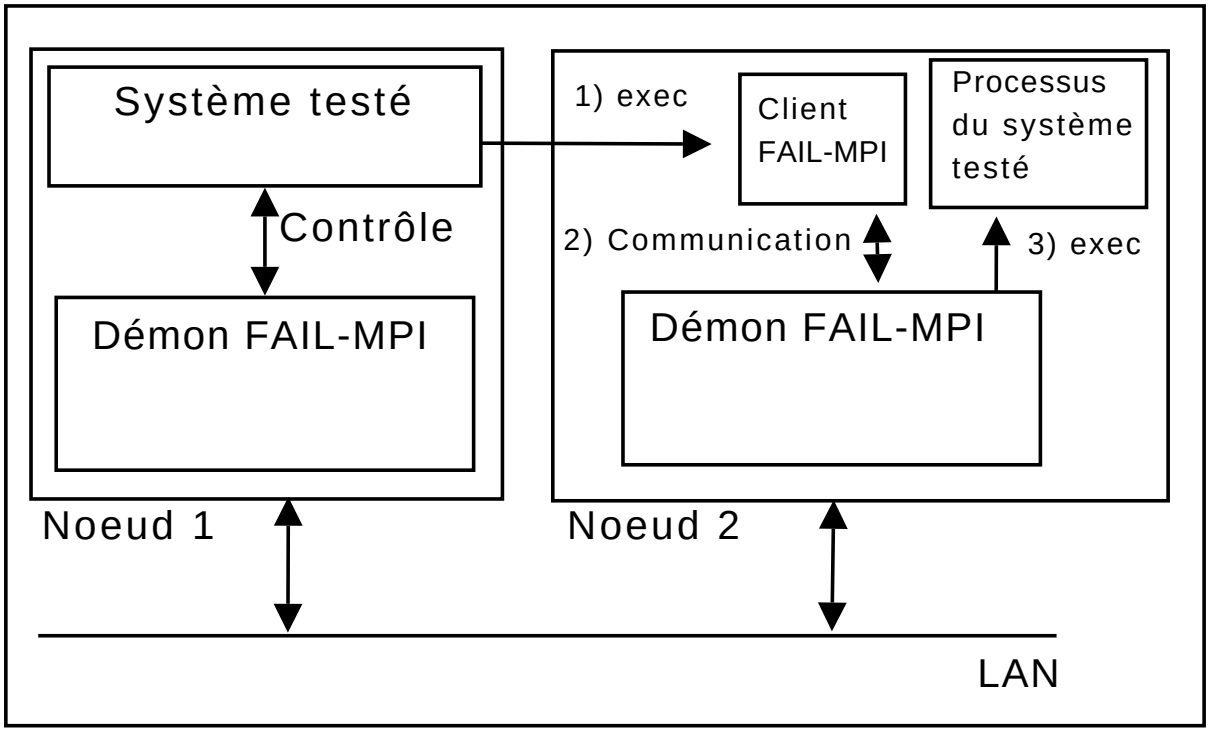

FIG. 7.19 – Schéma d'intégration de FAIL-MPI

pour l'injection de fautes qui peut être utilisé avec n'importe quelle application distribuée auto-déployante : FAIL-MPI (voir figure 7.19). La différence majeure entre FAIL-FCI et FAIL-MPI est que l'ancienne version ne nécessitait aucun changement de l'application testée, tandis que la nouvelle version fournit une interface pour les applications auto-déployantes, qui doit être utilisée pour supporter l'injection de fautes. Ne pas utiliser cette interface signifie que l'application auto-déployante ne souffrira d'aucune faute, tandis qu'en utilisant l'interface le scenario de fautes qui était défini sera utilisé, comme dans le middleware FAIL-FCI. Le schéma d'intégration est le suivant : au lieu de simplement lancer un nouveau processus en exécutant une ligne de commande, l'application auto-déployante s'enregistre avec un démon FAIL-MPI. Bien sûr, le démon FAIL-MPI est supposé déjà être exécuté sur la machine. Il est possible d'automatiser ce schéma pour que le changement dans le code source soit quasiment transparent, par exemple, en ayant la ligne de commande remplacée par un appel au démon FAIL-MPI en lui passant en argument la ligne de commande à exécuter. Ainsi, le démon FAIL-MPI gèrera l'application spécifiée par la ligne de commande à travers le débogueur pour injecter les fautes.

Une autre nouvelle caractéristique de FAIL-MPI est son aptitude à s'attacher à un processus déjà exécuté, pour que les processus qui n'étaient pas créés à partir d'une ligne de commande (comme ceux obtenus par un appel à la primitive `fork`) puissent aussi être

utilisés dans le framework FAIL-MPI. Cela requiert simplement de s'enregistrer avec le démon FAIL-MPI en utilisant l'identifiant du processus en argument pour qu'il puisse s'attacher au processus exécuté.

En utilisant ces deux extensions, il est possible que l'application testée ne soit pas lancée par le middleware FAIL-MPI (*i.e.* cela se passe dans les systèmes pair-à-pair ou les *desktop grid*, où les nouveaux utilisateurs sont supposés joindre le système). Un scénario de fautes pourrait typiquement avoir besoin de faire des actions quand un nouveau processus joint ou quitte le système. Une raison évidente pour cela est que l'on voudrait injecter des fautes seulement sur des processus qui sont effectivement exécutés. De plus, on pourrait être intéressé par ce qui se passe lorsqu'un processus s'interrompt et qu'un autre processus joint le système simultanément. Suite aux observations précédentes, nous avons intégré trois nouveaux déclencheurs FAIL (*i.e.* modèle pour les événements systèmes) dans le langage FAIL :

1. **onload** : ce déclencheur arrive quand un processus joint l'application distribuée testée,
2. **onexit** : ce déclencheur arrive quand un processus de l'application distribuée testée se termine correctement,
3. **onerror** : ce déclencheur arrive quand un processus se termine *de manière incorrecte.*

Par la suite, toutes les fonctionnalités développées dans FAIL-MPI ont été intégrées à FAIL-FCI pour obtenir un outil générique complet.

7.3.3 Evaluation de performance

Dans cette section, nous avons mis en œuvre l'implantation MPICH-Vcl avec plusieurs motifs de fautes et analysé les performances observées. Pour toutes les mesures, nous avons utilisé le benchmark NAS parallèle BT (Block Tridiagonal) [8] de classe B en faisant varier le nombre de nœuds participants. Ce benchmark fournit des schémas de communication complexes appropriés pour tester la tolérance aux fautes. La class B assure une écriture moyenne vers la mémoire et permet d'obtenir des mesures en un temps raisonnable.

Nous avons lancé les expériences sur GriDeXplorer en déployant un système d'exploitation Linux, avec une version 2.6.13-5 du noyau, incluant le module BLCR version 0.4.2 pour la prise de checkpoint. Nous avons compilé les benchmarks avec g77 de la version FSF 4.1.0 et l'option d'optimization habituelle (`-O3`).

Pour implanter les différents scénarios de fautes, nous avons utilisé une approche centralisée. Nous avons conçu un démon spécifique FAIL-MPI en utilisant le langage

FAIL pour coordonner l'injection de fautes pendant que les autres démons FAIL-MPI contrôlaient l'exécution des nœuds de calcul MPI. Le démon FAIL-MPI specifique est nommé "P1" par la suite. Pour toutes les mesures, P1 choisit un processus sujet aux fautes suivant le scenario de fautes et envoie un ordre de défaillance au démon FAIL-MPI le contrôlant. Quand un démon reçoit un ordre de défaillance, soit un nœud MPI est effectivement exécuté sur la machine correspondante, soit aucun nœud n'y est exécuté. S'il y a un tel nœud MPI, la faute est injectée et P1 reçoit un acquittement ; s'il n'y a aucun nœud MPI, P1 reçoit un acquittement négatif et peut choisir un autre nœud où injecter la faute.

Plus formellement, le scénario FAIL utilisé pour contrôler tous les nœuds de calcul MPI, est donné dans la figure 7.20. Chaque démon FAIL-MPI est constitué de deux états ; dans l'état 1, il attend l'événement `onload` pour aller dans l'état 2. Si un ordre de crash est reçu (line 2), un acquittement négatif est renvoyé et le démon reste dans l'état 1. Dans l'état 2, si le nœud MPI se termine normalement ou pour une autre raison quelconque, il retourne dans l'état 1 et attend qu'un nouveau nœud MPI s'enregistre. Quand un ordre de crash (line 6) est reçu, il renvoie un acquittement, arrête les processus et retourne à l'état 1.

```
Daemon ADV2 {
node 1 :
1    onload -> continue, goto 2;
2    ?crash ->!no(P1), goto 1;
node 2 :
3    onexit -> goto 1;
4    onerror-> goto 1;
5    onload -> continue, goto 2;
6    ?crash ->!ok(P1), halt, goto 1;
}
```

FIG. 7.20 – Scénario FAIL-MPI pour tous les nœuds de calcul MPI

Quelques unes des expériences introduisent trop de stress pour l'application et la bibliothèque de tolérance n'est pas capable de tolérer ce niveau de stress. Alors, l'application se gèle ou entre dans un cycle de rollback / crash. Si l'application boucle dans une série de rollback et crashes, cela dénote un niveau d'injection de fautes trop grand pour obtenir une quelconque progression du calcul. Si l'application est gelée, cela dénote un bogue

dans l'implantation. Pour détecter les deux cas, nous avons introduit un timeout pour toutes nos expériences. Après 1500 secondes, tous les composants de l'application (incluant les serveurs de checkpoints, l'ordonnanceur et le dispatcher) sont tués et l'expérience est marquée comme non terminée.

Dans toutes les expériences, nous avons distingué les expériences ne progressant plus à cause de la fréquence de fautes trop élevée (quand la fréquence de fautes est trop élevée, l'application n'a pas le temps d'atteindre la nouvelle vague de checkpoint avant d'être frappée par une nouvelle faute, ainsi elle ne peut plus progresser et semble être arrétée) des expériences ne progressant plus à cause d'un bogue dans l'implantation de la tolérance aux fautes. La différenciation de ces deux types d'expériences est faite par l'analyse des traces des différentes exécutions. Dans les figures suivantes, nous présentons le pourcentage des exécutions ne progressant plus avec des barres vertes et le pourcentage des exécutions « boguées » avec des barres rouges.

7.3.4 Impact de la fréquence des fautes

```
Daemon ADV1 {
node 1 :
1   always int ran = FAIL_RANDOM(0,N);
2   time_g timer = X;
3   timer ->!crash(G1[ran]), goto 2;
node 2 :
4   always int ran = FAIL_RANDOM(0,N);
5   ?ok   -> goto 1;
6   ?no   ->!crash(G1[ran]), goto 2;
}
```

FIG. 7.21 – Impact de la fréquence des fautes : scenario pour P1

Ces tests ont pour but de présenter l'impact de la fréquence des fautes sur les performances du benchmark BT de classe B pour 49 processus. 53 machines étaient dédiées à cette expérience, assurant ainsi que suffisamment de processeurs en réserve étaient trouvés dans tous les scénarios.

Le scénario, présenté dans la figure 7.21, injecte des fautes à une fréquence donnée. L'algorithme pour le démon P1 est le suivant : un processus est tout d'abord choisi

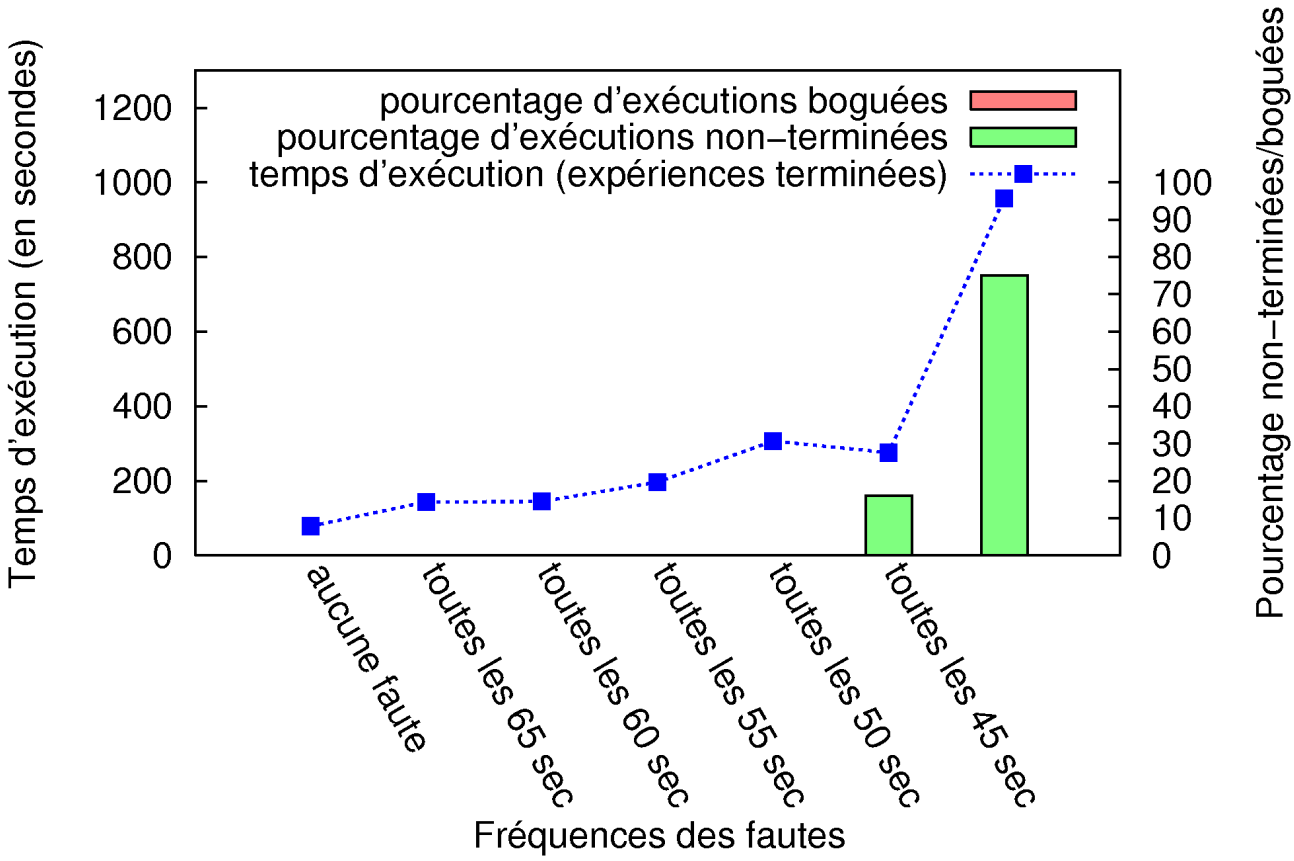

FIG. 7.22 – Impact de la fréquence des fautes : résultats

aléatoirement (de manière uniforme) (line 1), alors un timeout est programmé (line 2). Après expiration du timeout (line 3), l'ordre d'injection de fautes est envoyé au processus choisi. Si un acquittement est reçu, un nouveau processus est choisi aléatoirement et le cycle continue (line 5). Si un acquittement négatif est reçu (dénotant qu'aucun processus MPI n'est exécuté sur le même nœud que le processus choisi, ce qui peut se produire à cause d'une injection de fautes précédente), un autre processus est immédiatement choisi (line 4) et un ordre d'injection de fautes est envoyé à ce nouveau processus (line 6), jusqu'a ce qu'un acquittement soit éventuellement reçu.

Les mesures de performance sont présentées dans la figure 7.22. Les lignes en pointillées représentent le temps total d'éxecution du benchmark en fonction de la fréquence des fautes ; le pourcentage d'expériences non terminées en fonction du même paramètre est représenté en utilisant des barres vertes et le pourcentage d'expériences boguées est representé en utilisant des barres rouges. Toutes les experiences ont été exécutées 6 fois

et les valeurs présentées ici sont une moyenne des mesures obtenues.

Comme le décrit le scénario, aucune faute en cascade n'a été injectée dans cette expérience, et nous pouvons voir qu'il n'y a pas d'exécutions boguées. A partir d'un certain point, le temps entre deux fautes est inférieur au temps entre deux checkpoints, et lorsque cela arrive, l'application n'a pas le temps de progresser. A partir de ce point, des exécutions non terminées apparaissent, et augmentent avec la fréquence de fautes, jusqu'à un point où aucune execution ne se termine. De manière similaire, le mécanisme de rollback/recovery prend une part croissante du temps total d'exécution quand le nombre de fautes par minute augmente, et nous remarquons (avec la ligne pleine) que le temps d'exécution croît avec le nombre de fautes par minute. Ceci est partiellement contradictoire avec les mesures effectués pour une faute toutes les 45 secondes. Une analyse plus approfondie des traces de l'expérience démontre que, pour cette fréquence de fautes, les défaillances apparaissent juste après les vagues de checkpoint (cela est dû à la fréquence des vagues de checkpoint qui est configurée à une vague de checkpoint toutes les 30 secondes). Quand des défaillances apparaissent juste après une vague de checkpoint, le mécanisme de rollback/recovery est le plus efficace.

7.3.5 Impact de l'échelle

La figure 7.23 présente l'impact de l'échelle sur les performances du benchmark BT de classe B pour une fréquence donnée d'injection de fautes. Dans cette expérience, une faute est injectée toutes les 50 secondes et le nombre de nœuds exécutant l'application varie de 25 à 64 (BT a besoin d'un nombre carré de nœuds). Le scénario FAIL est le même que celui utilisé dans l'expérience précédente (voir la figure 7.21). Chaque expérience a été exécutée 5 fois et la moyenne des valeurs obtenues est présentée ici.

Avec 25 nœuds, une des cinq expériences ne s'est pas terminée. Toutes les expériences sont exécutées avec le même nombre de serveurs de checkpoint. Comme BT utilise un nombre approximativement constant de mémoire divisé de façon égale entre les nœuds de calcul, pour 25 nœuds, la taille de chaque image de checkpoint est plus grande que pour 36 nœuds ou plus. Donc chaque transfert d'image de checkpoint individuel prend plus de temps avec 25 nœuds que pour les autres tailles. Cela implique que le temps de checkpoint et de reprise est plus long pour 25 nœuds que pour les autres tailles. Pour une des expériences, la vague de checkpoint était synchronisée (par chance) avec l'injection de fautes (toutes les 50 secondes). Ainsi, il n'y avait plus aucune progression. Quand la taille de l'image de checkpoint décroît, et donc également le temps de checkpoint et reprise, ce phénomène apparaît avec une probabilité plus faible.

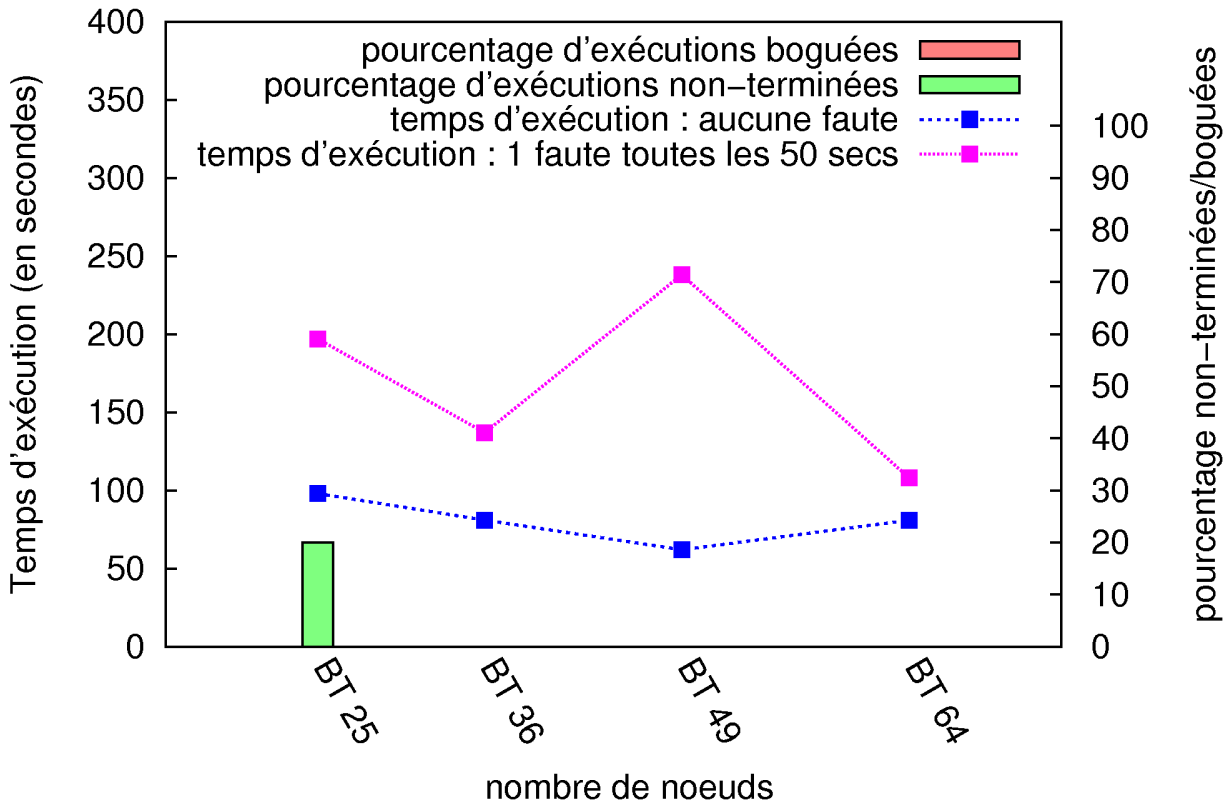

FIG. 7.23 – Impact de l'échelle

Le temps total d'exécution mesuré pour les expériences avec une faute injectée toutes les 50 secondes pour différentes tailles est apparemment chaotique. Il peut être très près du temps mesuré sans injection de fautes (pour 64 nœuds) ou jusqu'a 2.5 fois celui-ci. En effet, une analyse approfondie des mesures obtenues démontre que la variance de ces mesures croît avec le nombre de nœuds, et les valeurs moyennes présentées ici ne sont pas significatives. Des injections de fautes à des périodes régulières indépendantes du protocole, et des injections de fautes juste avant une vague de checkpoint ont un impact majeur sur les performances. Cependant, des injections de fautes juste après une vague de checkpoint ont un impact négligeable.

Afin de mesurer avec précision l'effet de la variation de ce paramètre, le nombre d'experiences à exécuter doit être plus important. Une autre solution est de mesurer précisément la date de l'injection de fautes comparée à celle de la dernière vague de checkpoint, et de mesurer l'impact de ce délai sur le temps total d'exécution. Cependant, cela requiert aussi la capacité de lire une variable du programme testé, ce qui est une fonctionnalité envisagée de FAIL-MPI, mais qui n'est pas encore implantée.

7.3.6 Impact des fautes simultanées

Dans ces tests, des fautes simultanées vont être injectées toutes les 50 secondes et leur impact sur les performances du benchmark BT de class B pour 49 processus va être observé . Ces mesures nous permettent d'obtenir un test de la tolérance aux fautes pour une implantation de MPI tolérante aux fautes avec le cas rare d'apparition de fautes simultanées. Le scénario de fautes est formellement décrit dans la figure 7.24.

Le scénario est une simple modification du précédent. A l'expiration d'un timeout, pas un, mais X (X correspond à l'abscisse sur la figure) processus sont choisis, un après l'autre, pour être victime d'une faute. Toutes les fois où le maître entre dans le nœud 1, il sélectionne uniformément un processus (line 2), ensuite à l'expiration du timer (line 4) il envoie un ordre de crash aux processus sélectionnés et entre dans le nœud 2. Toutes les fois où le processus entre dans le nœud 2, il sélectionne un autre processus (line 5). Dans le nœud 2, s'il reçoit un accusé de réception positif et s'il y a toujours des fautes à injecter (line 6), il envoie un ordre de crash aux processus sélectionnés, décrémente le nombre de fautes à injecter et entre encore une fois dans le nœud 2. Si un accusé de réception négatif est reçu (line 8), un autre processus est sélectionné en entrant encore une fois dans le nœud 2. Quand toutes les fautes ont été injectées avec succès (line 7), le nombre de fautes à injecter la prochaine fois est remis à X et P1 entre à nouveau dans le nœud 1, programmant ainsi un nouveau timeout.

```
Daemon ADV1 {
1  int nb_crash = X ;
node 1 :
2  always int ran = FAIL_RANDOM(0,52) ;
3  time_g timer = 50 ;
4  timer ->!crash(G1[ran]), goto 2 ;
node 2 :
5  always int ran = FAIL_RANDOM(0,52) ;
6  ?ok && nb_crash > 1 ->
     !crash(G1[ran]), nb_crash = nb_crash - 1, goto 2 ;
7  ?ok && nb_crash <= 1 -> nb_crash = X, goto 1 ;
8  ?no ->!crash(G1[ran]), goto 2 ;
}
```

FIG. 7.24 – Impact des fautes simultanées : scénario pour P1

La figure 7.25 présente le pourcentage d'expériences boguées et la moyenne du temps total d'exécution pour 6 experiences. On peut voir que pour une injection de 5 ou 6 fautes simultanées toutes les 50s, un tiers des expériences ont un comportement bogué. Une analyse complète des traces des exécutions démontre que toutes ces exécutions étaient gelées durant la phase de reprise après une injection de fautes. Ce phénomène n'apparaît pas spontanément avec moins de fautes simultanées, et une majorité des exécutions n'étaient pas sujettes à ce comportement, même avec de nombreuses phases de checkpoint et reprise (une vague de checkpoint est déclenchée toutes les 30 secondes, ainsi il y avait une moyenne de 6 à 7 vagues de checkpoint avec une injection de fautes toutes les 50 secondes, donc approximativement 4 fautes entraînant une reprise par exécution).

7.3.7 Chasse aux Bogues en utilisant FAIL-MPI

Afin de localiser précisément le bogue de MPICH-Vcl avec FAIL-MPI, nous avons exécuté un ensemble d'expériences visant à définir plus precisément la configuration entraînant ce comportement. Comme les traces des exécutions suggéraient que le bogue se manifestait lors de la reprise après la détection d'une faute, nous avons tout d'abord construit un scénario pour injecter des fautes pendant ce moment. Le scénario que nous avons utilisé est formellement donné par la figure 7.26.

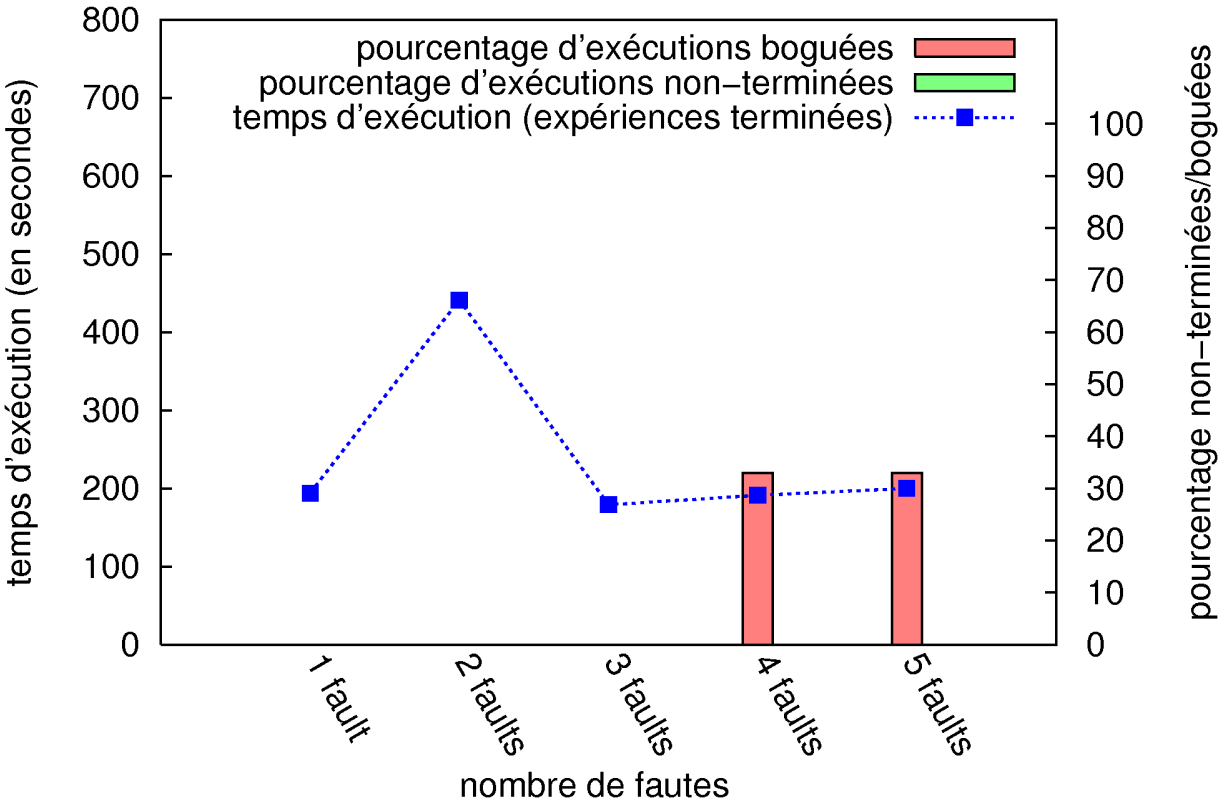

FIG. 7.25 – Impact des fautes simultanées : résultats

A cause de la complexité du scénario, nous avons modifié le scénario générique utilisé par les nœuds de calcul. Ils doivent maintenant compter le nombre de vague de restart, et envoyer un message *waveok* au processus de contrôl (P1) pour synchroniser l'injection de fautes. De manière générale, l'algorithme pour le démon P1 est pratiquement le même que celui utilisé pour les expériences de l'étude de l'impact de la fréquence de l'injection de fautes. Les lignes 1 à 6 sont utilisées pour assurer que la première faute est réellement injectée, car nous avons alloué plus de machines que nécessaires pour l'expérience, afin d'avoir des nœuds de réserve. Un nœud participant peut ne pas exécuter un démon MPI, et l'accusé de réception positif doit être reçu pour assurer qu'une faute a, en effet, été injectée. Ensuite, le démon P1 attend un message de la part d'un démon FAIL-MPI exécuté sur un nœud de calcul ce qui dénote le début d'une vague de reprise, et lui envoie un ordre de crash (line 7). Après avoir envoyé cet ordre, il se place dans un état vide (node 4) et n'injecte plus aucune autre faute pendant l'exécution de l'expérience.

Les performances mesurées sont présentées dans la figure 7.27. Nous avons noté que

```
Daemon ADV1 {
node 1 :
1  always int ran = FAIL_RANDOM(0,52);
2  time_g timer = 50;
3  timer ->!crash(G1[ran]), goto 2;
node 2 :
4  always int ran = FAIL_RANDOM(0,52);
5  ?ok -> goto 3;
6  ?no ->!crash(G1[ran]), goto 2;
node 3 :
7  ?waveok ->!crash(FAIL_SENDER), goto 4;
node 4 :
}
```

FIG. 7.26 – Impact des fautes synchronisées : scénario pour P1

même si nous injectons seulement 2 fautes en utilisant ce scénario, pour toutes les échelles, quelques experiences ne se sont pas terminées à cause d'un bogue dans l'implantation de la tolérance aux fautes. Cela démontre que le bogue est localisé dans cette partie de l'exécution et n'est pas une conséquence de la taille de l'application. Cependant, une large majorité des exécutions n'est pas sujette au bogue. Nous avons donc défini plus précisément les conditions de l'injection de fautes. A cette étape de l'étude, nous soupçonnions que le bogue se produisait seulement si un processus était sujet aux fautes alors qu'il participait à une vague de reprise pendant qu'un des autres processus n'avait pas fini de terminer son execution suite à la détection d'une faute. Dans ce cas, le dispatcher MPICH-Vcl détecte la défaillance d'un processus dans la nouvelle vague. Il semble que si cela se produit, il n'arrive plus à discerner l'état des processus et lesquel devraient être relancés.

Afin de démontrer cela, nous avons considéré un dernier scénario de fautes. Dans cette expérience, nous voulions exécuter le même scénario que celui utilisé dans l'expérience précédente mais en utilisant l'état des démons MPI pour déterminer le moment où injecter la seconde faute. Comme nous voulions nous assurer que le dispatcher MPICH-Vcl n'arrive plus à distinguer l'état du processus de la vague de reprise, nous devions injecter la faute après que la connexion a été établie et que quelques unes des communications initiales ont été faites. Ainsi, la seconde faute a été injectée juste avant que le démon de communication

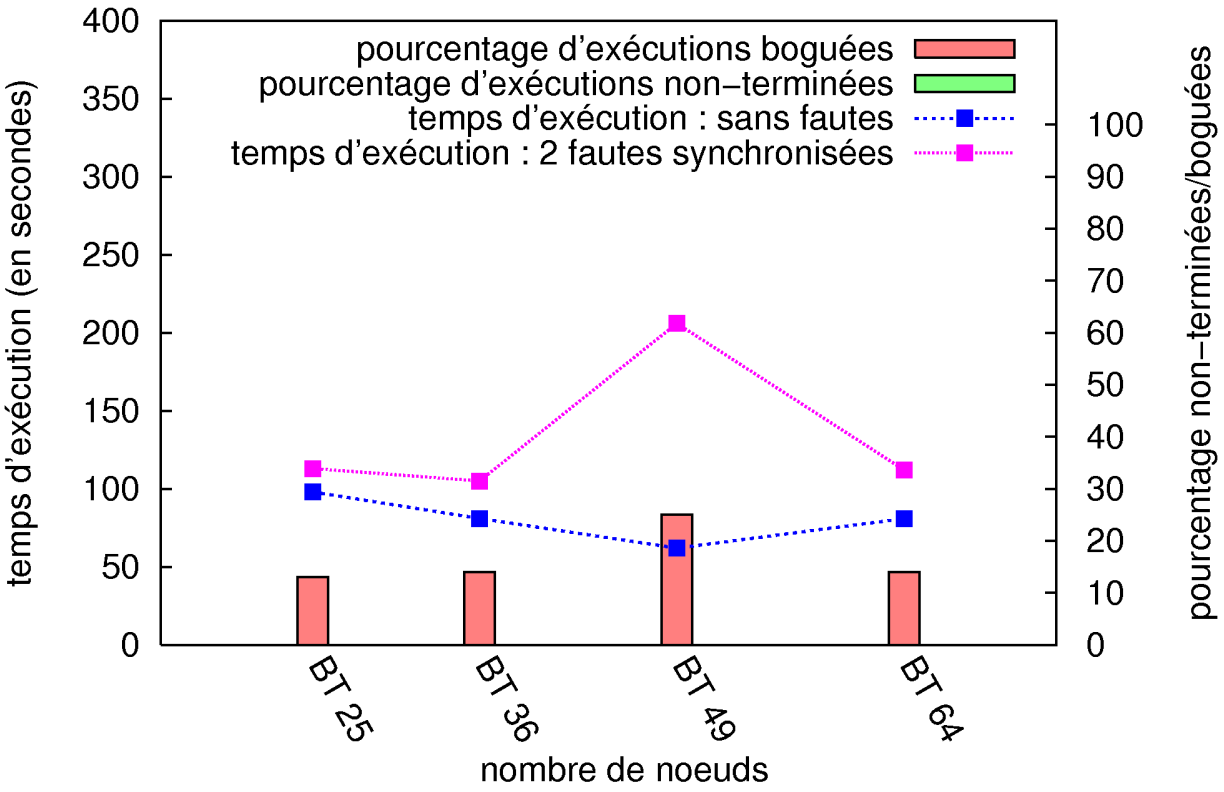

FIG. 7.27 – Impact des fautes synchronisées : résultats

MPI fasse appel à la fonction `localMPI_setCommand`. Cette fonction est appelée par le démon de communication après avoir échangé les arguments initiaux avec le dispatcher. Cela assure que le dispatcher considère ce démon comme correctement initialisé, ainsi il déclenchera le mécanisme de détection de fautes quand une faute sera injectée.

Le démon FAIL-MPI spécifique (P1) utilisé pour coordonner l'injection de fautes est décrit dans la figure 7.28. La seule modification comparée au scénario de fautes précédent est que lorsqu'il est dans le nœud 4, le processus P1 envoie un ordre *nocrash*, pour que les processus non destinés à être victimes d'une faute ne soient pas bloqués avant l'exécution de la fonction `localMPI_setCommand`. Le nœud 4 a été modifié, car après avoir envoyé un ordre de crash au premier démon de la vague de reprise, le démon P1 doit toujours attendre les messages de tous les autres démons de la deuxième vague (i.e vague de reprise) pour leur envoyer un message leur permettant de continuer leur exécution (line 8). Ainsi, seulement le processus ayant reçu l'ordre de crash (le premier de la vague de reprise a avoir contacté le dispatcher) sera victime d'une faute juste avant d'exécuter la fonction `localMPI_setCommand`.

```
Daemon ADV1 {
node 1 :
1   always int ran = FAIL_RANDOM(0,52);
2   time_g timer = 50;
3   timer ->!crash(G1[ran]), goto 2;
node 2 :
4   always int ran = FAIL_RANDOM(0,52);
5   ?ok -> goto 3;
6   ?no ->!crash(G1[ran]), goto 2;
node 3 :
7   ?waveok ->!crash(FAIL_SENDER), goto 4;
node 4 :
8   ?waveok ->!nocrash(FAIL_SENDER), goto 4;
}
```

FIG. 7.28 – Fautes synchronisées basées sur l'état de MPI : scénario pour P1

Les mesures obtenues sont présentées dans la figure 7.29. Dans tous les cas, toutes les expériences sont gelées pendant la vague de reprise. Ainsi, nous avons conclu que le bogue dans le processus de reprise illustré par l'injection de fautes simultanées apparaît quand le dispatcher détecte la défaillance d'un processus déjà dans une phase de reprise, alors qu'il y a toujours d'autres processus de la vague d'exécution précédente qui n'ont pas reçu l'ordre de terminaison.

7.3.8 Conclusion

Dans cette section, nous avons utilisé FAIL-MPI pour mettre sous tension l'implantation du protocole de Chandy-Lamport non bloquant MPICH-Vcl. Nous avons présenté les modifications apportées à la version précédente de l'outil d'injection de fautes FAIL-FCI. Nous avons ensuite évalué les propriétés de tolérance aux fautes de MPICH-Vcl grâce à plusieurs expériences utilisant le benchmark NAS BT dans le framework MPICH-Vcl.

FAIL-MPI est la première version de FAIL qui permet à l'application testée de dynamiquement lancer et stopper des processus pendant l'exécution. Pour cela, nous avons ajouté de nouveaux événements specifiques au langage FAIL (création, terminaison et interruption d'un processus). En utilisant cet outil, nous avons été capable de reproduire

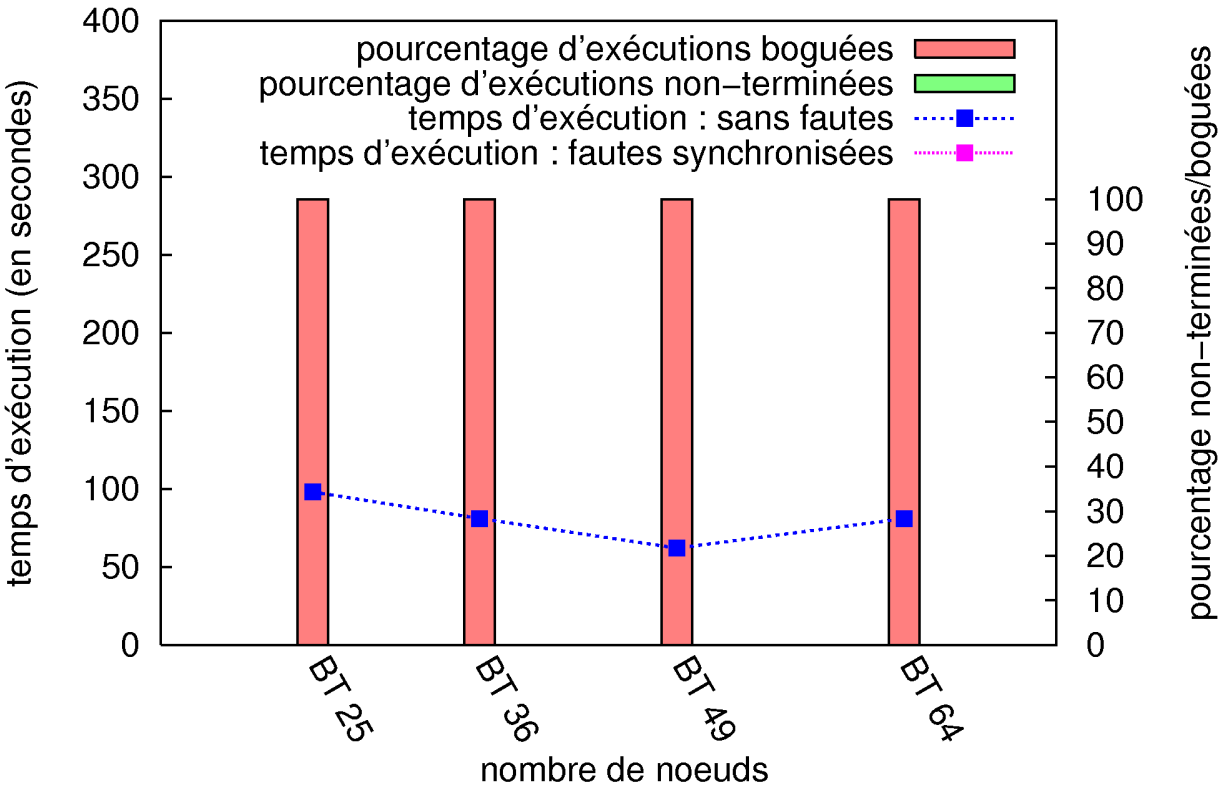

FIG. 7.29 – Fautes synchronisées basées sur l'état de MPI : résultats

automatiquement des mesures qui étaient auparavant faites manuellement, comme l'étude de l'impact de la fréquence des fautes sur le temps d'exécution [52]. Cela fournit l'opportunité d'évaluer plusieurs implantations différentes à des échelles importantes et de les comparer équitablement avec les mêmes scénarios de fautes.

Les mesures de temps d'exécution démontrent une grande variance dépendante de la différence entre la dernière vague de checkpoint et l'injection de fautes. Afin de fournir une variance plus faible des résultats, le langage FAIL et l'outil FAIL-MPI devrait être capable de lire et de modifier des variables internes de l'application testée.

Les dernières expériences de la section démontrent la puissance d'expressivité de FAIL-MPI permettant ainsi la localisation précise d'un bogue apparaissant très rarement dans le dispatcher de MPICH-Vcl. Nous avons été capable de démontrer que si une seconde faute frappe un processus faisant déjà partie d'une phase de reprise après qu'il se soit enregistré avec le dispatcher, et qu'au moins un autre processus de la vague précédente soit toujours en exécution, alors le dispatcher ne peut plus distinguer l'état de chaque processus et oublie de lancer au moins un nœud de calcul. Ce bogue, qui a été découvert

durant ces expériences, a été corrigé dans la nouvelle version de MPICH-Vcl.

FAIL-MPI est un outil adapté à la détection et l'isolation de bogues, ainsi qu'à l'évaluation de performance d'un système distribué complexe tolérant aux fautes sous des conditions précises d'apparition de fautes.

7.4 FreePastry

La popularité des services de partage de fichiers déployés sur Internet a récemment motivé considérablement la recherche sur les systèmes pair-à-pair qui s'est focalisée sur la conception de meilleurs algorithmes pair-à-pair et plus particulièrement dans le domaine de réseaux pair-à-pair structurés et des tables de hachages distribuées [57, 64, 77, 87], que nous allons simplement appeler DHTs. Ces systèmes font correspondre un grand espace d'identifiants sur l'ensemble des nœuds du système de façon déterministe et distribuée pour permettre le routage ou la recherche. Les DHTs effectuent généralement ces recherches en utilisant seulement $O(\log N)$ sauts dans un réseau de N nœuds où chaque nœud maintient seulement $O(\log N)$ lien vers un voisin, bien que des travaux récents ont exploré les changements induits par le stockage de plus ou moins de liens.

Pastry [65] est un substrat générique, passant à l'échelle et efficace, permettant l'écriture d'applications distribuées. Les nœuds Pastry forment un réseau pair-à-pair décentralisé, auto-organisé et tolérant aux fautes à travers Internet. Pastry fournit un routage des requêtes efficace, un mécanisme de localisation déterministe d'objets et une gestion de la charge indépendante de l'application en utilisant des DHTs. De plus, Pastry fournit un mécanisme qui supporte et facilite la réplication d'objets spécifiques à l'application, la gestion des caches et la reprise après défaillance.

FreePastry [32] est une implantation libre de Pastry visant un déploiement sur Internet.

Problèmes liés à l'environnement. Le principal problème pour une DHT est le phénomène intrinsèque aux systèmes distribués nommé « churn » : le processus perpétuel de l'arrivée et du départ des nœuds. En effet, ces systèmes sont caractérisés par un fort degré de « churn ». Une métrique du churn est le temps de session d'un nœud : le temps entre l'arrivée d'un nœud dans le réseau et son départ. Une autre métrique, liée au temps de session, est la fréquence des arrivées et des départs.

Plusieurs facteurs affectant le comportement des DHTs face à ce phénomène ont été définis. Les trois principaux facteurs sont les suivants :

- **La réparation réactive ou périodique après défaillance.** La réparation réactive est la stratégie où un nœud d'une DHT essaie de trouver un voisin de replacement immédiatement après avoir noté la défaillance de l'un de ses voisins. Il a été montré que, sous des conditions de limitation de la bande passante, la réparation réactive peut entraîner des cycles surchargeant le réseau, et causant une grande latence lors des *lookups* ou le retour de résultats erronés. D'un autre côté, un nœud d'une DHT peut réparer la défaillance d'un voisin à un temps fixé et périodique. Il a été montré que cette stratégie améliore les performances face au churn en permettant au système d'éviter les cycles.
- **Le calcul des timeouts des messages pendant le lookups.** La façon dont une DHT choisit les valeurs des timeouts pendant les lookups peut également grandement affecter ses performances face au churn. Si un nœud effectuant un lookup envoie un message à un nœud qui a quitté le réseaux, il doit, à un moment donné, abandonner sa requête et essayer un autre voisin. Il a été démontré que ces timeouts sont significatifs de la latence lors des lookup face au churn.
- **Le choix d'un voisin proche plutôt qu'un voisin distant.** La sélection d'un voisin proche (PNS) correspond au fait qu'un nœud d'une DHT essaie de selectionner les voisins les plus proches de lui en terme de latence lorsqu'un choix de voisin est nécessaire. Plusieurs algorithmes de découverte de voisins proches ont été définis et correspondent à différents compromis entre réduction de la latence et augmentation de la bande passante.

Le routage et la tolérance aux fautes dans Patry. Dans Pastry, les nœuds et les objets sont associés à des identifiants codés sur 128 bits. Ces identifiants sont appelés *nodeIds* et *clés* respectivement. Pastry fournit une primitive pour envoyer un message à une clé qui route le message au nœud vivant dont le nodeId est numériquement le plus proche de la clé.

La structure de routage maintenue par chaque nœud est constituée d'un ensemble de feuilles et d'une table de routage (voir figure 7.30). Chaque entrée de la structure de routage contient le nodeId et l'adresse IP d'un nœud. L'ensemble des feuilles contient les $l/2$ nodeIds voisins sur les deux côtés du nodesId du nœud local. Dans la table de routage, les nodeIds et les clés sont interprétés comme des entiers non-signés en base 2^b (où b est un paramètre dont la valeur par défaut est 4). La table de routage est une matrice de $128/b$ lignes sur 2^b colonnes. L'entrée de la ligne r et de la colonne c de la table de routage contient un nodeId qui partage ses r premiers chiffres avec le nodeId du nœud local, et a le $(r+1)^{ime}$ chiffre égal à c. Si il n'y a pas un tel nodeId ou si le nodeId local satisfait

cette contrainte, l'entrée reste vide. En moyenne, seulement $log_{2^b}N$ lignes ont des entrées non-vides.

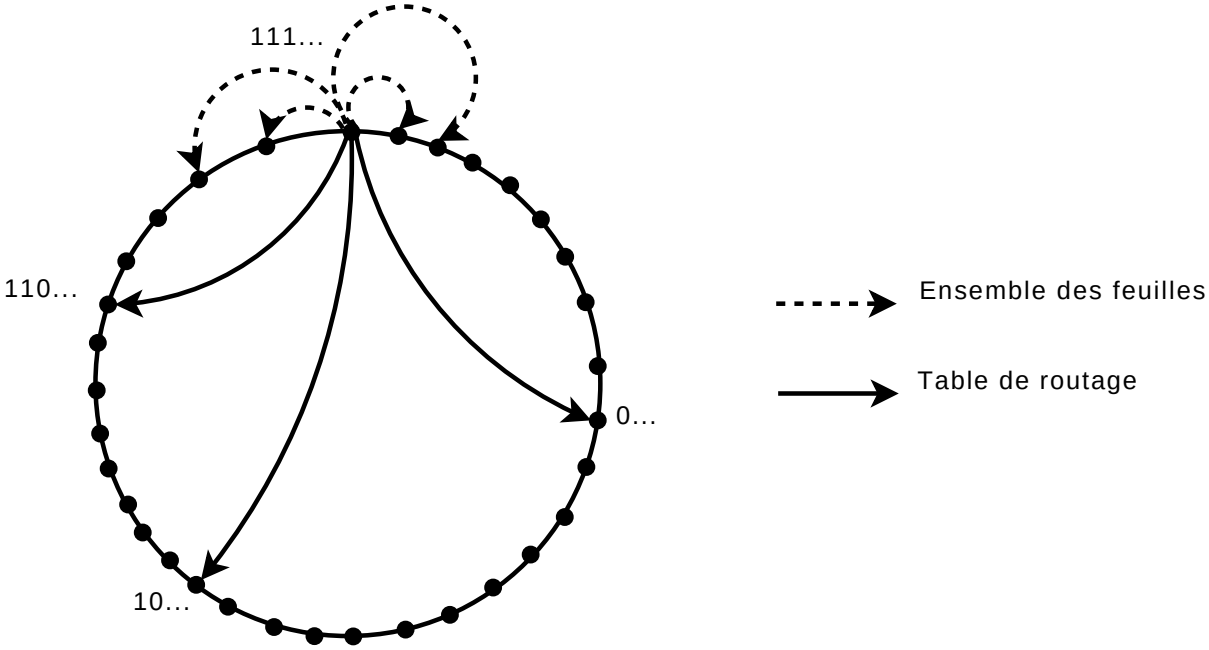

FIG. 7.30 – Le voisinage d'un nœud dans Pastry

Pastry route un message vers une clé en n'utilisant pas plus de $log2bN$ sauts en moyenne. A chaque saut, le nœud local transmet normallement le message à un nœud dont le nodeId partage, avec la clé, un préfixe qui est au moins un chiffre plus long que le préfixe que la clé partage avec le nodeId du nœud local. Si aucun de ces nœuds n'est connu, le message est transmis à un nœud dont le nodeId est numériquement plus proche de la clé et partage un préfixe avec la clé au moins aussi long. S'il n'existe pas de tel nœud, le message est délivré au nœud local.

Pastry met à jour la structure de routage quand les nœuds joignent et quittent le réseau. Les défaillances sont gérées de la façon suivante. Pastry utilise le sondage périodique pour la détection de défaillances. Chaque nœud envoie un message *keep-alive* aux membres de son ensemble de feuilles toutes les T_{ls} secondes. Puisque l'appartenance à un ensemble de feuilles est symétrique, chaque nœud devrait recevoir un message *keep-alive* de la part de chacun des membres de son ensemble de feuilles. Si il ne le fait pas, le nœud envoie une sonde explicite et suppose qu'un membre est mort si il ne reçoit pas une réponse au bout de T_{out} secondes. Additionnellement, chaque nœud envoie une sonde à chaque entrée de sa table de routage toutes les T_{rt} secondes. Puisque les tables de routage ne sont pas symétriques, les nœuds répondent à ces sondes. Si aucune réponse n'est reçue au bout de T_{out} secondes, une autre sonde est envoyée. Le nœud est supposé défaillant si

aucune réponse n'est reçue suite à la seconde sonde au bout de T_{out} secondes.

Les entrées sont retirées de la table de routage mais il est nécessaire de les remplacer avec d'autres nœuds. Il est suffisant de remplacer les entrées de l'ensemble des feuilles pour corriger l'état d'un nœud, mais il est important de remplacer les entrées de la table de routage pour obtenir un routage ayant un coût logarithmique. Le remplacement de l'ensemble des feuilles est effectué grâce aux informations sur les membres de l'ensemble des feuilles contenues dans les messages *keep-alive*. La maintenance de la table de routage est faite en demandant périodiquement à un nœud dans chaque ligne de la table de routage la ligne correspondante dans sa table de routage, et lorsqu'un emplacement de la table de routage est trouvé vide pendant le routage, le nœud du saut suivant est invité à retourner n'importe quelle entrée qu'il pourrait avoir pour cet emplacement. Ces mécanismes sont décrits avec plus de détail dans [18].

7.4.1 Scénario de fautes

En utilisant FAIL-FCI avec l'extension Java, notre but était de tester les capacités de tolérance aux fautes de FreePastry. En particulier, nous voulions graduellement injecter plus de fautes pour évaluer le point de rupture de l'implantation FreePastry, suivant différents critères. Contrairement aux approches précédentes [17] qui ont été faites par simulation, notre approche utilise l'application distribuée Java effective.

Pour cela, nous avons écrit, en utilisant FAIL, un scenario de fautes générique, où chaque nœud a la probabilité x de s'interrompre toutes les y secondes. Le code source FAIL est décrit dans la figure 7.31.

Nous allons maintenant décrire de manière non formelle le code source mentionné précédemment. Premièrement, deux automates sont définis : `ADV1` et `ADV2`. Ensuite, l'automate `ADV1` est associé à une machine `P1` (qui exécutera le code *dummy*), tandis que `ADV2` est associé à 30 machines (formant le groupe `G1`), chacune exécutant un fichier `.jar` avec les mêmes paramètres. `ADV2` s'exécute de la manière suivante : le démon attend tout d'abord que le programme se soit chargé, mais avant que la fonction *main* soit exécutée, le programme est stoppé. L'exécution du programme continue quand l'automate `ADV1` envoie le message *start*. Ensuite, toutes les cinq secondes, un événement de type *timer* est activé. Quand le timer expire, avec une probabilité de 0.1, le processus testé s'interrompt définitivement, tandis qu'avec une probabilité de 0.9, le processus continue son exécution.

```
Daemon ADV1 {
  node 1 : before(main) -> !start(G1), continue, goto 2 ;
  node 2 : }

Daemon ADV2 {
  node 1 : before(main) -> stop, goto 2 ;
  node 2 : ?start -> continue, goto 3 ;
  node 3 : always time_g timer = 5 ;
           always int random = FAIL_RANDOM(0,100) ;
           timer && random <= 10 -> halt, goto 4 ;
           timer && random > 10 -> continue, goto 3 ;
  node 4 : }

Computer P1 {
  program = "dummy" ;
  daemon = ADV1 ; }

Group G1 {
  size = 30 ;
  program = "-classpath FreePastry-1.4.1.jar :DHT_Simple.jar
             DistSimple 5000 lri7-209 5000" ;
  daemon = ADV2 ; }
```

FIG. 7.31 – Scénario de fautes pour FreePastry.

7.4.2 Configuration expérimentale

Nos experiences ont été faites sur des machines tournant sous Linux 2.6.7. Six machines étaient équipées chacune d'un processeur de 2083 MHz et de 885 Mb de RAM, six machines étaient équipées chacune de deux processeurs de 1533 MHz chacun et de 885 Mb de RAM. Sept machines étaient équipées chacune d'un processeur de 1800 MHz et de 504 Mb de RAM. Toutes les machines étaient connectées grâce à un réseau Ethernet de 100 Mbps. Toutes les machines exécutaient la version 1.4.1 de FreePastry. Tous les tests ont été exécutés 100 fois et la moyenne des resultats obtenus a été effectuée pour chaque test.

7.4.3 Influence de la périodicité des fautes

Dans ce test, nous avons fixé pour chaque nœud la probabilité d'apparition de fautes à 0.5. Nous avons fait varier le temps entre chaque injection probable de fautes pour tous les nœuds de 1 secondes à 5 secondes. Nous avons exécuté le test sur 19 machines. Dans tous les cas, le réseau FreePastry a été capable de se reconfigurer pour que les nœuds interrompus soit correctement retirés de la structure distribuée.

7.4.4 Influence de la probabilité des fautes

Pour ce test, nous avons fixé pour chaque nœud la périodicité d'apparition de fautes potentielles à 10 secondes. Nous avons fait varier la probabilité d'apparition de fautes de 0.1 à 1 avec une incrémentation de 0.1. Nous avons lancé le test sur 19 machines. Dans tous les cas avec une probabilité inférieure à 0.9(inclu), le réseau FreePastry était capable de se reconfigurer afin que les nœuds tués soit correctement retirés de la structure distribuée. Pour une probabilité de 1, tous les nœuds sont tués, il n'est donc pas surprenant que le réseau FreePastry ne puisse pas gérer ce cas.

7.4.5 Influence du nombre de nœuds

Pour ce test, nous avons fixé, pour chaque nœud, la périodicité d'apparition de fautes potentielles à 10 secondes et la probabilité de fautes à 0.5. Nous avons fait varier le nombre de nœuds (et donc de machines) de 19 à 35. Il s'avère que jusqu'à 34 nœuds, le réseau Pastry est capable de se reconfigurer. Cependant, pour 35 nœuds, dans 12% des exécutions, le réseau FreePastry n'a pas été capable de se reconfigurer et a fini par créer au moins deux réseaux séparés.

7.4.6 Vue d'ensemble

Finalement, il s'avère que FreePastry est généralement capable de gérer la défaillance de processus participants en se restructurant pour que la table de hachage distribuée soit toujours maintenue malgré la dynamicité induite. Cependant, avec notre configuration, il semble y avoir un point de rupture où le nombre de processus défaillant possible est trop grand pour que FreePastry puisse le gérer. Nous envisageons approfondir cette hypothèse en utilisant un plus grand nombre de machines.

7.4.7 Conclusion

Nous avons étendu la platforme FAIL-FCI pour permettre le support d'applications distribuées Java de la même façon que les applications natives l'étaient. Ainsi, notre nouvel outil permet d'injecter des fautes dans les applications effectives, suivant les scénarios, sans avoir à modifier le code source du programme Java ou de la machine virtuelle Java.

En tant que preuve de faisabilité, nous avons utilisé une application distribuée Java disponible (FreePastry) et utilisé FAIL pour spécifier un scénario de fautes générique qui était alors automatiquement géré par l'infrastructure FAIL-FCI. Les résultats que nous avons obtenus montrent que la périodicité et la probabilité des fautes sont peu importantes, *i.e.* le réseau FreePastry est capable de se reconstruire dans tous les cas, mais que leur nombre compte (quand le nombre de nœuds potenciellement défaillant augmente, le réseau peut être partitionné en plusieurs réseaux). D'autres études sont nécessaires, utilisant des scénarios de fautes plus complexes et d'autre types d'applications distribuées Java.

7.5 Conclusion du chapitre

Les tests d'injection de fautes dans XtremWeb, MPICH-V et FreePastry ont été présentés dans ce chapitre.

Pour XtremWeb, les tests d'injection de fautes quantitatives et qualitatives qui ont été réalisés montre que sa version C++ souffre d'un problème de gestion de ressources, au niveau du dispatcher quand des workers sont victimes de défaillances, entraînant sa propre défaillance. On peut aussi déduire de l'expérimentation que XtremWeb devrait implémenter un mécanisme pour faire attention aux défaillances des workers quand une tâche est soumise car lorsque des fautes apparaissent juste après la réception d'une tâche, les performances sont plus mauvaises que lorsqu'elles apparaissent juste après la fin d'un calcul (même si dans ce cas le worker a perdu du temps à exécuter du calcul inutilement).

Pour MPICH-V, des tests d'injection de fautes ont été effectués afin de déterminer l'impact de la fréquence des fautes, de l'echelle et de l'injection de fautes simultanées sur les performances de celui-ci. On a pu ainsi constater une grande variance dépendante du moment de l'injection de fautes comparé à la dernière vague de *checkpoint*. Nous avons pu également découvrir un bogue. Grâce à la puissance d'expressivité de FAIL et à une série de tests visant à découvrir la configuration du système qui menait à l'apparition de ce bogue, nous avons pu le localiser précisément, ce qui a permis la correction de ce bogue dans une nouvelle version de MPICH-Vcl.

Pour FreePastry, les tests effectués montrent que la périodicité et la probabilité d'apparition de fautes sont peu importantes par rapport à la maintenance du réseau logique, *i.e.* le réseau FreePastry est capable de se reconstruire dans tous les cas, mais que l'échelle compte. En effet, quand le nombre de nœuds potentiellement défaillants augmente, le réseau peut être partitionné en plusieurs réseaux.

Ces tests nous montre que FAIL-FCI est adapté à l'étude d'applications distribuées et qu'il comble les nombreux inconvénients des autres approches présentées dans le chapitre 2. En effet, dans les tests effectués sur MPICH-V, des scénarios de fautes complexes ont pu être spécifiés afin d'exhiber la configuration du système qui menait à l'apparition du bogue qui a été découvert et dans tous les tests, l'aspect aléatoire de l'injection des fautes a été utilisé. De plus, du fait de la faible intrusion de l'injecteur de fautes, des résultats relativement précis sur les performances de ces applications ont pu être obtenus. Ensuite, grâce à son principe de fonctionnement, ces tests ont pu être réalisés sans modifier le code source de ces trois applications. Et enfin, l'intégration de ces applications a été faite, malgrés leur architectures très différentes, grâce au fonctionnement générique de notre injecteur de fautes.

Chapitre 8

Stress d'Applications

8.1 Introduction

Les sections 7.2.4 et 7.2.5 ont montré comment utiliser FAIL-FCI pour obtenir une vue de la résistance aux fautes de XtremWeb en utilisant une approche unifiée pour l'analyse quantitative et qualitative. Nous allons maintenant montrer que le même outil peut être configuré pour gérer également des expériences de stress d'une application.

En fait, nous avons vu, lors d'une collaboration avec les chercheurs de l'Université de Coimbra (Portugal), que pour stresser une application on utilise un programme dédié et spécifique. En effet, dans [43] nous avons utilisé l'outil de test pour les services de grilles QUAKE pour stresser l'intergiciel OGSA-DAI. Ces tests sont présentés dans la section 8.2. Tout d'abord nous présentons OGSA-DAI, puis l'outil QUAKE et enfin nous présentons les résultats que nous avons obtenus.

Or, les mécanismes mis en œuvre pour ces tests semblaient être pilotables par FAIL-FCI. Nous avons donc effectué des tests de stress de XtremWeb en utilisant FAIL-FCI. Ces tests sont présentés dans la section 8.3. Ce que l'on entend par « expérience de stress » dans ce cas, c'est tester les performances quand le dispatcher doit gérer un nombre croissant de connexions de workers. En effet, cela permet de mesurer le surcoût de la gestion des workers par le dispatcher pour différents taux de connexions. Quand le nombre de workers disponibles augmente, la puissance de calcul augmente également, mais le dispatcher doit gérer plus de connexions et les performances globales ne sont pas nécessairement meilleures.

8.2 Utilisation d'une application dédiée

8.2.1 Présentation de OGSA-DAI

OGSA-DAI [59] est un intergiciel qui permet à des données, comme des bases de données relationnelles ou XML, d'être accessibles en tant que services. Il inclut une collection de composants pour faire des requêtes, transformer et délivrer des données de plusieurs façons différentes. Il inclut également un outil simple pour le développement des applications clientes. Pour résumer, OGSA-DAI fournit aux utilisateurs un moyen d'adapter leur ressources de données aux Grilles.

Le diagramme de la figure 8.1 représente l'architecture globale de la plate-forme OGSA-DAI. La couche *Donnée* représente les ressources de données qui peuvent être utilisées via OGSA-DAI. Les ressources de données peuvent être des bases de données relationnelles, des bases de données XML et des fichiers. Les requêtes aux services web de OGSA-DAI ont un format unique indépendant de la ressource de données utilisée par le service.

La couche *Logique* représente le coeur de OGSA-DAI qui est constitué de plusieurs composants : les ressources de services de données qui sont directement connectées aux ressources de données. Chaque ressource de services de données est reliée à une ressource de données. Les ressources de services de données sont responsables de la gestion des sessions, du traitement des documents qui décrivent les actions qu'une ressource de services de données devraient faire, de la génération des documents de réponse qui contiennent les résultats d'une requête sur une base de données, de la transmission du flux de données des ressources de services de données à partir et vers les clients.

La couche *Présentation* encapsule les fonctionnalités requises pour exposer les ressources de services de données en utilisant les interfaces de sevices web. L'interface du service Web peut être exécutée sur deux types de spécifications : *Web Services Interoperability* (WSI) [84] et *Web Services Resource Framework* (WSRF) [85]. La version WSI est exécutée sur Jakarta Tomcat [48] et Axis [7], tandis que la version WSRF est exécutée avec l'outil Globus [33]. La couche *client* permet à tous les client d'interagir, de façon transparente, avec une ressource de données via le service web correspondant.

L'interface de OGSA-DAI est un ensemble de services Web qui, dans le cas du WSI, requiert un container SOAP pour manipuler les requêtes arrivantes et les traduire pour le moteur interne de OGSA-DAI. Une description détaillée du moteur interne de OGSA-DAI peut être trouver dans [59]. L'aspect intéressant à prendre en compte est le service Web qui manipule la couche *transport* utilisant les messages SOAP. L'outil de test QUAKE

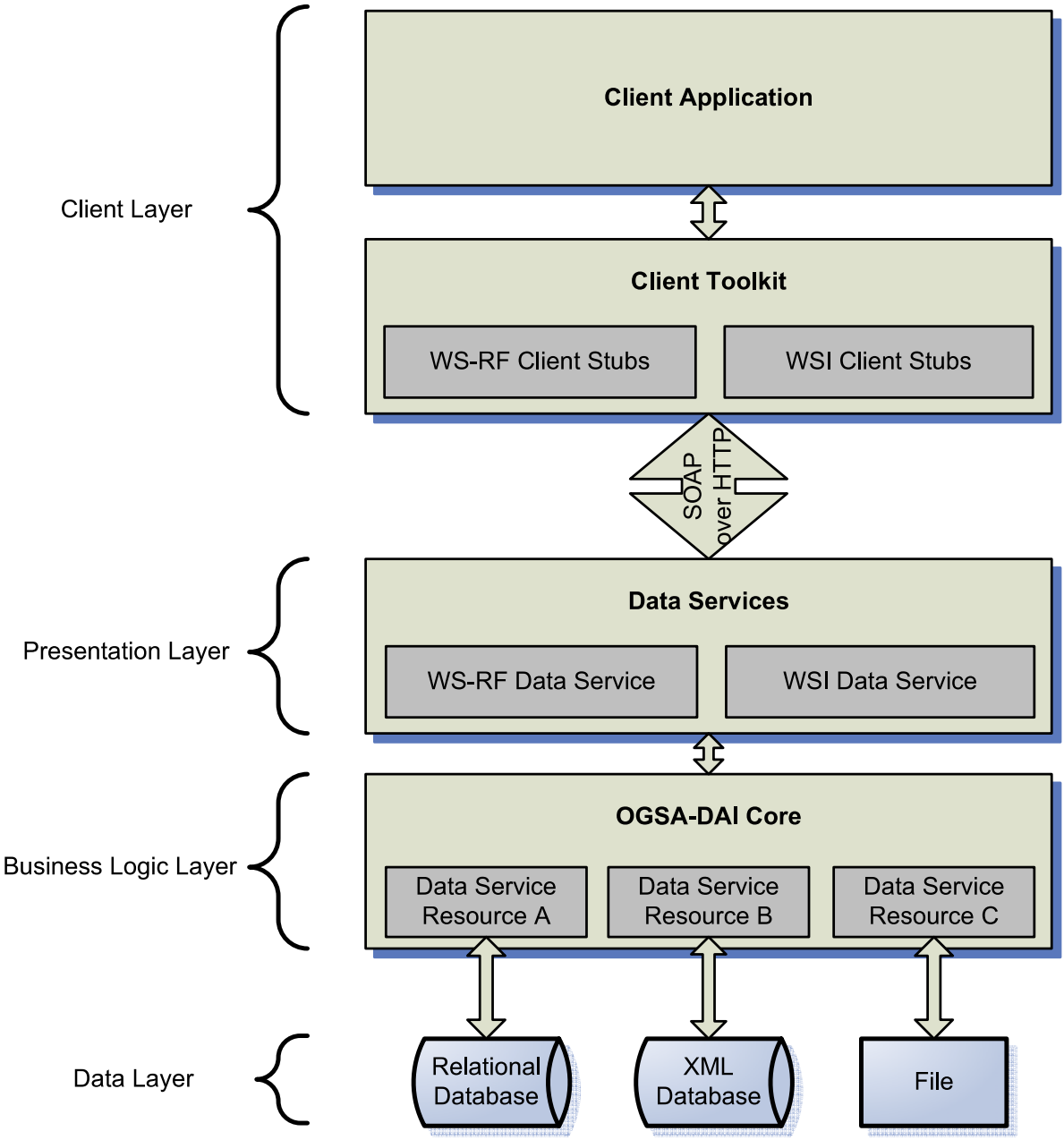

FIG. 8.1 – L'architecture de OGSA-DAI

peut être facilement appliqué à OGSA-DAI car il utilise les spécifications standards des services Web.

L'intergiciel OGSA-DAI est utilisé dans plusieurs projets importants sur les Grilles [60] tels que : AstroGrid, BIoDA, Biogrid, BioSimGrid, Bridges, caGrid, COBrA-CT, Data Mining Grid, D-Grid, eDiaMoND, ePCRN, ESSE, FirstDIG, GEDDM, GeneGrid, GEODE,

GEON, GridMiner, InteliGrid, INWA, ISPIDER, IU-RGRbench, LEAD, MCS, myGrid, N2Grid, OntoGrid, ODD-Genes, OGSA-WebDB, Provenance, SIMDAT, Secure Data Grid, SPIDR, UNIDART et VOTES. Cette liste est clairement représentative de l'importance de OGSA-DAI et de l'intérêt des tests que nous avons effectués.

8.2.2 QUAKE : un outil de test pour les services de Grilles

QUAKE [74] est un outil de test de fiabilité pour vérifier les performances et le niveau de fiabilité de services Web et de services de grilles. Il est composé d'un ensemble de composants logiciels, comme le montre la figure 8.2.

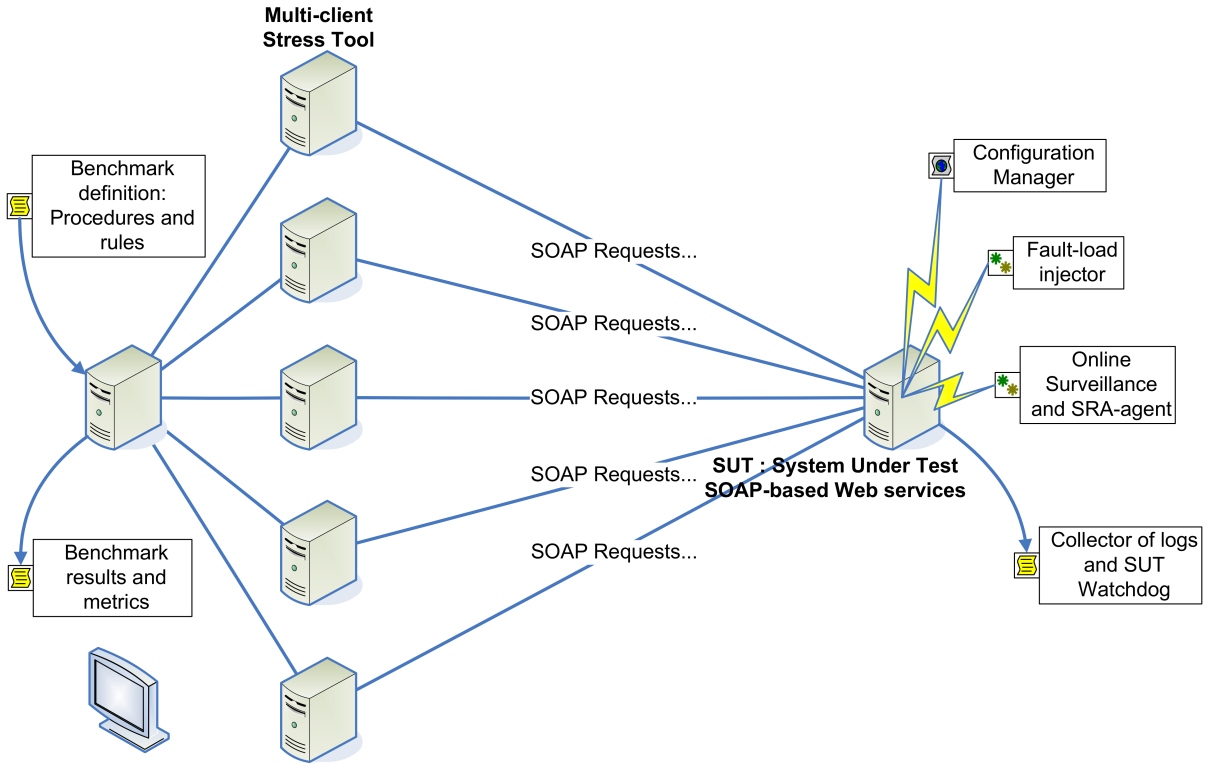

FIG. 8.2 – L'architecture de QUAKE

Ses principaux composants sont le BMS (*Benchmark Management System* ou système de gestion du jeu d'essai) et le SUT (*System-Under-Test* ou système testé). Le système testé consiste en un serveur SOAP exécutant des services Web ou des services de Grilles. L'application testée n'est pas limitée à une application basée sur SOAP : En effet, l'infrastructure du jeu d'essais peut aussi être utilisée avec d'autres exemples d'applications client/serveur utilisant d'autres intergiciels de technologies différentes.

Il y a plusieurs machines clientes qui invoquent des requêtes dans le serveur en utilisant des requêtes SOAP. Le BMS est une collection d'outils logiciels qui permet l'exécution

automatique du jeu d'essai. Il inclut un module pour sa définition, un ensemble de procédures et de règles définissant la charge qui sera produite dans le système testé, un module qui collecte tous les résultats du jeu d'essai et produit quelques resultats qui sont exprimés comme un ensemble de métriques de fiabilité. Le BMS peut activer un ensemble de clients (exécutés sur des machines distinctes) qui injecte la charge définie dans le système testé en faisant des requêtes au service Web. L'exécution des machines clientes est synchronisée et tous les résultats partiels collectés par chaque client individuel sont fusionnés en un ensemble global de résultats qui génère l'estimation finale des métriques de fiabilité. Le BMS inclut un outil de rapport qui présente les resultats finaux dans un format lisible et graphique.

Les resultats générés par chaque exécution du jeu d'essais sont exprimés ainsi : le débit par rapport au temps, le temps total d'exécution, la latence moyenne, la fonctionnalité des services, l'occurrence des défaillances, la caractérisation de ces défaillances (*crash, hang, zombie-server*), l'exactitude des résultats finaux et les scénarios de défaillance qui sont observés par les machines clientes (messages d'erreur explicites ou *timeouts*).

QUAKE inclut une liste de différentes charges qui peuvent être choisies au début de chaque expérience : *poisson*, *normal*, *burst* et *spike*. Dans cette étude, nous avons juste utilisé la distribution *burst* qui génère le maximum de charge.

Du côté du système testé, il y a quatre modules qui font également partie de l'outil de test QUAKE : un injecteur de charge, un gestionnaire de configuration, un collecteur de logs et de résultats des tests et un « watchdog » du système testé. L'injecteur de charge est un module optionel. C'est un programme externe qui est exécuté sur la même machine que le SUT et qui utilise ainsi les mêmes ressources système, consumant une ou plusieurs ressources du système d'exploitation suivantes : (a) Consommation de la mémoire en utilisant une distribution *ramp-up* ; (b) Création d'un grand nombre de threads ; (c) Utilisation massive des descripteurs de fichiers ; (d) Consommation des connexions avec la base de données. Le gestionnaire de configuration aide à la définition des paramètres de configuration de l'intergiciel du système testé. Il est absolument vrai que les paramètres de configuration peuvent avoir un impact considérable sur la robustesse du SUT. En changeant ces paramètres dans différentes exécutions du test cela nous permet de déterminer l'impact de ces paramètres dans les résultats de fiabilité. Le système testé devrait être également installé avec un module pour la collecte de données découlant de l'exécution du test. Ces données de log seront alors envoyées au serveur BMS qui les fusionnera et les comparera avec les données collectées à partir des machines clientes. Le dernier module est le *SUT-Watchdog* qui détecte quand un SUT est *crashé* ou est suspendu lors

de l'exécution du test. Quand une telle défaillance est détectée, le *watchdog* relance le SUT et les applications associées, permettant ainsi une exécution automatique des tests sans l'intervention de l'utilisateur.

Il y a quelques autres outils qui peuvent être utilisés pour tester les performances des services Web SOAP : SOAtest [76] et TestMaker [80]. Cependant, ces outils sont principalement destinés à tester les fonctionnalités des applications SOAP et à collecter quelques courbes de performance.

QUAKE a quelques différences fondamentales : il vise à étudier les attributs de fiabilité, il inclut une approche différente pour les distributions de charge, un module de charge pour affecter les ressources du système testé et il est principalement destiné à la collecte de metriques de fiabilité.

8.2.3 Résultats obtenus

Dans cette section, nous présentons les tests que nous avons faits sur OGSA-DAI en utilisant l'outil QUAKE dans une infrastructure de grille à grande échelle : Grid5000 [34]. La plupart des tests ont été exécutés sur GriDeXplorer, le site d'Orsay, qui est un composant majeur de la plate-forme Grid5000. Pour les expériences de passage à l'échelle, 3 clusters de Grid5000 ont été utilisés. Nous avons utilisé 200 machines du cluster d'Orsay, 50 machines du cluster de Sophia et 50 machines du cluster de Lille. En utilisant cette configuration, nous étions capable de monter à 300 clients durant nos expériences. L'environnement déployé était un système Debian, avec le kernel 2.6.13-5, incluant Java 1.5.0, Tomcat 5.0.28, Axis 1.2.1 et OGSA-DAI WSI 2.2.

Test de Tomcat+Axis Comme OGSA-DAI est un intergiciel basé sur Tomcat et Axis, la première étude consistait à mesurer le niveau de fiabilité d'une simple application de services Web déployée avec Tomcat/Axis exécutée dans un des sites de Grid5000. Le test était exécuté avec 40 clients et 100 000 requêtes par client. Nous avons utilisé une application synthétique qui exécute un calcul très simple. La version du container WSI était : Tomcat 5.0.28 et Axis 1.2.1. Les résultats obtenus sont présentés dans la figure 8.3 et la figure 8.4.

Dans la figure 8.3, nous pouvons voir que le débit est décroissant dans le temps jusqu'à un certain point où le serveur crash : à ce point seulement 665 078 requêtes ont été exécutées, au lieu des 4,000,000 qui étaient attendues.

Dans la figure 8.4, nous pouvons voir la latence augmenter avec le temps jusqu'à un certain point où le serveur crash (approximativement 70.5 minutes après le lancement du

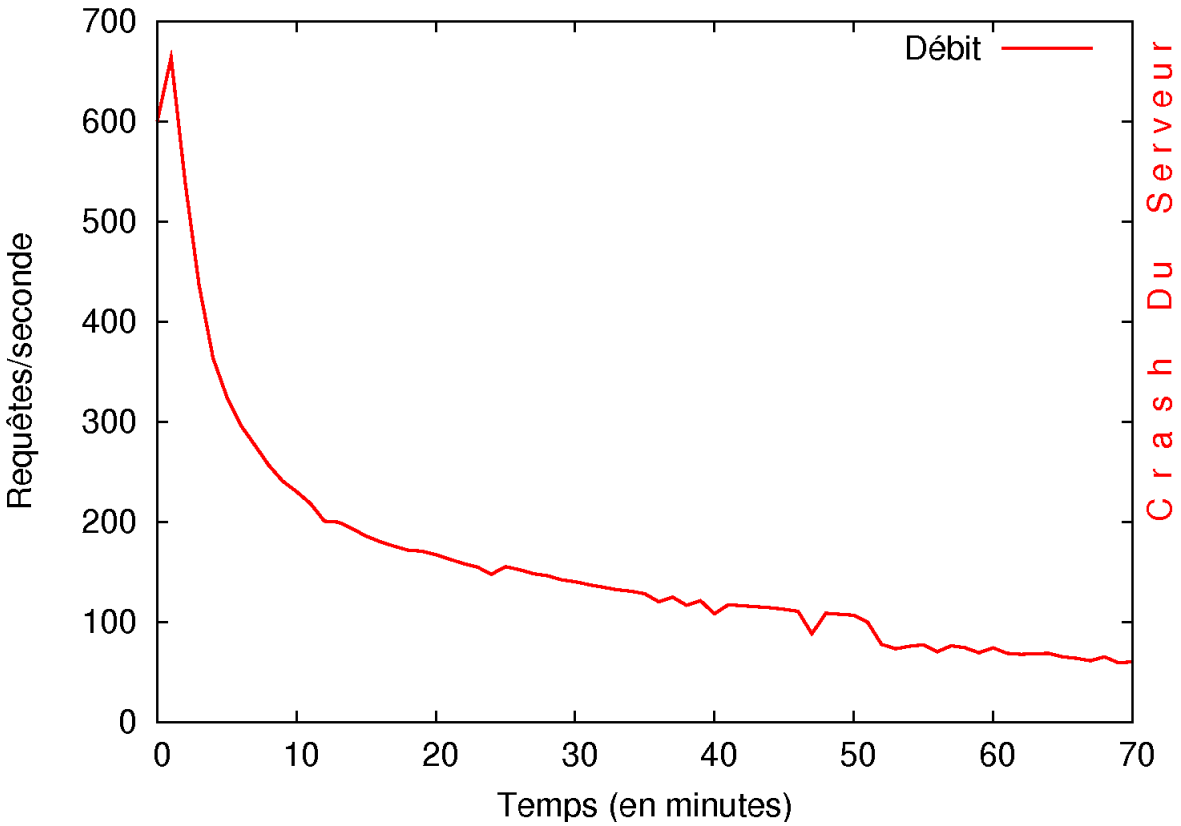

Fig. 8.3 – Débit avec Tomcat/Axis

test).

Cette première étude a été faite avec une application synthétique. Nous avons ensuite décidé d'implanter un service Web sur Axis qui fonctionnerait comme une interface pour accéder à une base de données : il reçoit une requête SQL et retourne les lignes demandées. Nous avons alors exécuté le même type de tests avec également 40 clients et les resultats obtenus sont présentés dans la figure 8.5 et dans la figure 8.6. Nous pouvons voir que les résultats sont très similaires à ceux obtenus dans la première expérience. Le débit décroît dans le temps jusqu'à ce que le serveur crashe après 80 minutes d'exécution.

Ce comportement était le même que celui observé dans [74] où une étude de fiabilité de SOAP a été faite et a montré que Axis 1.3 souffrait d'importantes fuites de mémoire. Des mécanismes de correction sont alors nécessaires pour éviter les défaillances des applications qui utilisent cette implantation de SOAP.

C'est cette étude particulière qui nous a incitée à évaluer le niveau de fiabilité de OGSA-DAI : si Axis 1.2 et 1.3 souffre d'importantes fuites de mémoire et que OGSA-DAI utilise Axis 1.2.1, qu'est-ce qu'il en est des applications utilisant les services de données de OGSA-DAI ?

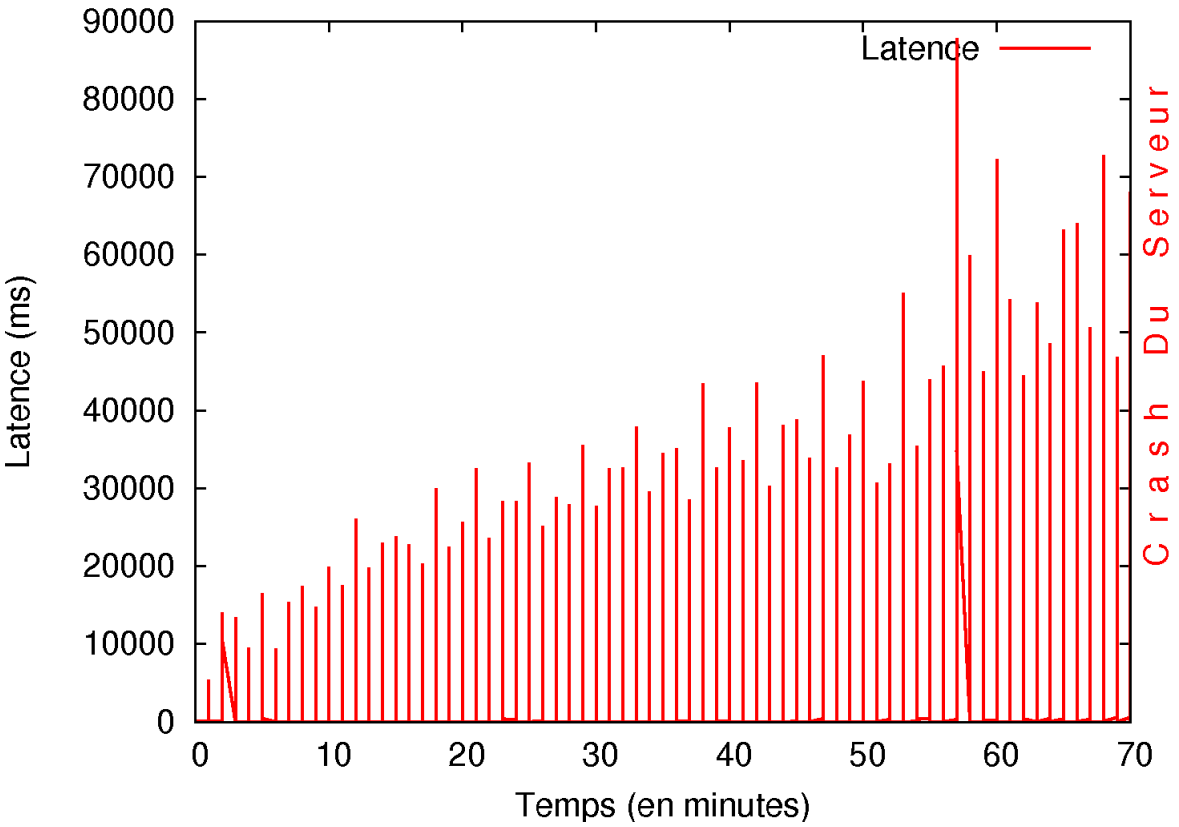

FIG. 8.4 – Latence avec Tomcat/Axis

Test de OGSA-DAI Puisque OGSA-DAI utilise la plate-forme Axis1.2.1 pour le déploiement de ses propres services Web, il est donc légitime de supposer que OGSA-DAI devrait souffrir sévèrement des effets des fuites de mémoire de Axis. Pour le montrer, nous avons effectué plusieurs tests de la plate-forme OGSA-DAI en utilisant GriD eXplorer (GDX).

La première expérience a été faite avec 25 nœuds, chacun exécutant 100,000 requêtes. La figure 8.7 présente le débit observé et la figure 8.8 la latence observée. On a pu ainsi constater que les performances de OGSA-DAI sont restées stables durant toutes les expériences qui ont été faites avec la distribution *burst* maximale. Le débit est relativement stable avec une moyenne de 71.43 requêtes/seconde ainsi que la latence qui a une valeur moyenne de 349.1 ms.

Nous avons répété ce test mais en augmentant le nombre de clients pour voir l'impact de l'échelle sur l'intergiciel OGSA-DAI. Différentes expériences ont été faites avec 25, 100, 200 et 300 clients. Les résultats sont présentés dans la figure 8.9.

Nous pouvons voir que le débit est le même pour toutes les expériences et que la latence augmente de façon linéaire. Le serveur peut parfaitement gérer 300 requêtes simultanées

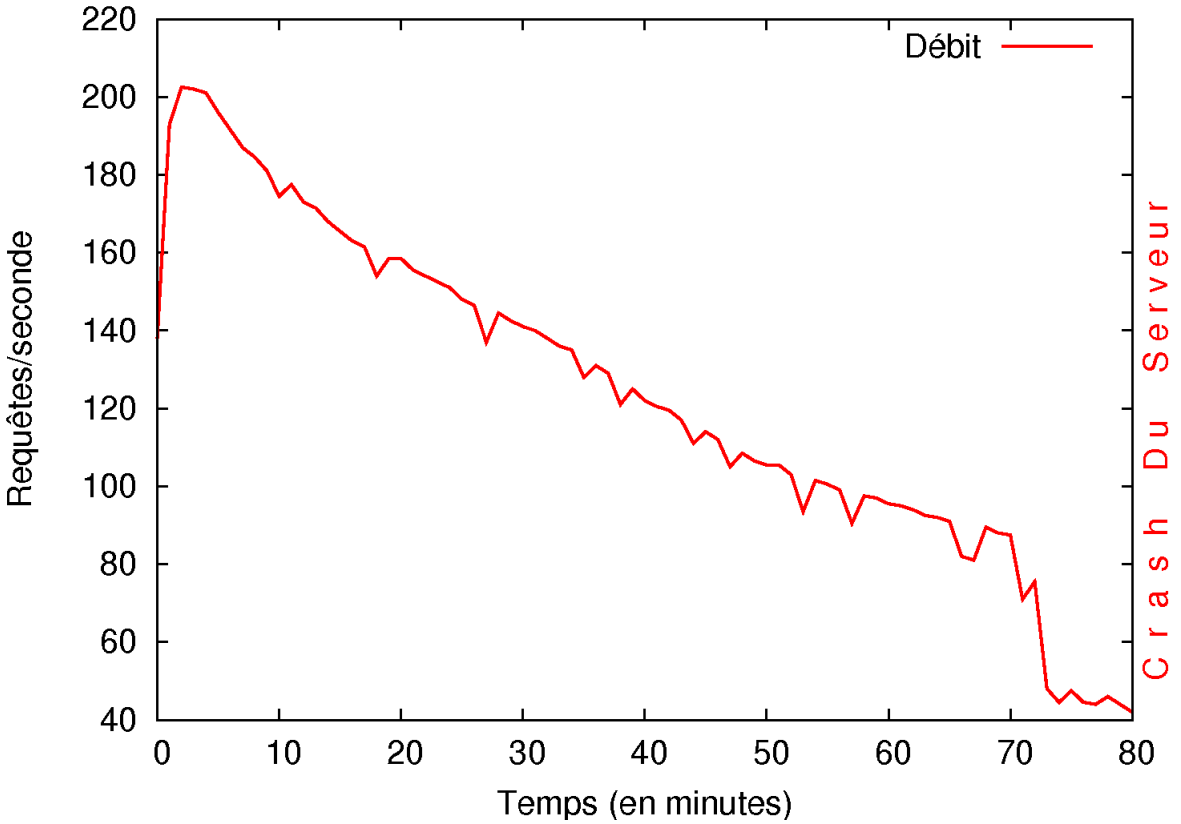

Fig. 8.5 – Débit avec Tomcat/Axis (application synthétique de base de données)

de clients accédant aux services de données en utilisant une distribution *burst*.

Il est clair que OGSA-DAI possède un mécanisme de contrôle de la congestion du flux qui fixe le débit à une certaine valeur indépendemment du nombre de clients simultanés. Bien que cela puisse expliquer l'impact de l'échelle sur le serveur, cela n'explique pas pourquoi les fuites de mémoire présentes dans Axis ne se manifestent pas dans un environnement utilisant OGSA-DAI.

Nous avons alors commencé à nous intéresser à quelques paramètres de configuration de OGSA-DAI, et en particulier aux paramètres principaux qui sont, a priori, susceptibles d'affecter le mécanisme de contrôle de la congestion du flux :

- *maximum simultaneous request* : définie combien de requêtes peuvent être traitées simultanément ;
- *request queue length* : correspond à la longueur de la queue où les requêtes sont stockées avant leur traitement.

Pour tous les tests précédents, le nombre maximal de requêtes simultanées était le même que le nombre de clients : par exemple, pour 100 clients, le *maximum simultaneous request* était configuré à 100. La longueur de la queue était toujours configurée à 20.

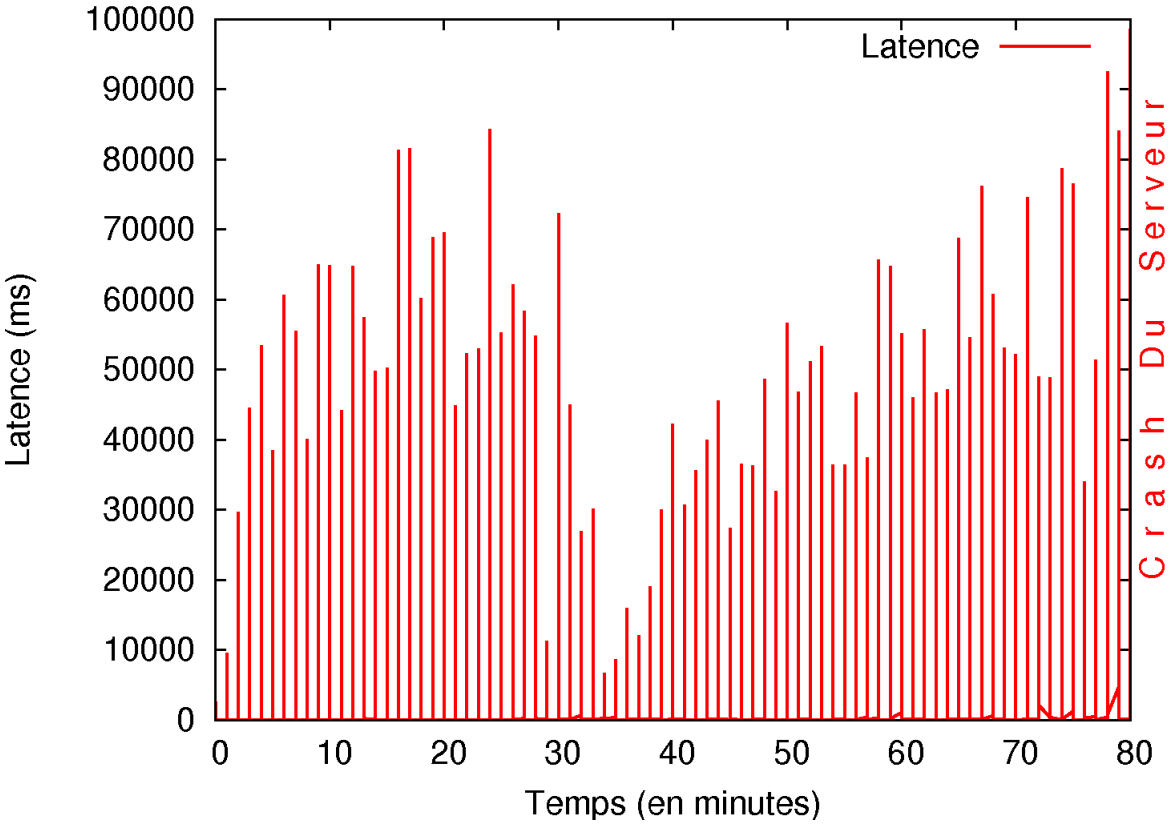

FIG. 8.6 – Latence avec Tomcat/Axis (application synthétique de base de données)

Nous avons alors encore exécuté le test de passage à l'échelle, mais avec un *maximum simultaneous request* deux fois plus petit et une plus grande queue pour voir l'impact de ces paramètres sur la performance. Les résultats que nous avons obtenus sont pratiquement les mêmes que ceux présentés dans la figure 8.9. Donc, apparemment, ces paramètres n'ont aucune influence sur les performances, au moins jusqu'à cette échelle.

Après ce premier ensemble de résultats, nous n'avions toujours aucune réponse à notre question fondamentale : comment OGSA-DAI peut être aussi stable alors qu'il utilise Axis1.2.1 qui souffre de fuites de mémoire ?

Nous avons alors fait d'autres expériences et inspecté le code de l'implantation d'OGSA-DAI, en essayant de comprendre son fonctionnement interne. Une chose qui a attiré notre attention est la méthode qui est appelée avant l'exécution d'une opération consommant de la mémoire. Cette méthode calcule la mémoire nécessaire pour l'exécution de l'opération, et vérifie alors si la mémoire totale consommée est au dessus d'un certain seuil prédéfini. Si elle ne l'est pas, alors l'opération peut être exécutée. Sinon, l'intergiciel fait plusieurs appels explicites au ramasse miettes de la machine virtuelle Java (JVM) pour libérer de la mémoire.

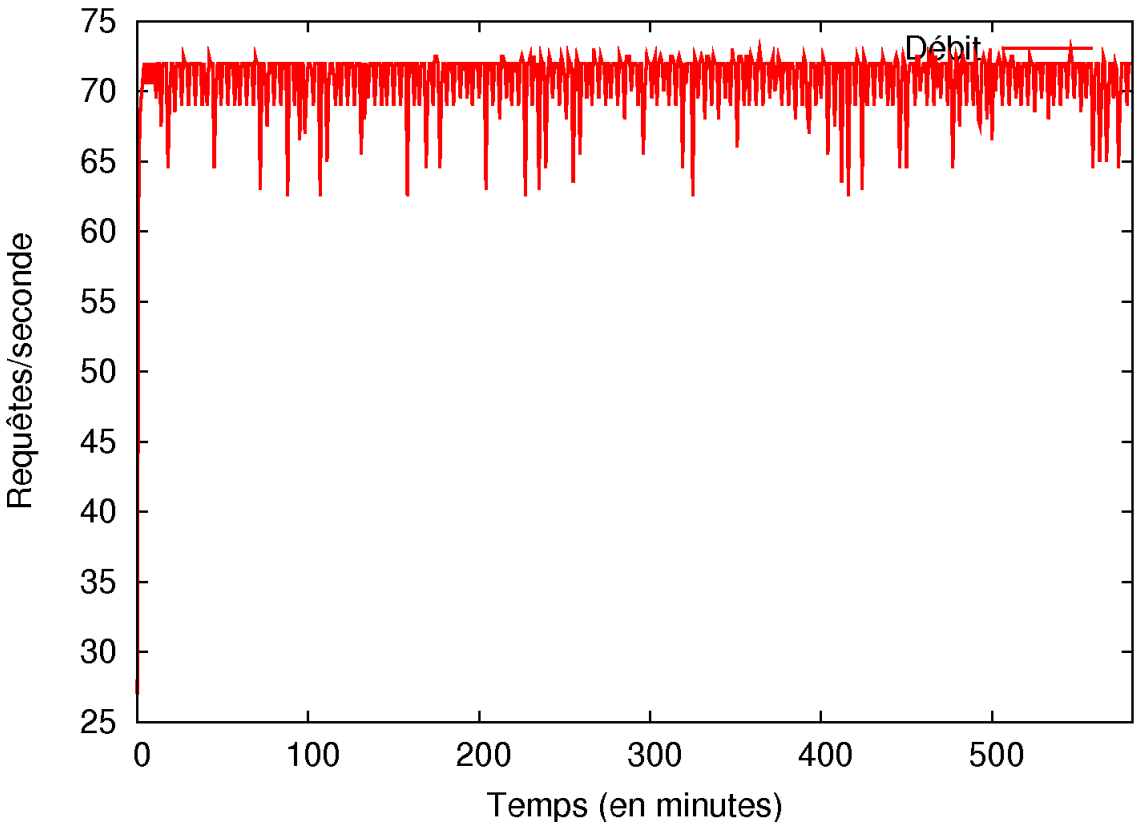

Fig. 8.7 – Débit avec OGSA-DAI(WSI)

Le seuil par défaut pour la consommation de mémoire est de 85% de la pile de la JVM, ainsi tous les tests que nous avions précédemment exécutés utilisaient cette valeur. Nous avons alors fait d'autres tests en changeant la valeur de seuil : 50%, 85%, 99% et en désactivant le mécanisme. L'idée étant de voir si c'est ce mécanisme qui est le principal responsable de la stabilité du serveur OGSA-DAI. Les resultats obtenus sont présentés dans la figure 8.10.

Ce mécanisme oblige l'utilisation plus fréquente du mécanisme de ramasse miettes, mais il n'est pas le mécanisme qui fournit la robustesse de OGSA-DAI. En effet, quand on désactive ce mécanisme, on peut voir une légère instabilité dans la consommation de la mémoire mais le serveur n'est jamais défaillant.

Enfin, il y avait un dernier point qui a capté notre attention : le fait que le service Web déployé avec OGSA-DAI soit étendu à l'application. Cela veut dire que l'application service Web est instanciée seulement une foi et ensuite partagée avec toutes les requêtes qu'elle reçoit et avec chaque client différent. Ce qui n'est pas la façon habituelle de déployer des services Web à moins que le service Web ne fournissent des données globales qui devraient être partagées par tous les clients. Il est assez courant qu'une application de

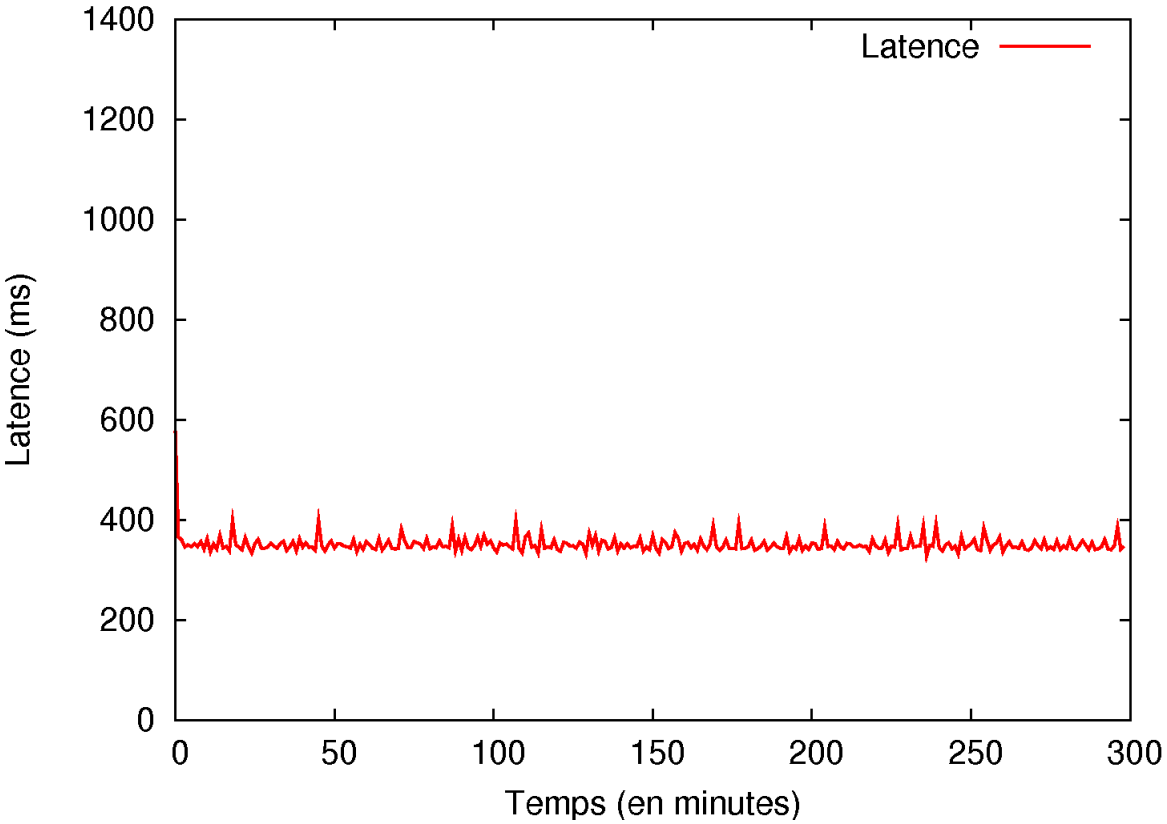

FIG. 8.8 – Latence avec OGSA-DAI(WSI)

services Web ait besoin de stocker des informations sur une session particulière afin qu'elle puisse corréler les requêtes futurs avec les précédentes faites par un même utilisateur. Ce genre d'application utilise souvent une étendue aux sessions afin de pouvoir gérer les informations sur les clients dans un modèle fondé sur les sessions.

Comme nous l'avons vu dans la section 8.2.1, OGSA-DAI utilise les services Web comme une couche d'interface pour les ressources de données. Il n'a vraiment pas besoin de corréler les différentes requêtes, il a juste besoin de créer une interface qui sera la même pour toutes les requêtes indépendemment du client d'où elles viennent. Cela justifie clairement l'utilisation d'une étendue à l'application dans le déploiement des services Web. Mais cela soulève la question : serait-ce la raison de la stabilité de OGSA-DAI ?

Pour comprendre l'impact de l'étendue des services Web dans la robustesse de l'application, nous avons fait deux tests différents :

1. un test avec un service Web déployé sur Axis utilisant une étendue à l'application ;
2. un autre test utilisant OGSA-DAI mais dont l'étendue a été changée aux sessions.

Dans les deux tests, nous avons utilisé l'outil QUAKE avec 10 clients simultanés invoquant chacun 10 million de requêtes sur le serveur.

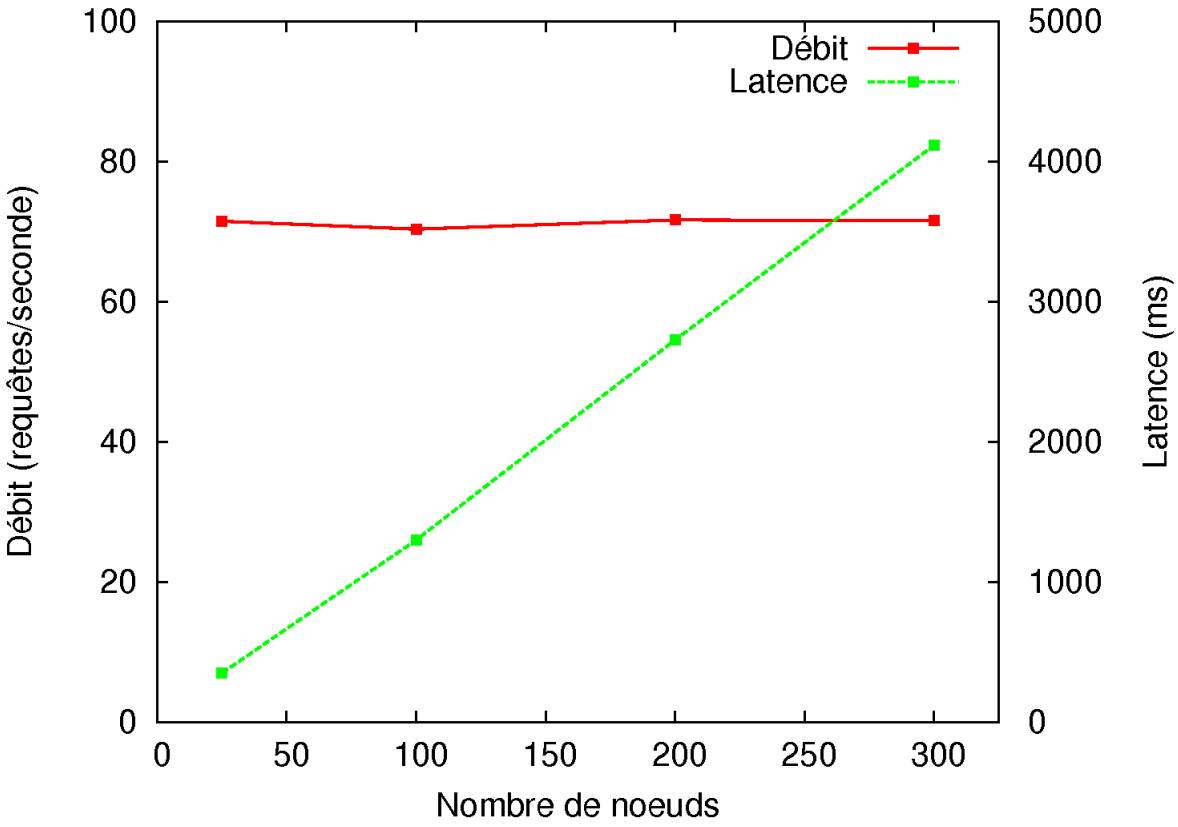

FIG. 8.9 – Impact de l'échelle sur OGSA-DAI(WSI)

La figure 8.11 et la figure 8.12 montrent respectivement le débit et la latence pour l'application synthétique exécutée sur Axis1.2.1 avec une étendue du service web à l'application.

La figure 8.13 et la figure 8.14 montrent respectivement le débit et la latence pour OGSA-DAI avec une étendue aux sessions.

Dans la figure 8.11 et la figure 8.12, nous pouvons voir que Axis1.2.1 ne souffre pas de fuites de mémoire si les services Web sont déployés avec une étendue à l'application. Contrairement, nous pouvont voir dans la figure 8.13 et dans la figure 8.14 que OGSA-DAI est très instable si l'étendue du service Web est configurée sur la session : l'application avec OGSA-DAI crash après 42.5 minutes d'exécution et est seulement capable de traiter 2020 requêtes. Le débit est très instable et n'atteint que de très faibles valeurs. Ces résultats sont assez intéressants et expliquent finalement pourquoi la configuration par défaut de OGSA-DAI était aussi stable.

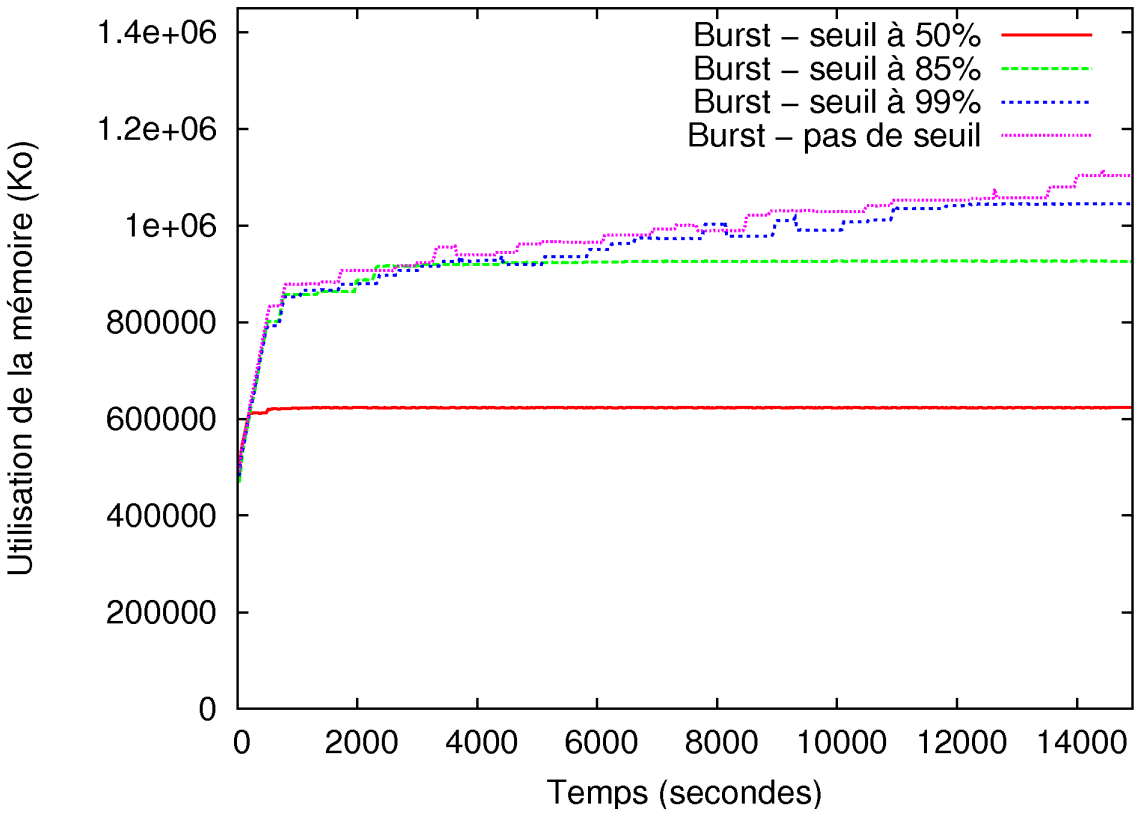

FIG. 8.10 – Impact de la valeur de seuil sur l'utilisation de la mémoire

Conclusions. Pour résumer les résultats obtenus, on peut comparer les résultats de la figure 8.13 et 8.14 avec ceux de la figures 8.7, 8.8 et 8.9. Dans ces trois figures, nous pouvons voir que OGSA-DAI est très stable et fournit un débit soutenu de 70 requêtes/seconde, même quand on augmente le nombre de client jusqu'a 300. Dans la figure 8.13, nous pouvons voir que le débit est très instable et très faible en comparaison (en moyenne de 0.78 requêtes/seconde). Nous pouvons également faire une remarque similaire sur la latence des requêtes : dans la figure 8.9, les résultats impliquant 300 clients ont montré une latence moyenne de 4000 millisecondes alors que dans la figure 8.14, la latence moyenne est de 11 842 millisecondes (presque 12 secondes) avec seulement 10 clients simultanés.

Tous ces résultats sont dus au fait que la configuration par défaut de OGSA-DAI définit une étendue du service Web à l'application. Dans ce cas, OGSA-DAI ne déclenche pas les fuites de mémoire de Axis1.2.1. Si l'étendue est configurée à la session, OGSA-DAI déclenchera les sévères fuites de mémoire de Axis1.2.1 et la fiabilité résultante sera à prendre largement en considération.

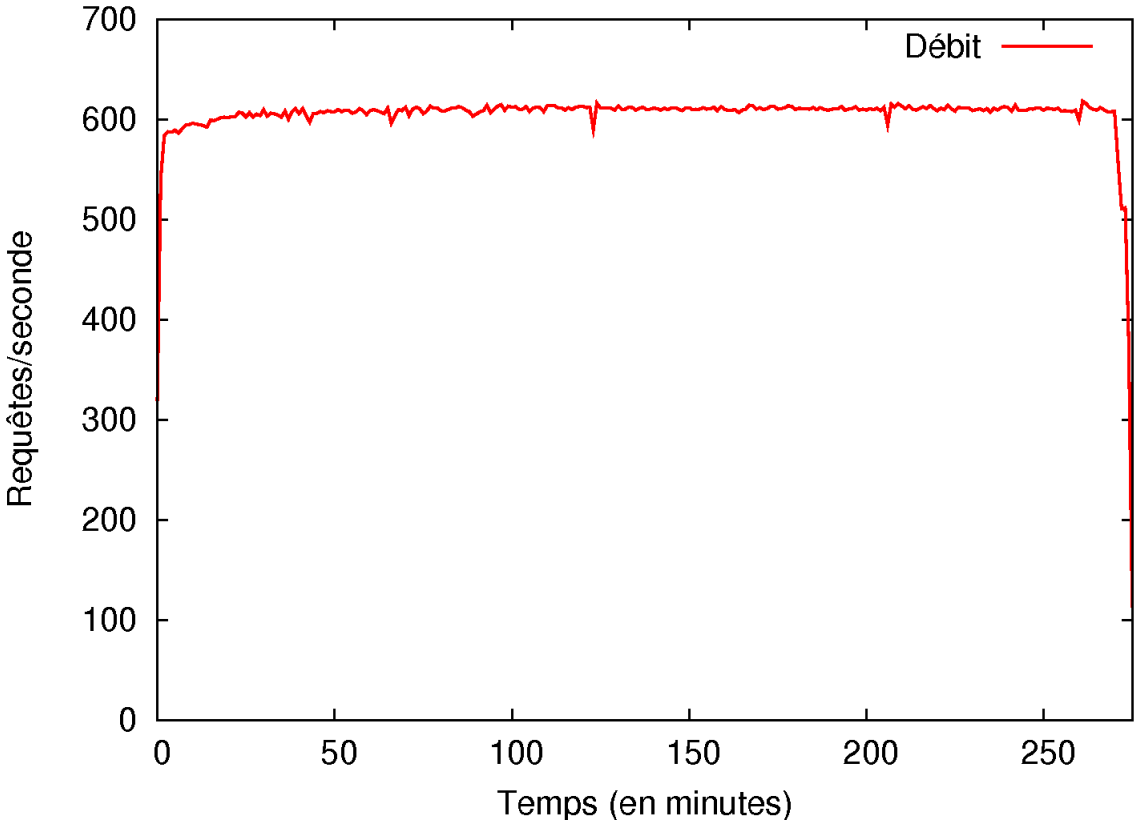

FIG. 8.11 – Débit avec Tomcat/Axis(services Web étendus à l'application)

La bonne nouvelle de ces résultats est le fait que OGSA-DAI est très stable et se comporte considérablement bien dans des scénarios de passage à l'échelle et de test de charge de travail comme le montre les tests exécutés avec 300 nœuds de Grid5000. C'est une bonne nouvelle pour tous les projets sur les grilles qui utilisent OGSA-DAI et c'est une reconnaissance de l'excellent travail qui a été fait par l'équipe de OGSA-DAI.

D'un autre côté, nous avons prouvé que la stabilité de OGSA-DAI est due au fait qu'il utilise une étendue des services Web à l'application, qui heureusement ne déclenche pas les fuites mémoires de Axis1.2.1. Si pour une quelconque raison, un programmeur a besoin d'implanter des sessions pour les différents clients et désire changer l'étendue du service Web aux sessions, alors, il sera confronté à un problème critique de fiabilité de l'intergiciel OGSA-DAI. C'est un fait intéressant qui devrait être pris en compte par l'équipe qui développe l'intergiciel OGSA-DAI.

Nous avons prouvé qu'avoir un outil de test pour les services Web et les applications pour les grilles est crucial pour la communauté. En utilisant l'outil QUAKE, nous avons été capable de détecter et de comprendre les raisons de la stabilité de OGSA-DAI, mais aussi des potentielles fuites de mémoire de Axis qui pourrait le faire passer pour un

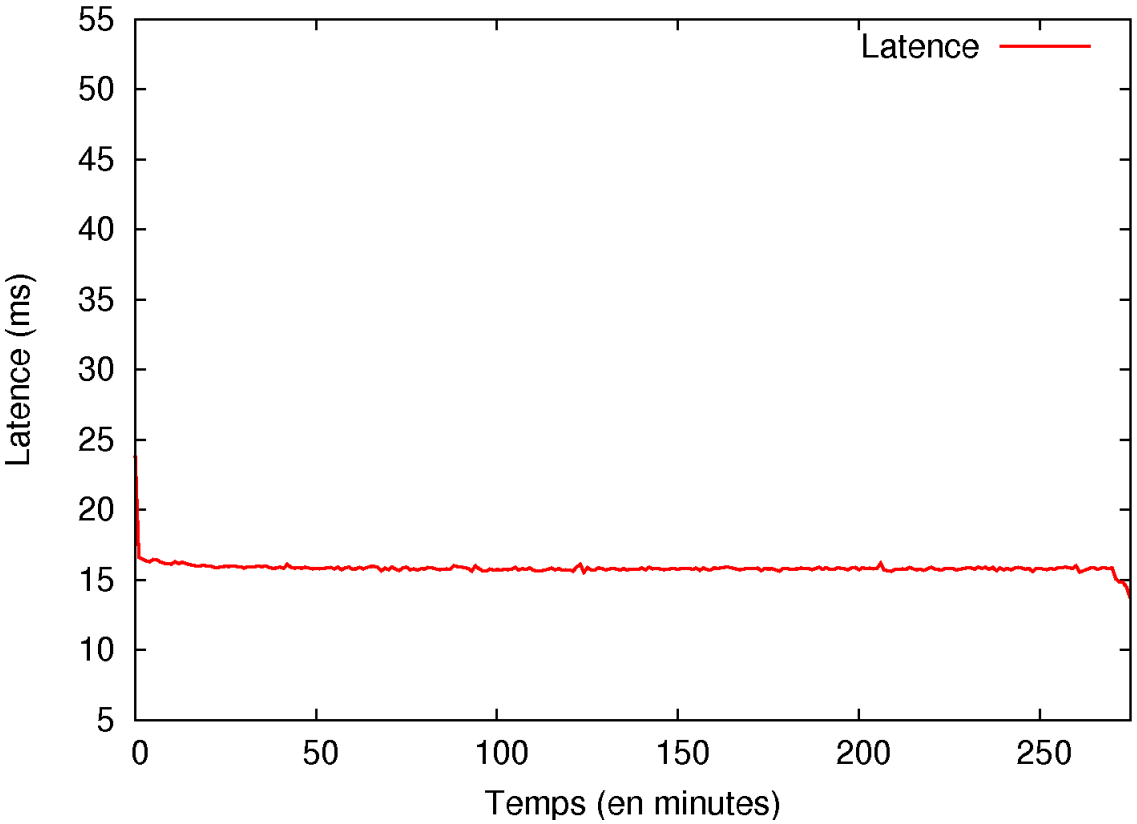

FIG. 8.12 – Latence avec Tomcat/Axis(services Web étendus à l'application)

intergiciel très instable si elles sont déclenchées.

Une autre bonne nouvelle pour la communauté est le fait que Axis 2 a déjà été implanté et cette nouvelle version corrige la plupart des problèmes des versions précédentes. Cependant, il y a encore de nombreuses applications (comme OGSA-DAI) qui utilise les version 1.2 ou 1.3 de Axis et qui pourraient alors potentiellement souffrir de sérieuses fuites mémoires si elles utilisent des services Web étendus aux sessions.

8.3 Utilisation de FAIL-FCI

Les tests de stress en utilisant FAIL-FCI [42] ont été effectués sur XtremWeb. Pour cela, nous avons utilisé un scénario de fautes légèrement différent de ceux utilisés lors des tests d'injection de fautes dans XtremWeb. L'ensemble de tâches est le même que lors de ces tests, et le client et dispatcher XtremWeb sont toujours sur la même machine. Le *test begin time* est le temps auquel le client XtremWeb et le dispatcher sont lancés (le dispatcher est alors en attente de connexions de workers pour l'exécution des tâches). Ensuite, un worker XtremWeb particulier est lancé avec la probabilité y toutes les x

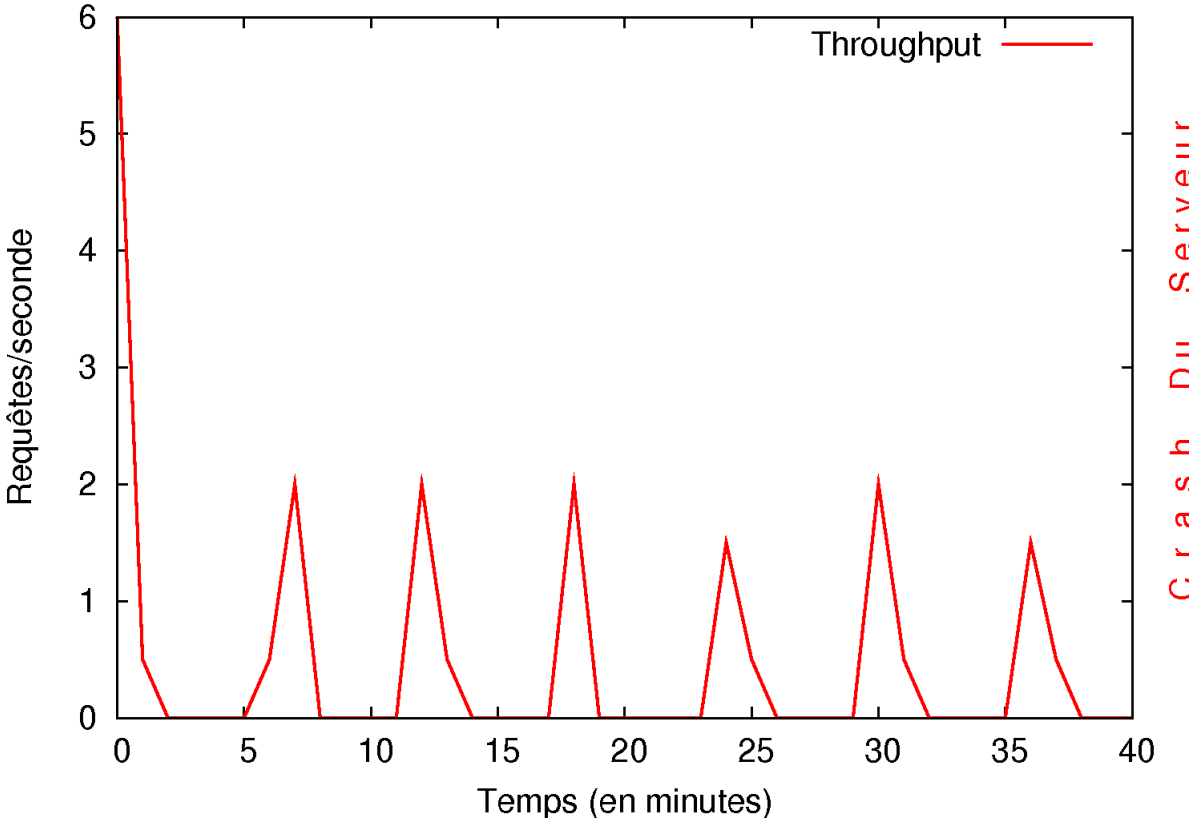

FIG. 8.13 – Débit avec OGSA-DAI(service Web étendu à la session)

secondes. Quand le client s'arrête (après avoir reçu l'acquittement du dispatcher), le temps correspondant est enregistré en tant que *test end time*.

8.3.1 Le scénario FAIL

Un scénario possible correspondant, utilisant le language FAIL, est décrit dans la figure 8.15 (avec $x = 1$ et $y = 10\%$) :

Nous avons exécuté ces tests en faisant varier x de 1 à 9 secondes (avec une incrémentation de 2 secondes), et en faisant varier y de 10% à 100% dans la configuaration `Cluster`, en faisant varier x de 1 à 2 secondes (avec une incrémentation de 2 secondes), et en faisant varier y de 10% à 100% dans la configuration GrideXplorer.

8.3.2 Résultats obtenus

Les résultats obtenus concernant le temps d'exécution global sont résumés dans la figure 8.16 et la figure 8.17. Nous n'avons pas collecté d'informations sur les défaillances du dispatcher, car nous n'en avons observé aucune.

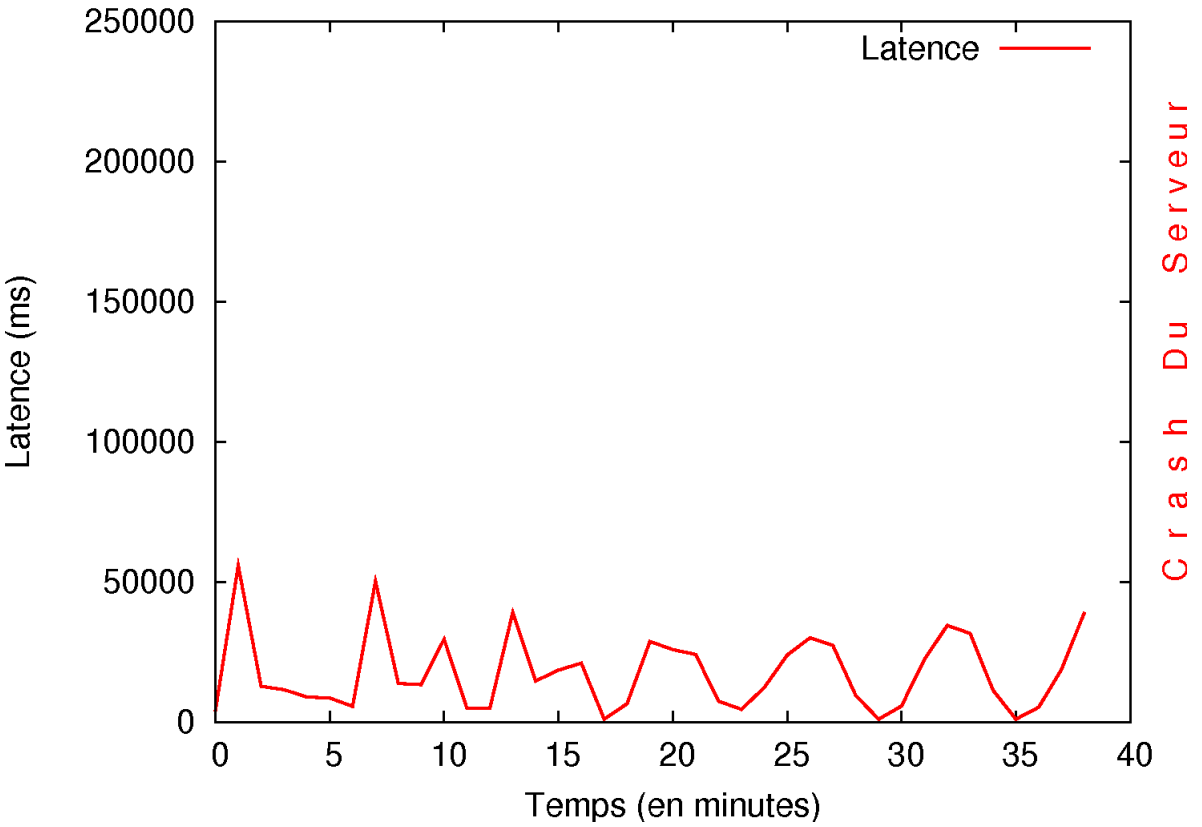

FIG. 8.14 – Latence avec OGSA-DAI(service Web étendu à la session)

On s'attendait à ce que la courbe ait la forme d'un « U » au moins pour le test avec $x = 1$ (avec l'ajout d'un worker toutes les secondes) : si quelques workers arrivent au même moment, les performances sont mauvaises ; si plusieurs workers arrivent au même instant, le dispatcher peut les gérer et les performances sont meilleures ; si trop de workers arrivent au même instant, le dispatcher pourrait être surchargé et les performances globales pourraient être plus mauvaises qu'avec moins de workers. Dans la configuration `Cluster`, toutes les courbes sont décroissantes, ce qui signifie que plus il y a de workers disponibles, plus la complétion du calcul est rapide. Cela signifie aussi que la version C++ de XtremWeb peut gérer efficacement l'arrivée simultanée de 30 nouveaux workers (ce qui est le cas quand $y = 100\%$). Au contraire, la configuration `GrideXplorer` montre que plus il y a de workers arrivant simultanément dans le système, plus le temps nécessaire pour compléter le calcul est long. Ainsi, avec 160 workers, dont la majorité arrive simultanément, le dispatcher XtremWeb est clairement stressé mais la dégradation observée évolue faiblement.

```
spyfunc main ;

Daemon ADV1 {
   node 1 : before(main) -> continue, !go(G1), goto 2 ;
   node 2 : }

Daemon ADV2 {
   node 1 : before(main) -> stop, goto 2 ;
   node 2 : ?go -> stop, goto 3 ;
   node 3 : always int x = FAIL_RANDOM(1,100) ;
            always time_g timer = 1 ;
            timer && x <= 10 -> continue, goto 4 ;
            timer && x > 10 -> stop, goto 3 ;
   node 4 : }
```

FIG. 8.15 – Scénario de fautes pour le stress d'application.

8.4 Conclusion du chapitre

Dans ce chapitre, nous avons présenté les résultats que nous avons obtenus sur le stress de OGSA-DAI en utilisant l'outil de test pour les services de grilles QUAKE et les résultats de stress de XtremWeb en utilisant FAIL-FCI comme outil de stress.

Les résultats obtenus sur OGSA-DAI montrent que celui-ci est très stable grâce à sa configuration par défaut qui définit une étendue du service Web à l'application. Dans ce cas, OGSA-DAI ne déclenche pas les fuites de mémoire de Axis1.2.1. Si l'étendue est configurée à la session, OGSA-DAI déclenchera les sévères fuites de mémoire de Axis1.2.1 et la fiabilité résultante sera a prendre largement en considération.

Ainsi, nous avons prouvé qu'avoir un outil de test pour les services Web et les application pour le grilles est crucial pour la communauté. En utilisant l'outil QUAKE nous avons été capable de détecter et de comprendre les raisons de la stabilité de OGSA-DAI mais aussi des potentielles fuites de mémoire de Axis qui pourraient le faire passer pour un intergiciel très instable si elles sont déclenchées. Une bonne nouvelle pour la communauté est le fait que Axis 2 a déjà été implanté et cette nouvelle version corrige la plupart des problèmes des versions précédentes. Cependant, il y a encore de nombreuses applications (comme OGSA-DAI) qui utilisent les version 1.2 ou 1.3 de Axis et qui pourraient alors

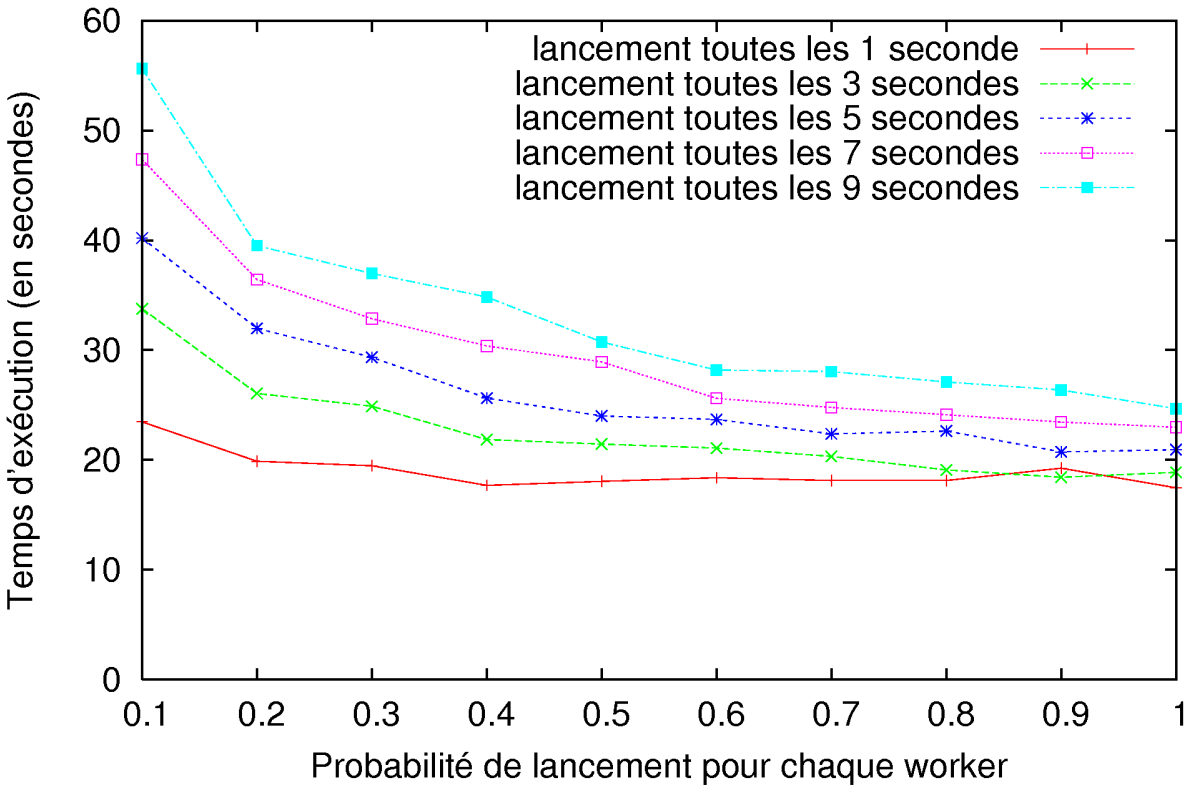

FIG. 8.16 – Test de stress : cluster avec 30 workers

potentiellement souffrir de sérieuses fuites mémoires si elles utilisent des services Web étendus aux sessions.

Or, puisque les mécanismes mis en œuvre pour les tests effectués sur OGSA-DAI semblent être pilotables par FAIL-FCI pour les application distribuées en générale, Il s'avère donc intéressant d'utiliser FAIL-FCI pour le stress d'applications distribuées et de disposer ainsi d'un unique outil permettant l'injection de fautes et le stress d'applications distribuées. Nous avons donc effectué des tests de stress de XtremWeb en utilisant FAIL-FCI.

Ce que l'on entend par « expérience de stress » dans le cas de XtremWeb, c'est tester les performances quand le dispatcher doit gérer un nombre croissant de connexions de workers. En effet, cela permet de mesurer le surcoût de la gestion des workers par le dispatcher pour différents taux de connexions. Quand le nombre de worker disponible augmente, la puissance de calcul augmente également, mais le dispatcher doit gérer plus de connexions et les performances globales ne sont pas nécessairement meilleures.

On constate ainsi que la version C++ de XtremWeb peut gérer efficacement l'arrivée

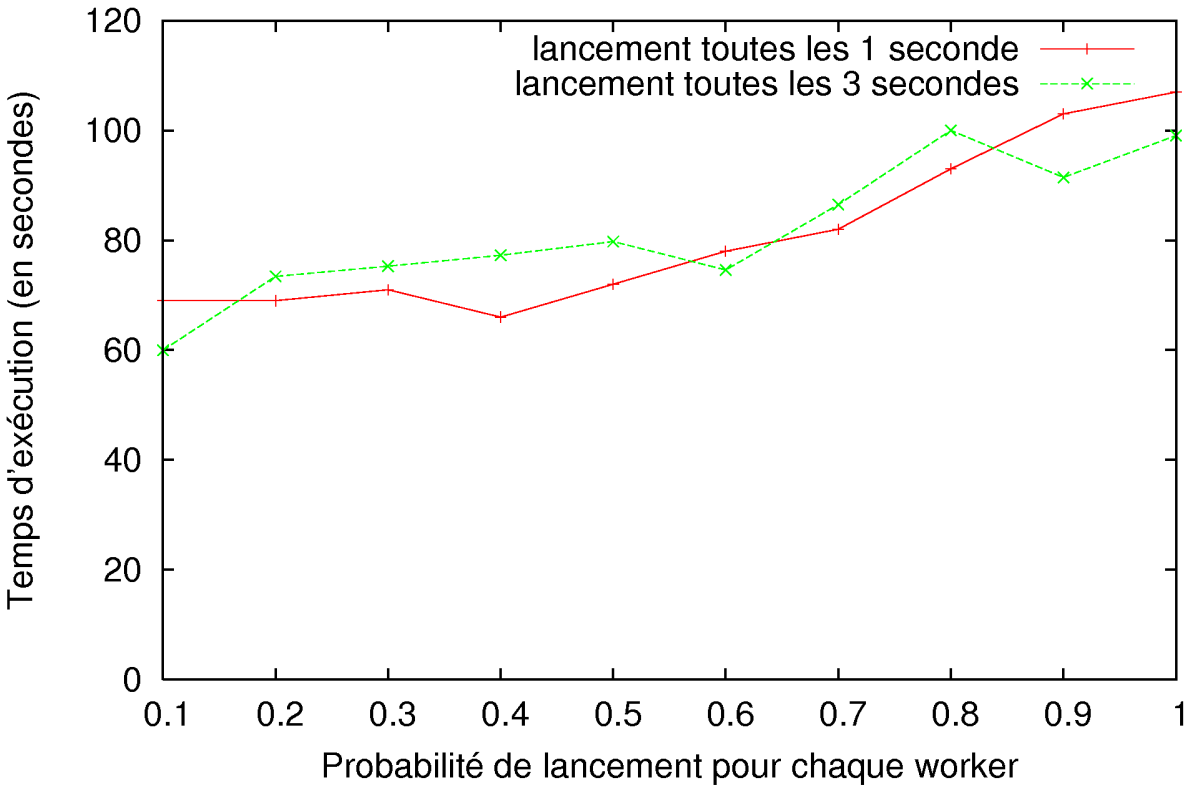

FIG. 8.17 – Test de stress : GriDeXplorer avec 160 workers

simultanée de 30 nouveaux workers. Au contraire, avec 160 workers, dont la majorité arrive simultanément, le dispatcher XtremWeb est clairement stressé, *i.e.* plus il y a de workers arrivant simultanément dans le système et plus le temps nécessaire pour compléter le calcul est long, mais la dégradation observée évolue faiblement.

Chapitre 9

Simulation d'Utilisateurs dans un Réseau

9.1 Introduction

De nos jours, les systèmes distribués à grande échelle sont de plus en plus utilisés en raison de leur efficacité. Les Desktop grids, introduits précédemment, utilisent les cycles oisifs des ordinateurs de bureau pour effectuer du calcul à grande échelle et du stockage de données pour un coût relativement faible. Mais à cause de l'échelle de ce type de systèmes et de la nature des nœuds participants, les différents hôtes sont très volatiles [61]. Et même si ces types de systèmes sont très utilisés, la volatilité des hôtes dans les grilles d'ordinateurs de bureau commence juste à être conprise [51].

La caractérisation de la volatilité des nœuds est essentielle pour créer des modèles et des simulations précis de ces plates-formes. Quelques travaux ont été faits en rassemblant des traces de disponibilité à partir de grilles d'ordinateurs de bureau réels et en les utilisant pour caractériser la volatilité de chaque système [50]. Ces traces sont utiles pour la simulation guidée par traces et la construction de modèles prédicatifs, génératifs ou explicatifs. Ils ont également permis d'obtenir des statistiques sur les hôtes qui reflètent leur volatilité.

Ensuite arrive le problème de l'extraction d'un modèle à partir de ces traces. Plus récemment, les systèmes pair à pair ont utilisé des distributions statistiques comme fondement de leurs hypothèses sur la disponibilité des nœuds et quelques études sur l'adéquation de plusieurs distributions statistiques potentielles ont été faites [58, 82, 83]. Elles ont permis le développement d'une méthode automatique pour la modélisation de la disponibilité de ressources distribuées au sein d'une entreprise et dans le monde. Les modèles de

disponibilité des machines sont basés sur la correspondance d'une distribution statistique aux données observées. Ceci est fait automatiquement par l'estimation des paramètres nécessaires à une distribution à partir d'un ensemble de mesures de disponibilité donné.

Nos travaux dans le domaine contribuent à aller un pas plus loin en donnant la possibilité d'émuler un environnement volatil proche de la réalité à partir d'une distribution statistique. Nous montrons ainsi comment utiliser notre injecteur de fautes pour exécuter une application distribuée dans un cluster ou une grille où le comportement des différents nœuds (i.e la volatilité) suit une distribution prédéfinie. Cette procédure est automatique, reproductible et permet de voir le comportement d'une application dans un environnement plus proche de la réalité.

Combiné aux travaux précédemment cités, cela permet d'avoir un outil complètement automatique qui extrait la disponibilité à partir de traces réelles, génère la distribution statistique correspondante et la reproduit sur un système. Ainsi, le comportement de l'application réelle peut être testé avant l'étape de production. Cela permet également de vérifier l'efficacité d'une application et de la configurer de manière optimale selon le type de système visé.

Dans la section 9.2, nous introduisons les travaux précédents qui ont été faits sur l'extraction d'une distribution à partir des traces et nous expliquons le choix de la distribution de Weibull. Dans la section 9.3, nous présentons la distribution choisie. Dans la section 9.4, nous expliquons l'incorporation de cette distribution dans notre outil FAIL-FCI. Finalement, dans la section 9.5, nous présentons les expérimentations que nous avons faites sur XtremWeb, l'intégration dans FCI du mécanisme d'émulation de prise en main de la machine par un utilisateur lambda et les résultats obtenus en utilisant cette distribution sur différentes configurations pour analyser l'exactitude du comportement induit des nœuds comparé à la distribution utilisée ainsi que l'efficacité de XtremWeb dans ces différentes configurations.

9.2 Etudes de la disponibilité des ressources

Dans cette section, nous allons nous concentrer sur les résultats obtenus par Daniel Nurmi, John Brevik et Rich Wolski sur la détermination automatique d'une distribution de disponibilité en faisant correspondre une distribution statistique à des données de disponibilité [58, 82, 83].

Dans leurs travaux, ils utilisent trois ensembles de données qu'ils pensent être représentatifs du comportement des hôtes actuellement en présence sur Internet. Le premier en-

semble de données a été obtenu à partir du laboratoire des étudiants en Informatique du UCSB's CSIL. Chaque mesure enregistre le temps à partir duquel une station de travail est capable d'exécuter un processus utilisateur jusqu'au moment où il n'en est plus capable (83 machines du laboratoire CSIL pendant 8 semaines). Le deuxième ensemble de données a été obtenu à partir d'une poule Condor exécutée à l'université du Wisconsin. Chaque mesure de disponibilité Condor correspond au temps à partir duquel le dispatcher Condor commence un des processus de monitoring jusqu'au moment où la station de travail allouée évince le processus (210 machines pendant une période de 6 semaines). Le troisième ensemble de données a été obtenu à partir d'un travail de Long, Muir et Golding dans lequel ils ont mesuré à distance la disponibilité de certains hôtes sur Internet (1170 machines sur une période de 3 mois).

Dans leur étude, ils ont testé l'adéquation de plusieurs distributions incluant l'exponentiel, l'hyperexponentiel, la Log-normal, la Pareto and la Weibull. Il ont pu ainsi conclure que les deux familles de distributions qui correspondent le mieux aux données qu'ils ont rassemblées sont la Weibull et l'hyperexponentielle. La distribution de Weibull est souvent utilisée pour modéliser la durée de vie d'un objet, incluant des composants physiques d'un système mais aussi pour modéliser la disponibilité de ressources informatiques. L'hyperexponentiel a été utilisée pour modéliser auparavant la disponibilité des machines spécialement quand les données observées nécessitent un modèle qui permet d'avoir une approximation d'une grande variété de formes.

Afin de faire correspondre la plupart des distributions statistiques utilisées aux données observées, ils ont implanté des scripts Matlab qui trouvent les paramètres MLE (Maximum Likelihood Estimation). Cependant, le problème de trouver les paramètres MLE pour l'hyperexponentiel tend à être numériquement trop difficile pour des ensembles de données très grands, c'est pourquoi ils utilisent un logiciel EMpht à la place.

A partir des résultats présentés dans leurs travaux, ils s'accordent empiriquement sur la supériorité des distributions Weibull ou hyperexponentielle dans la modélisation de la disponibilité des ressources.

Quelques autres travaux précédents ont utilisé la distribution exponentielle ou Pareto, mais leurs résultats montrent que les distributions Weibull et hyperexponentielle sont des choix plus fidèles à la réalité. En effet, ces distributions sont significativement meilleures pour capturer la distribution des temps de disponibilité comparée à la distribution exponential ou Pareto. De plus, les deux distributions peuvent être calculées automatiquement à partir des données de mesure de disponibilité.

L'hyperexponentielle, bien qu'elle corresponde généralement mieux aux données, est

significativement plus complexe que le modèle Weibull à cause 1) de son nombre de paramètres à estimer bien plus important, 2) du fait que le paramètre de phase doit être décidé itérativement et 3) de sa non compatibilité avec la méthode MLE utilisée pour les autres distributions.

La distribution Weibull, avec ses deux paramètres MLE-calculable et sa structure mathématique relativement simple, semble être un meilleur choix si la rapidité et la complexité d'obtention d'un modèle sont à prendre en compte. C'est pourquoi nous avons décidé d'utiliser la distribution Weibull pour nos tests.

9.3 La distribution Weibull

La distribution Weibull (créée par Waloddi Weibull) est une distribution de probabilité continue avec la fonction de densité suivante :

$$f(x) = \alpha\beta^{-\alpha}x^{\alpha-1}e^{-(x/\beta)^{\alpha}} \tag{9.1}$$

pour $x \geq 0$, et $f(x) = 0$ pour $x < 0$, où $\alpha > 0$ est le paramètre de *forme* et $\beta > 0$ est le paramètre *d'échelle* de la distribution. On peut noter que la fonction de densité générale de Weibull a un troisième paramètre θ pour la *localisation*. Quand $\theta = 0$, cela la réduit à la distribution à 2 paramètres.

La fonction de distribution cumulative pour la Weibull à 2 paramètres est la suivante :

$$F(x) = 1 - e^{-(x/\beta)^{\alpha}} \tag{9.2}$$

pour $x \geq 0$, et $F(x) = 0$ pour $x < 0$.

Pour une variable aléatoire donnée U tirée selon une distribution uniforme dans l'intervalle (0,1), alors la variable

$$X = \beta(-ln(U))^{1/\alpha} \tag{9.3}$$

a une distribution de Weibull avec les paramètres α et β qui suit la forme de la fonction de distribution cumulative.

9.4 La bibliothèque Weibull pour FAIL-FCI

La figure 9.1 montre un scénario standard FAIL utilisant la fonction prédéfinie de FAIL *FAIL_RANDOM* pour générer un temporisateur aléatoire suivant une distribution uniforme (ligne 3 du scénario). Quand ce temporisateur expire, alors l'application testée est définitivement interrompue en utilisant la commande FAIL *halt* (ligne 4 du scenario).

```
1   Daemon ADV2
2   { node 1 :
3         time_g timer = FAIL_RANDOM (0,60);
4         timer -> halt, goto 2;
5     node 2 : }
```

Fig. 9.1 – Scénario FAIL-FCI standard

La figure 9.2 montre comment utiliser la distribution Weibull au lieu de la distribution uniforme. La fonction adéquate de la bibliothèque Weibull doit être chargée (ligne 1 du scénario) et peut alors être utilisée comme une fonction prédéfinie de FAIL (ligne 5 du scénario).

```
1   function int weibull (int, int)
          in library "libfail/lib/weibull/weibull.so";
2
3   Daemon ADV2
4   { node 1 :
5         time_g timer = weibull (6, 12);
6         timer -> halt, goto 2;
7     node 2 : }
```

Fig. 9.2 – Scénario utilisant la bibliothèque Weibull

Les utilisateurs peuvent créer de nouvelles bibliothèques pour manipuler d'autre distributions de probabilité. Il est préférable d'écrire les bibliothèques en C/C++ (car les démons FAIL-FCI sont écrits en C/C++). Les fichiers d'en-tête des fonctions de probabilité et le chemin vers le fichier contenant la bibliothèque compilée doivent être fournis (comme le montre la ligne 1 de la figure 9.2). La meilleure façon d'utiliser une bibliothèque de distribution est de créer une fonction qui initialise le générateur de nombres

aléatoires et retourne un nombre suivant la distribution correspondante lors du premier appel, et qui retourne juste le nombre aléatoire lors de tous les autres appels (c'est le fonctionnement de la bibliothèque Weibull).

9.5 XtremWeb

9.5.1 Prise en charge de XtremWeb

Comme nous l'avons déjà vu dans la section 7.2.1, la politique d'activation définit la disponibilité d'un *worker* ; un *worker* est disponible quand il peut exécuter un *job* sans intrusion pour l'utilisateur de l'ordinateur. Le *activator.class* définit la politique d'activation (Quand le *worker* peut-il exécuter son *job* ?). Les différentes politiques ont également été présentées dans cette section.

Pour nos expérimentations, chaque démon FAIL-FCI doit manipuler son worker Xtrem Web local. Pour cela, nous avons décidé de créer un nouvel *Activator XtremWeb* : *File-Activator.class*. Cet Activator teste l'existence d'un fichier particulier. Si ce fichier existe, alors le worker peut exécuter son job ; dans le cas contraire, il ne le peut pas.

Nous avons ensuite créé deux nouvelles primitives d'injection de fautes pour FAIL-FCI qui permettent la création de ce fichier particulier (*xtremweb_ activate()*) et la suppression de celui-ci (*xtremweb_ desactivate()*). Ces primitives sont stockées dans une nouvelle bibliothèque FAIL-FCI. En utilisant ce mécanisme, un démon FAIL-FCI peut ainsi contrôler la disponibilité d'un worker pour émuler l'utilisation d'un ordinateur par son propriétaire.

9.5.2 Configuration expérimentale

Les tests ont été exécutés sur 79 machines de GriDeXplorer exécutant Linux (Ubuntu). Toutes les machines étaient équipées d'un bi-processeur de 64 bits d'une fréquence de 1994 Mhz chacun, d'une mémoire vive d'une capacité de 2 Gb, et étaient connectées grâce à un réseau Ethernet Gigabit. Le dispatcher était exécuté sur une machine particulière non soumise au scénario FAIL et une autre machine particulère a été utilisée pour synchroniser le lancement des tests.

9.5.3 Résultats

La figure 9.3 présente le scénario FAIL que nous avons utilisé pour les tests de XtremWeb avec simulation d'utilisateurs.

```
1   function int weibull (int, int)
              in library "libfail/lib/weibull/weibull.so";
2   function int xtremweb_activate ()
              in library "libfail/lib/xtremweb/xtremweb.so";
3   function int xtremweb_desactivate ()
              in library "libfail/lib/xtremweb/xtremweb.so";
4   spyfunc main;
5
6   Daemon ADV1
7   { time_g the_timer = 30;
8      node 1 : the_timer ->!go(G1), goto 2;
9      node 2 : }
10
11  Daemon ADV2
12  { time_g the_timer = 10;
13    node 1 : int active = 0;
14        before(main) -> continue,
                          active=xtremweb_desactivate(),goto 2;
15    node 2 :?go -> continue, goto 3;
16    node 3 : int active = 0;
17        the_timer -> active = xtremweb_activate(), goto 4;
18    node 4 : int active = 0;
19        always time_g timer = weibull (6, 128);
20        timer -> active = xtremweb_desactivate(), goto 3; }
21
22  Computer P1
23  { program = "ls"; daemon = ADV1; }
24
25  Group G1
26  { program = "xtremweb.worker.Worker -xwconfig
                /opt/XtremWeb-1.8.0/conf/xtremweb.worker.conf";
27    size = 77; daemon = ADV2; }
```

FIG. 9.3 – Scénario FAIL pour l'émulation de la disponibilité des nœuds.

L'automate *ADV2* sera exécuté sur toutes les machines exécutant un worker XtremWeb qui forment ainsi le groupe *G1*. L'automate *ADV1* sera exécuté sur une machine n'exécutant aucun composant de XtremWeb et servira uniquement à la synchronisation du lancement du test. En effet, lors de son exécution il enverra le message *go* aux machines du groupe *G1* (ligne 8). Lors de leur exécution, les automates du groupe *G1* positionnent la machine hôte dans l'état occupé grâce à la commande *xtremweb_desactivate* (ligne 14), *i.e.* la machine est occupée par un utilisateur, et se positionnent ensuite dans l'état 2 où ils attendent le message *go* pour réellement commencer le scénario (ligne 15).

Le scénario est donc implanté par l'état 3 et 4 de l'automate *ADV2*. L'état 3 correspond à l'état où la machine n'est pas disponible et, au bout de 30 secondes, la machine est repositionnée comme étant disponible (ligne 17) et l'automate passe dans l'état 4 qui correspond à l'état où la machine est disponible. Un timer est alors déterminé selon une distribution de Weibull (ligne 19) et lorsque celui-ci expire (ligne 20), la machine est repositionnée comme étant non disponible et l'automate retourne dans l'état 3. Ce processus sera ainsi répété jusqu'a la fin du test.

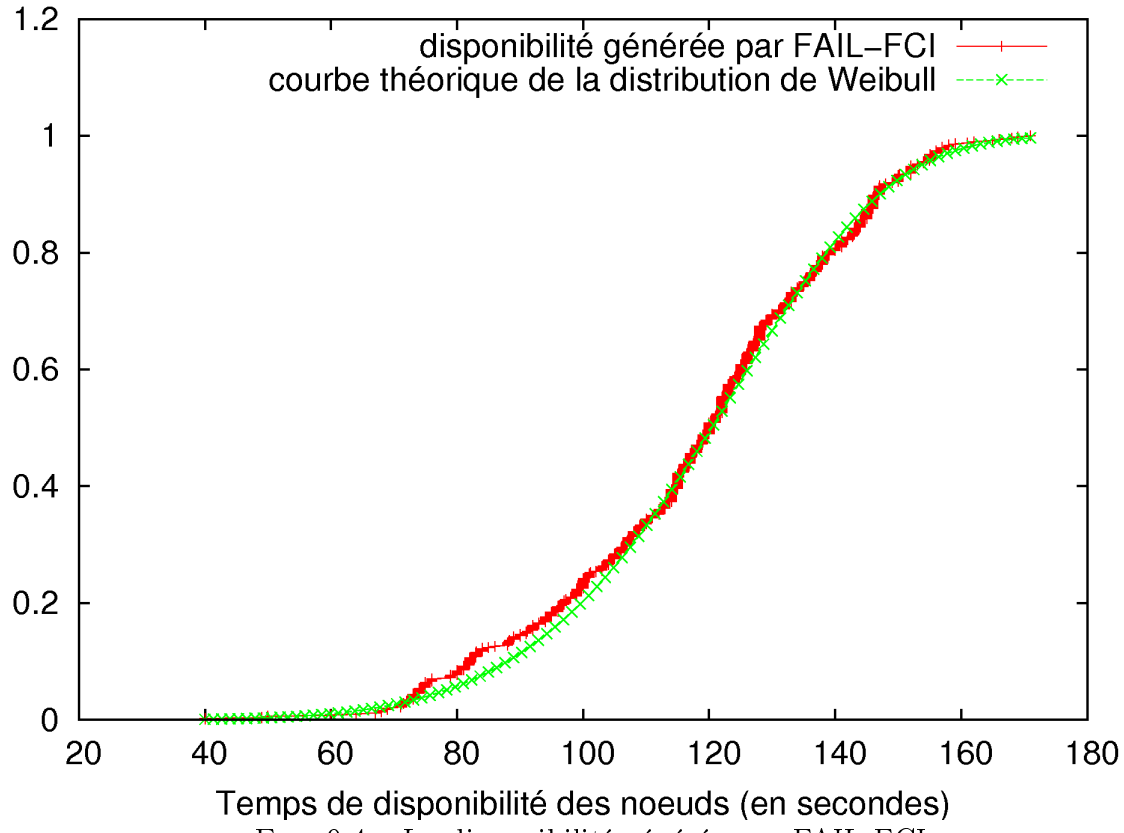

FIG. 9.4 – La disponibilité générée par FAIL-FCI

Ce scénario a été effectué à différentes échelles (en utilisant 10, 22, 45 et 77 machines),

mais ces échelles n'étant pas suffisamment grandes pour avoir un impact réel sur l'exécution de XtremWeb, seulement les resultats obtenus avec 77 machines seront présentés.

La figure 9.4 présente la distribution de la disponibilité des nœuds générée par FAIL-FCI comparée à la courbe théorique de la distribution de Weibull.

Comme on peut le constater, la courbe de disponibilité des machines est très proche de la courbe théorique.

La figure 9.5 présente la détection par XtremWeb de la disponibilité d'une machine (la disponibilité d'une machine étant celle générée par FAIL-FCI).

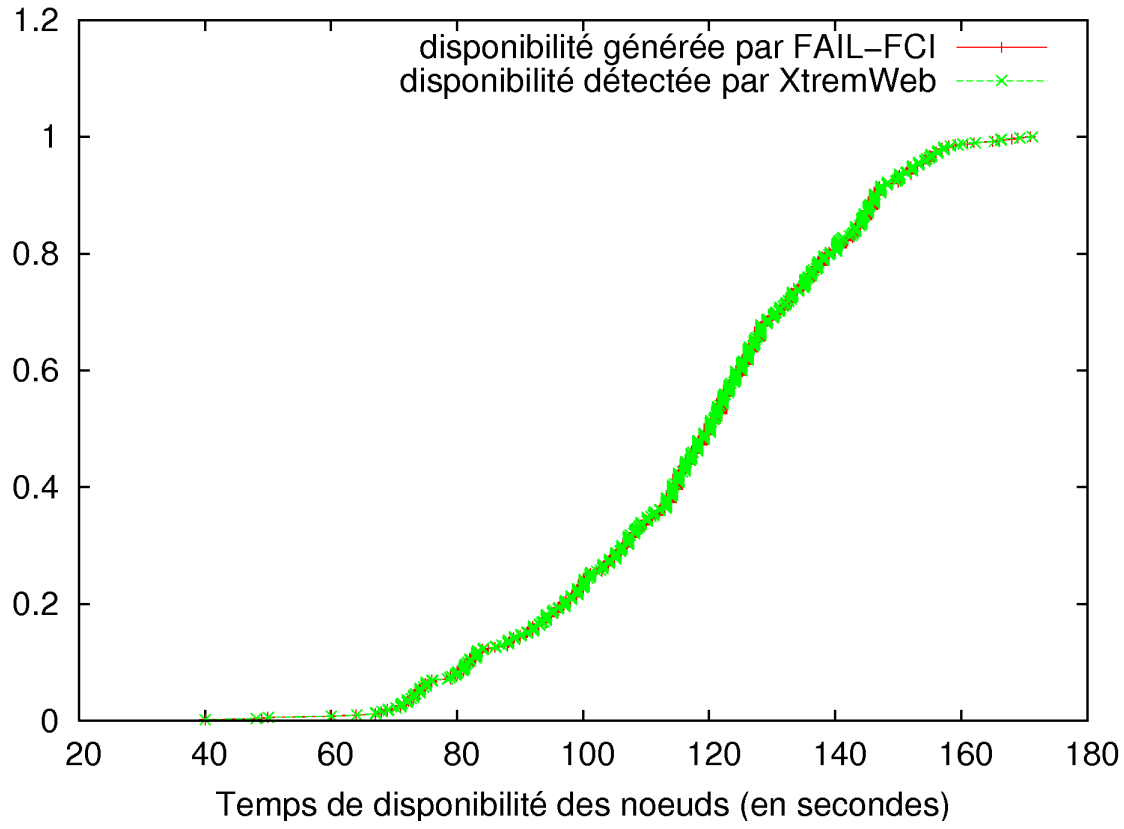

Fig. 9.5 – La disponibilié détectée par XtremWeb

Comme on peut le constater, la courbe représentant la disponibilité détectée par XtremWeb « colle » parfaitement à celle représentant la disponibilité générée par FAIL-FCI. En fait, le délai de détection d'un changement d'état d'une machine par XtremWeb lors des tests est d'environ 1 seconde. En effet, il y a un délai d'une seconde pour détecter qu'une machine est libre et d'une seconde pour détecter qu'elle ne l'est plus, ce qui explique pourquoi la disponibilité détectée par XtremWeb soit si proche de celle générée.

La figure 9.6 présente la distribution de la durée d'exécution des tâches. Cette courbe est à lire avec beaucoup de précaution. En effet, la distribution de la durée des tâches

comparée à la distribution de disponibilité n'a pas de sens si les tâches à exécuter ne sont pas infinies. Par exemple, pour une disponibilté des machines fixée à 100 secondes et des tâches nécessitant 10 secondes de calcul, la distribution de la durée des tâches sera de 10 secondes et n'aura strictement rien en commun avec la distribution de la disponibilité des machines. Lors des tests que nous avons effectués, chaque tâche à exécuter est suffisamment longue pour couvrir la disponibilité d'un nœud et le mécanisme de reprise des tâches après un arrêt dû à la prise en main de la machine par un utilisateur a été désactivé. Ainsi, lorsqu'une machine n'est plus disponible et qu'une tâche est alors arrêtée, elle est abandonnée et lorsque la machine est à nouveau disponible, une nouvelle tâche est exécutée. De cette façon, la distribution de la durée des tâches à une signification et, dans le meilleur des cas, se superpose à la distribution de disponibilité des machines.

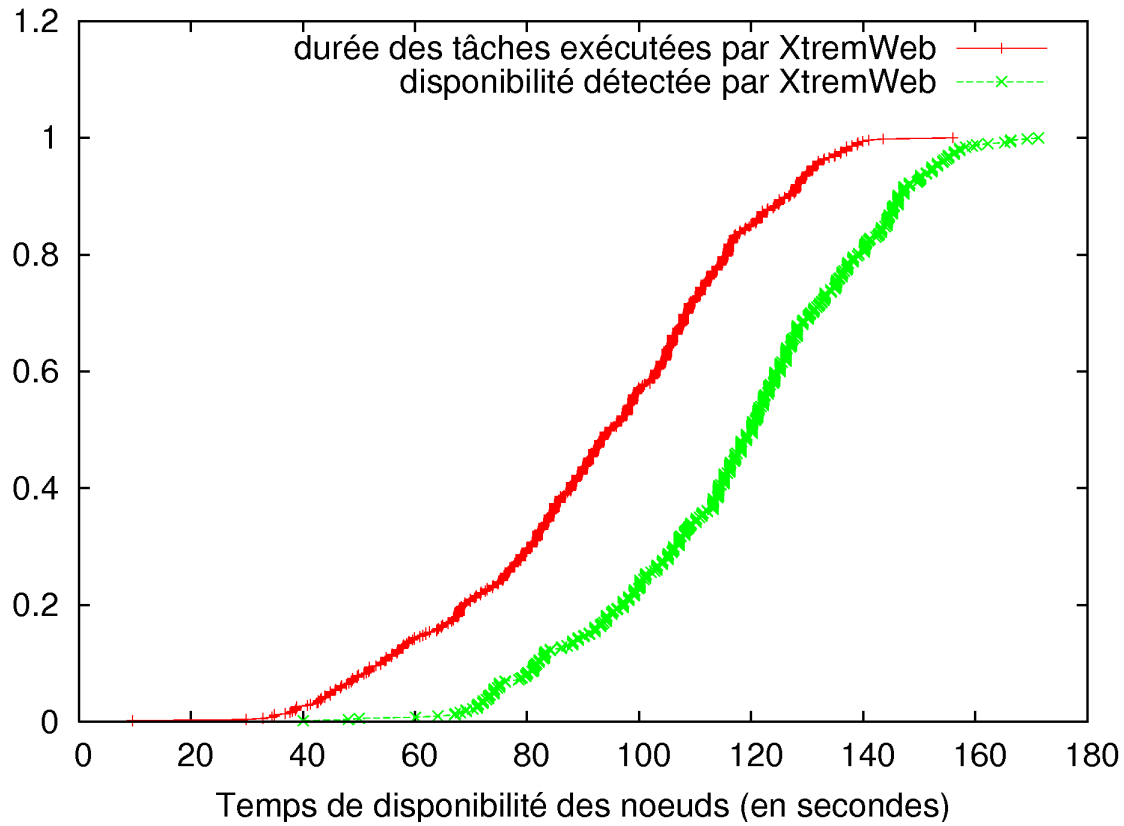

FIG. 9.6 – L'exécution des tâches par XtremWeb

Sur la figure, on remarque que la courbe de durée des tâches a la même forme que la courbe de disponibilité des machines mais avec un écart entre elles. Cette écart correspond au temps de latence entre le moment où la machine est disponible et le moment où une tâche est effectivement exécutée.

Cette latence est présentée dans la figure 9.7. Sur cette figure, on remarque que la

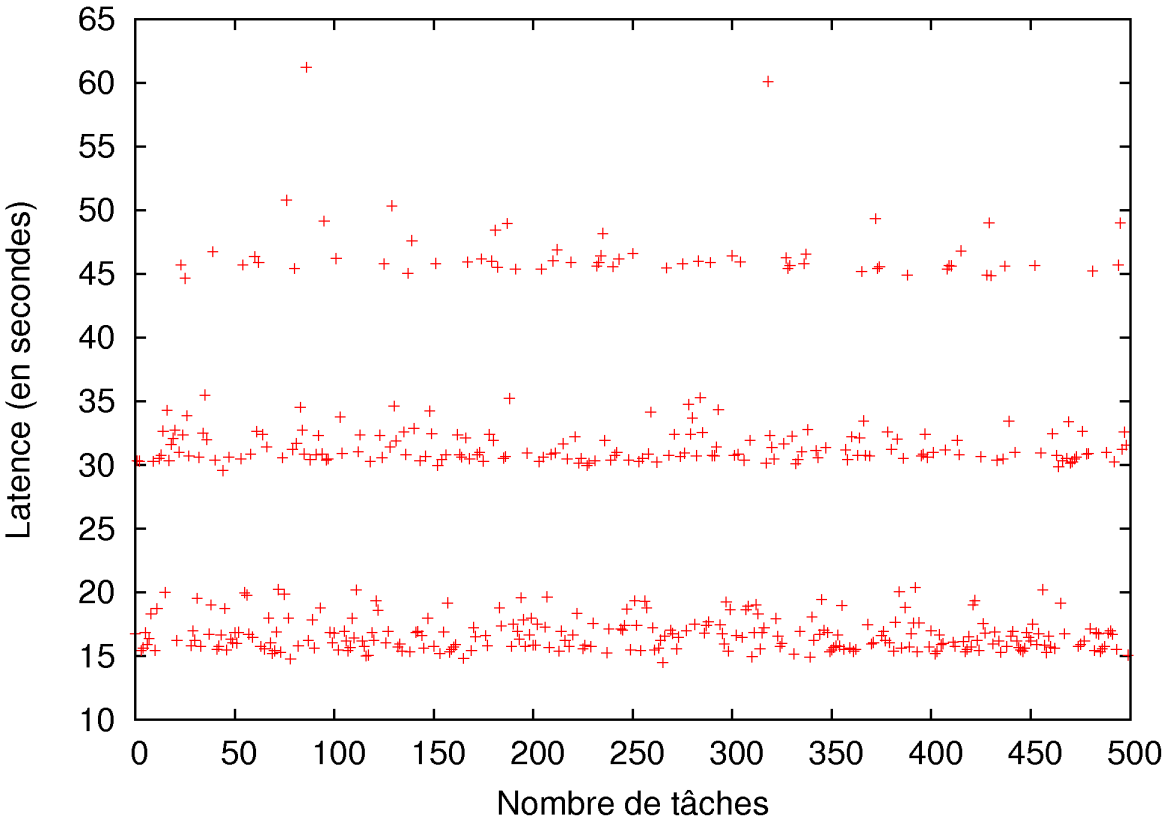

Fig. 9.7 – Latence avant l'exécution des tâches

latence varie entre 15 et 60 secondes avec une majorité des latences valant 15, 30 et 45 secondes. La latence moyenne étant de 25 secondes. La latence représente du temps processeur libre, *i.e.* non utilisé par un utilisateur, mais également non utilisé pour l'exécution d'une tâche par les worker xtremweb. Par contre, ce temps de latence inclut le temps nécessaire à la préparation de l'exécution d'une tâche (par exemple, le temps de téléchargement du binaire correspondant à la tâche à partir du dispatcher vers le worker). Dans nos tests ce temps de latence représente environ 21% de la disponibilité des nœuds. Ce pourcentage de temps inutilisé est dépendant de la configuration de XtremWeb, des tâches, mais également du nombre de machines exécutant un worker et de leur comportement. Dans nos tests, l'impact important de la latence est dû principalement à la grande volatilité des nœuds qui a été générée par FAIL-FCI.

9.6 Conclusion

Bien que les tests que nous avons faits sur la simulation des utilisateurs ne soient que des tests préliminaires et n'apportent pas d'informations concrètes sur le fonctionnement

et l'impact de la configuration de XtremWeb, ils nous montrent que FAIL-FCI peut être utilisé pour simuler des utilisateurs dans un réseau et comment le faire.

Le but des tests est de vérifier le comportement de XtremWeb face à une distribution de disponibilité donnée, correspondant à un certain type de réseaux et d'utilisateurs, telle qu'une salle informatique d'un collège, d'une entreprise, etc, et de déterminer la configuration optimale de XtremWeb pour utiliser au mieux les machines lorsqu'elles sont libres. La configuration d'une application basée sur le vol de cycle peut également jouer sur son intrusion lorsqu'un utilisateur « à la main » et il faut, dans dans ce cas, déterminer le juste compromis entre intrusion et performance. D'autres tests doivent être faits en modifiant la configuration de XtremWeb et en utilisant d'autres distributions de disponibilité.

En effet, combiné à un outil permettant d'extraire une distribution à partir de traces (comme celui présenté dans [58, 82, 83]), notre outil permet alors de déterminer la configuration optimal d'une application basée sur le vol de cycles face à un type de comportement réaliste des utilisateurs en générant un comportement des machines suivant cette distribution lors des tests. Ainsi, il permet de calibrer l'application en fonction du type du réseau où elle va être utilisée et du comportement des utilisateurs qui vont partager les machines avec elle.

Chapitre 10

Conclusions et Perspectives

Bien que l'injection de fautes ne soit pas une technique nouvelle, notamment pour tester et vérifier qu'une implantation d'un système satisfait sa spécification, son application aux systèmes distribués n'a été abordée que très récemment. En effet, l'injection de fautes dans de tels systèmes est un réel défi du fait de la difficulté de coordonner l'injection de fautes et les différents processus du système. Bien que quelques outils aient été développés, ils nécessitent un investissement considérable en temps pour les utilisateurs, généralement à cause de la phase d'implantation, par ces derniers, de l'injecteur de fautes spécifique à l'application testée, mais aussi en raison de la difficulté, voire de l'incapacité à spécifier un scénario de fautes suffisamment élaboré. C'est pour cela que nous avons opté pour la création d'un langage dédié à la spécification de scénarios de fautes pour l'émulation d'applications distribuées dans un environnement de type GriDeXplorer.

Ce langage, appelé FAIL (FAult Injection Language), permet la spécification de scénarios de fautes complexes et potentiellement probabilistes ou reproductibles sans aucune modification du code source de l'application instrumentée. Le choix d'une interface avec un débogueur pour injecter les fautes nous permet de faciliter le travail de l'utilisateur et de ne pas réclamer la modification du code source. En effet, l'utilisateur n'a qu'à écrire le scénario de fautes en FAIL et aucune autre implantation ne lui est demandé. Bien que le type de fautes injectées soit actuellement limité au crash d'un processus, à son relancement ou à son arrêt momentané, on peut envisager (grâce au choix d'une injection de fautes par débogage) d'autres types de fautes, mais cela dépasse le cadre de cette thèse.

Nous avons ainsi implanté une plate-forme d'injection de fautes, nommée FCI, utilisant FAIL comme langage de description de scénario de fautes. Un compilateur du langage FAIL permet de générer le code nécessaire à l'exécution du scénario de fautes et un outil, nommé *fcideploy* permet d'automatiser l'étape de configuration, de compilation du code

généré et d'exécution du scénario de fautes.

Des tests du surcoût induit par l'utilisation de FCI ont été réalisés afin de déterminer l'impact de notre solution sur le temps d'exécution de l'application distribuée. Nous avons ainsi montré que le modèle « event-driven » de notre injecteur de fautes limite son impact, *i.e.* le surcoût calculé, sur le temps d'exécution de l'application à moins de 0.31% pour les cas considérés.

Ensuite des tests d'injection de fautes ont été réalisés pour étudier la tolérance aux fautes de différentes applications distribuées telles que XtremWeb, MPICH-V et FreePastry en injectant des fautes effectives grâce à FAIL-FCI.

Des tests d'injection de fautes quantitatives et qualitatives ont été réalisées sur la version C++ de XtremWeb. Les resultats que nous avons obtenus sur XtremWeb montre qu'il souffre d'un problème de gestion de ressources, au niveau du dispatcher quand des workers sont victimes de défaillances, entraînant sa propre défaillance. On peut aussi déduire de l'expérimentation que XtremWeb devrait implémenter un mécanisme pour faire attention aux défaillances des workers quand une tâche est soumise car lorsque des fautes apparaissent juste après la réception d'une tâche, les performances sont plus mauvaises que lorsqu'elles apparaissent juste après la fin d'un calcul (même si dans ce cas le worker a perdu du temps à exécuter du calcul inutilement).

Des tests d'injection de fautes ont été effectués sur MPICH-V afin de déterminer l'impact de la fréquence des fautes, de l'echelle et de l'injection de fautes simultanées sur les performances de celui-ci. On a pu ainsi constater une grande variance dépendante du moment de l'injection de fautes comparé à la dernière vague de *checkpoint*. Nous avons pu également découvrir un bogue. Grâce à la pruissance d'expressivité de FAIL et à une série de tests visant à découvrir la configuration du système qui menait à l'apparition de ce bogue, nous avons pu le localiser précisément, ce qui a permis la correction de ce bogue dans une nouvelle version de MPICH-Vcl.

Des tests d'injection de fautes ont également été faits sur FreePastry et montrent que la periodicité et la probabilité d'apparition de fautes sont peu importantes par rapport à la maintenance du réseau logique, *i.e.* le réseau FreePastry est capable de se reconstruire dans tous les cas, mais que l'echelle compte. En effet, quand le nombre de nœuds potenciellement défaillants augmente, le réseau peut être partitionné en plusieur réseaux.

Des tests de stress ont également été effectués sur XtremWeb, pour montrer que notre outil pouvait également être utilisé pour le stress d'applications comme les tests que nous avons effectués en utilisant l'outil de stress QUAKE en collaboration avec l'équipe qui l'a dévéloppé.

Des tests de simulation d'utilisateur ont été effectués sur la version officielle de XtremWeb, écrite en Java, afin de montrer que notre outil pouvait être utilisé pour simuler des utilisateurs dans un réseau et afin de montrer comment le faire. En effet, combiné à un outil permettant d'extraire une distribution à partir de traces (comme celui présenté dans [58, 82, 83]), notre outil permet alors de déterminer la configuration optimale d'une application basée sur le vol de cycles face à un type de comportement réaliste des utilisateurs en générant un comportement des machines suivant cette distribution lors des tests. Ainsi, il permet de calibrer l'application en fonction du type du réseau où elle va être utilisée et du comportement des utilisateurs qui vont partager les machines avec elle.

Dans toutes ces expériences, nous avons vu comment utiliser le même outil pour l'injection de fautes, le test de l'application par stress de celle-ci et la simulation d'utilisateurs dans un réseaux. Les injections de fautes peuvent être faites en utilisant une approche *quantitative* (comme dans la plupart des études en relation), mais elles peuvent aussi être faites en utilisant une approche *qualitative* plus originale, où les fautes sont injectées à un moment précis de l'état logique de l'application testée.

Bien que l'ensemble des fautes injectées possibles soit très large, le langage qui décrit les scénarios de fautes est de haut niveau et indépendant du langage utilisé dans l'application testée. Cela permet de séparer le travail des programmeurs de l'application et des testeurs, pour que l'expertise de chacun soit utilisée dans le domaine adéquat.

Perspectives Les autres pistes pour l'avenir seraient l'utilisation de notre injecteur de fautes dans des systèmes à plus grande échelle, typiquement en utilisant la virtualisation au sein de la plate-forme Grid5000 pour permettre, par exemple, l'étude d'applications pair-à-pair dans un environnement encore plus proche de la réalité (en terme de nombre de nœuds). En effet, combiné aux mécanismes de simulation d'utilisateurs dans un réseau, présentés dans le chapitre 9, l'environnement ainsi déployé serait alors très proche d'un environnement pair-à-pair déployé sur Internet. Un autre point important dans l'étude du comportement des machines est l'étude des virus. En effet, une telle étude nous permettrait de générer une injection de fautes corrélées comme celles qui se produisent quand un virus est diffusé à travers le réseau et permettrait alors de vérifier le comportement des applications dans un tel environnement. Ensuite, notre étude s'est cantonné à l'injection de fautes très simples de type *crash*. Il serait très intéressant d'étudier d'autres types de fautes tels que la corruption mémoire (qui pourrait être implantée en utilisant le débogueur) ou la corruption des messages comme dans Orchestra [24, 25, 26, 27, 28]. Enfin, il serait aussi intéressant de définir un *Benchmark* de tolérance aux fautes en spécifiant une batterie de scénarios de fautes qui servirait alors de base de comparaison pour

les applications tolérantes aux fautes comme les *NAS Parallel Benchmark* servent de base de comparaison de performance pour les applications de calcul haute performance.

Bibliographie

[1] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Goofi : Generic object-oriented fault injection tool. In *Proc. International Conference on Dependable Systems and Networks (DSN 2001)*, 2001.

[2] Rajeev Alur. *Techniques for automatic verification of real-time systems.* PhD thesis, Stanford University, Stanford, CA, USA, 1992.

[3] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *ICALP '90 : Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, pages 322–335, London, UK, 1990. Springer-Verlag.

[4] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2) :183–235, 1994.

[5] Guillermo A. Alvarez and Flaviu Cristian. Centralized failure injection for distributed, fault-tolerant protocol testing. In *International Conference on Distributed Computing Systems*, pages 0–, 1997.

[6] David P. Anderson. Boinc : A system for public-resource computing and storage. In *GRID '04 : Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[7] Apache axis, `http ://ws.apache.org/axis`.

[8] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS Parallel Benchmarks 2.0. Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, 1995.

[9] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM*, 12(5) :260–261, 1969.

[10] Nathalie Bertrand. *Modèles stochastiques pour les pertes de messages dans les protocoles asynchrones et techniques de vérification automatique.* Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France, septembre 2006.

[11] Aurélien Bouteiller, Franck Cappello, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier, and Frédéric Magniette. MPICH-V2 : a fault tolerant MPI for volatile nodes based on pessimistic sender ba sed message logging. In *High Performance Networking and Computing (SC2003)*. Phoenix USA, IEEE/ACM, November 2003.

[12] Patricia Bouyer. *Modèles et algorithmes pour la vérification des systèmes temporisés*. Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France, avril 2002.

[13] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2) :323–342, 1983.

[14] S. Budkowski and P. Dembinski. An introduction to estelle : a specification language for distributed systems. *Comput. Netw. ISDN Syst.*, 14(1) :3–23, 1987.

[15] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Herault, Frédéric Magniette, Vincent Néri, and Oleg Lodygensky. Computing on large-scale distributed systems : Xtrem web architecture, programming models, security, tests and convergence with grid. *Future Gener. Comput. Syst.*, 21(3) :417–437, 2005.

[16] J. Carreira, H. Madeira, and J. Silva. Xception : Software fault injection and monitoring. In *Processor Functional Units, in Pre-prints 5th Int. Working Conf. on Dependable Computing for Critical Applications (DCCA-5), (Urbana-Champaign, IL, USA)*, pages 35–49, 1995.

[17] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Security for structured peer-to-peer overlay networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02)*, Boston, MA, December 2002.

[18] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Microsoft Research, May 2002.

[19] R. Chandra, R. M. Lefever, M. Cukier, and W. H. Sanders. Loki : A state-driven fault injector for distributed systems. In *Proc. of the Int. Conf. on Dependable Systems and Networks*, June 2000.

[20] Ramesh Chandra, Michel Cukier, Ryan M. Lefever, and William H. Sanders. Dynamic node management and measure estimation in a state-driven fault injector. In *Symposium on Reliability in Distributed Software*, pages 248–257, 2000.

[21] K. M. Chandy and L.Lamport. Distributed snapshots : Determining global states of distributed systems. In *Transactions on Computer Systems*, volume 3(1), pages 63–75. ACM, February 1985.

[22] Michel Cukier, Ramesh Chandra, David Henke, Jessica Pistole, and William H. Sanders. Fault injection based on a partial view of the global state of a distributed system. *srds*, 00 :168, 1999.

[23] Perrin D. and Pin J.E. Infinite words, 2001. available on : http ://www.liafa.jussieu.fr/ jep/resumes/infinitewords.html, 2001.

[24] S. Dawson, F. Jahanian, and T. Mitton. Orchestra : A fault injection environment for distributed systems. In *26th International Symposium on Fault-Tolerant Computing (FTCS)*, pages 404–414, Sendai, Japan, June 1996.

[25] Scott Dawson and Farnam Jahanian. Deterministic fault injection of distributed systems. In *Dagstuhl Seminar on Distributed Systems*, pages 178–196, 1994.

[26] Scott Dawson and Farnam Jahanian. Probing and fault injection of protocol implementations. In *International Conference on Distributed Computing Systems*, pages 351–359, 1995.

[27] Scott Dawson, Farnam Jahanian, and Todd Mitton. A software fault injection tool on real-time mach. In *IEEE Real-Time Systems Symposium*, pages 130–140, 1995.

[28] Scott Dawson, Farnam Jahanian, Todd Mitton, and Teck-Lee Tung. Testing of fault-tolerant and real-time distributed systems via protocol fault injektion. In *Symposium on Fault-Tolerant Computing*, pages 404–414, 1996.

[29] EINSTEIN@home. http ://einstein.phys.uwm.edu.

[30] G. Fagg and J. Dongarra. FT-MPI : Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *7th Euro PVM/MPI User's Group Meeting2000*, volume 1908 / 2000, Balatonfüred, Hungary, september 2000. Springer-Verlag Heidelberg.

[31] G. Fedak, C. Germain, V. Néri, and F. Cappello. Xtremweb : A generic global computing system. In *Proceedings of IEEE Int. Symp. on Cluster Computing and the Grid*, 2001.

[32] `http ://freepastry.rice.edu`.

[33] Globus toolkit, `http ://www.globus.org/toolkit`.

[34] Grid5000, `https ://www.grid5000.fr/mediawiki/index.php/Grid5000 :Home`.

[35] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI message passing interface sta ndard. *Parallel Computing*, 22(6) :789–828, September 1996.

[36] S. Han, K. Shin, and H. Rosenberg. Doctor : An integrated software fault injection environment for distributed real-time systems. In *Int. Computer Performance and Dependability Symp. (IPDS'95), Erlangen, Germany*, pp.204-13, IEEE Computer Society Press, 1995.

[37] D. Henke. Loki – an empirical evaluation tool for distributed systems : The experiment analysis framework. Master's thesis, University of Illinois, Urbana, IL, 1998.

[38] T. Herault, P. Lemarinier, A. Bouteiller, and F. Cappello. The MPICH-V project `http ://www.mpich-v.net/`.

[39] William Hoarau, Pierre Lemarinier, Thomas Herault, Eric Rodriguez, Sébastien Tixeuil, and Franck Cappello. Fail-mpi : How fault-tolerant is fault-tolerant mpi ? In *Proceedings of Cluster 2006, Barcelona, Spain*, September 2006.

[40] William Hoarau, Luis Silva, and Sébastien Tixeuil. Integrated research in grid computing, chapter fault-injection and dependability benchmarking for grid computing middleware. *CoreGRID*, Springer Verlag 2006.

[41] William Hoarau and Sébastien Tixeuil. A language-driven tool for fault injection in distributed applications. In *Proceedings of the IEEE/ACM Workshop GRID 2005*, November 2005. also available as LRI Research Report 1399, february 2005, at `http ://www.lri.fr/~hoarau/fail.html`.

[42] William Hoarau and Sébastien Tixeuil. Easy fault injection and stress testing with fail-fci. *In Second CoreGRID Workshop on Grid and Peer to Peer Systems Architecture, Paris, France*, January 2006.

[43] William Hoarau, Sébastien Tixeuil, Nuno Rodrigues, Décio Sousa, and Luis Silva. Integrated research in grid computing, chapter benchmarking the ogsa-dai middleware. Springer Verlag 2007.

[44] William Hoarau, Sébastien Tixeuil, and Fabien Vauchelles. Fault injection in distributed java applications. In *Proceedings of the International Workshop on Java for Parallel and Distributed Computing (joint with IPDPS 2006), Greece*, page to appear, April 2006.

[45] William Hoarau, Sébastien Tixeuil, and Fabien Vauchelles. Fail-fci : Versatile fault-injection. In *Future Generation Computer Systems*, 2007.

[46] Wolfgang Hoschek. *A Unified Peer-to-Peer Database Framework for XQueries over Dynamic Distributed Content and its Application for Scalable Service Discovery.* PhD thesis, Technical University of Vienna, March 2002.

[47] E. Roman J. Duell, P. Hargrove. The design and implementation of berkeley lab's linux checkpoint/restart. Technical Report publication LBNL-54941, Berkeley Lab, 2003.

[48] Jakarta tomcat, `http ://jakarta.apache.org/tomcat`.

[49] Eric Jenn, Jean Arlat, Marcus Rimen, Joakim Ohlsson, and Johan Karlsson. Fault injection into VHDL models : The MEFISTO tool. In *Proceedings of the 24th International Symposium on Fault Tolerant Computing, (FTCS-24), IEEE, Austin, Texas, USA*, pages 66–75, 1994.

[50] D. Kondo, M. Taufer, C. Brooks, H. Casanova, and A. Chien. Characterizing and evaluating desktop grids : An empirical study. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'04)*, April 2004.

[51] Derrick Kondo, Gilles Fedak, Franck Cappello, Andrew A. Chien, and Henri Casanova. On resource volatility in enterprise desktop grids. In *E-SCIENCE '06 : Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, page 78, Washington, DC, USA, 2006. IEEE Computer Society.

[52] Pierre Lemarinier, Aurélien Bouteiller, Thomas Herault, Géraud Krawezik, and Franck Cappello. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In *IEEE International Conference on Cluster Computing (Cluster 2004)*. IEEE CS Press, 2004.

[53] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the condor distributed processing sy stem. Technical Report 1346, University of Wisconsin-Madison, 1997.

[54] S. Lumetta and D. Culler. The mantis parallel debugger. In *Proceedings of SPDT'96 : SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 118–126, Philadelphia, Pennsylvania, May 1996.

[55] H. Madeira, J. Carreira, and J. Silva. Injection of faults in complex computers. In *IEEE Workshop on Evaluation Techniques for Dependable Systems*, San Antonio, Texas, October 1995.

[56] Henrique Madeira, Mario Zenha Rela, Francisco Moreira, and Joao Gabriel Silva. Rifle : A general purpose pin-level fault injector. In *European Dependable Computing Conference*, pages 199–216, 1994.

[57] P. Maymounkov and D. Mazieres. Kademlia : A peer-to-peer information system based on the xor metric. In *Proceedings of IPTPS02*, Cambridge, USA, March 2002.

[58] D. Nurmi, J. Brevik, and R. Wolski. Modeling machine availability in enterprise and wide-area distributed computing environments. Technical Report CS2003-28, U.C. Santa Barbara Computer Science Department, October 2003.

[59] Ogsa-dai, `http ://www.ogsadai.org.uk`.

[60] Projects that use ogsa-dai, `http ://www.ogsadai.org.uk/about/projects.php`.

[61] G. Fedak P. Malecot, D. Kondo. Xtremlab : une plateforme pour l'observation et la caracterisation des grilles de pc sur internet, RENPAR 2006.

[62] Jan K. Pachl. Protocol description and analysis based on a state transition model with channel expressions. In *Proceedings of the IFIP WG6.1 Seventh International Conference on Protocol Specification, Testing and Verification VII*, pages 207–219, Amsterdam, The Netherlands, The Netherlands, 1987. North-Holland Publishing Co.

[63] J. Pistole. Loki–an empirical evaluation tool for distributed systems : The run-time experiment framework. Master's thesis, University of Illinois, Urbana, IL, 1998.

[64] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01 : Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[65] Antony Rowstron and Peter Druschel. Pastry : Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218 :329– ? ?, 2001.

[66] Grzegorz Rozenberg and Arto Salomaa. *Handbook of Formal Languages*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[67] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Ja son Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework : System-initiated checkpointing. In *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

[68] D.C. Schmidt. Reactor : An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. In *Proceedings of the 1st Pattern Languages of Programs Conference*, August 1994.

[69] D.C. Schmidt and S.D. Huston. *C++ Network Programming : Mastering Complexity Using ACE and Patterns*. Addison-Wesley Longman, 2002. ISBN 0-201-60464-7.

[70] D.C. Schmidt and S.D. Huston. *C++ Network Programming : Systematic Reuse with ACE and Frameworks*. Addison-Wesley Longman, 2003. ISBN 0-201-79525-6.

[71] Specification and description language (sdl), ITU-T Recommendation Z.100, International Telecommunication Union, November 1999.

[72] Michael Shirts and Vijay S. Pande. Screen savers of the world unite ! In *Sciences, vol. 290*, pages pp. 1903–1904, 2000.

[73] Volkmar Sieh. Fault-injector using unix ptrace interface. Internal Report 11/93, IMMD3, Universitt Erlangen-Nrnberg, 1993.

[74] Luis Silva, Henrique Madeira, and Joao Gabriel Silva. Software aging and rejuvenation in a soap-based server. In *NCA '06 : Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications*, pages 56–65, Washington, DC, USA, 2006. IEEE Computer Society.

[75] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI : The Complete Reference*. The MIT Press, 1996.

[76] Parasoft soatest, `http ://www.parasoft.com`.

[77] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[78] D.T. Stott and al. Nftape : a framework for assessing dependability in distributed systems with lightweight fault injectors. In *In Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pages 91–100, March 2000.

[79] W. T. Sullivan, III, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A new major SETI project based on Project SERENDIP data and 100,000 personal computers. In C. Batalli Cosmovici, S. Bowyer, and D. Werthimer, editors, *IAU Colloq. 161 : Astronomical and Biochemical Origins and the Search for Life in the Universe*, pages 729–+, January 1997.

[80] Pushtotest testmaker, `http ://www.pushtotest.com`.

[81] Uwe Wildner. Swifler : Software implemented control flow error injection, 1996.

[82] R. Wolski, D. Nurmi, and J. Brevik. An analysis of availability distributions in condor. In *Workshop on Next-Generation Software (w/IPDPS)*, March 2007.

[83] Rich Wolski, Daniel Nurmi, John Brevik, Henri Casanova, and Andrew Chien. Models and modeling infrastructures for global computational platforms, ipdps, p. 224a, 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 10, 2005.

[84] Web services interoperability (ws-i), `http ://www.ws-i.org`.

[85] Web services resource framework (wsrf), `http ://www.oasis-open.org/committees/tc home.php ?wg abbrev=wsrf`.

[86] V.C. Zandy. libckpt, 2005. http ://www.cs.wisc.edu/˜zandy/ ckpt/.

[87] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry : A resilient global-scale overlay for service deployment. In *IEEE Journal on Selected Areas in Communications*, 2003.

Annexe A

La Grammaire de FAIL

Les blancs

Les caractères espace, retour chariot et tabulation sont considérés comme blancs. Ils servent uniquement à séparer les différents mots. Ils n'ont aucun impact sur la grammaire dans FAIL et peuvent donc être insérés un nombre quelconque de fois entre n'importe quels mots.

Les commentaires

Les commentaires peuvent être écrits en étant insérés entre les signes "/*" et "*/". On peut aussi mettre une ligne du code en commentaire grâce aux signes "//". Le commentaire débute alors aux signes "//" et se termine à la fin de la ligne (délimitée par un caractère retour chariot). Les commentaires dans FAIL peuvent être imbriqués.

Les mots clés

Les mots clés du langage FAIL sont les suivants (triés alphabétiquement) : after, always, before, bool, Computer,continue, daemon, Daemon, false, function, goto, Group, halt, in, init, int, library, ln, node, one, program, restart, size, spyfunc, stop, tabc, time_g, time_l, true.

Les identificateurs (*ident*)

Les identificateurs sont constitués d'une suite non vide de caractères dont le premier est un alphabétique (masjuscule ou minuscule) et dont tous les autres sont alphanumériques ou '_'. Leur longueur est arbitraire. Les mots clés de FAIL ne peuvent pas être acceptés comme identificateurs.

Les constantes (*iconst*, *bconst*)

Les constantes sont les constantes entières (*iconst*) et les constantes booléennes (*bconst*).

- une constante entière est une suite non vide de chiffres.
- les constantes booléennes sont “true” et “false”.

Les chaines de caractères (*string*)

Une chaine de caractères commence et se termine par le caractère '"'. Elle contient une suite de caractères autre que '\\' et '"' ou une séquence spéciale.

Les séquences spéciales sont :

- '\\' (le caractère '\'),
- '\ ' (le caractère espace ' '),
- '\"' (le caractère '"').

La grammaire FAIL

La syntaxe acceptée par le langage FAIL est donnée par la grammaire BNF (Backus-Naur Form) suivante, dont le symbole de départ est *prog* :

prog : := ε
| *prog* *decl*

decl : := function *type* *ident* (*type_list*) in library *string* ;
| spyfunc *ident* ;
| Daemon *ident* { *automate* }
| Computer *ident_list* { *comp_decl* }
| Group *ident* { *group_decl* }

comp_decl : := ε
| *comp_decl program_eq* ;
| *comp_decl daemon_eq* ;

group_decl : := ε
| *group_decl size_eq* ;
| *group_decl program_eq* ;
| *group_decl daemon_eq* ;

ident_list : := *ident*
| *ident_list* , *ident*

type_list : := ε
| *type_list* , *type*

type : := int
| bool
| time_g
| time_l
| tabc

size_eq : := size = *iconst*

program_eq : := program = *string*

daemon_eq : := daemon = *ident*

automate : := *inits_always regles nodes*

inits_always : := ε
| *inits_always in_all*

in_all : := *ident* = *expr* ;
| *ident* = *string* : *iconst* ;
| int *ident* = *expr* ;
| bool *ident* = *expr* ;
| ln *ident* = *string* : *iconst* ;
| tabc *ident* = *expr* ;
| once int *ident* = *expr* ;
| once bool *ident* = *expr* ;
| once ln *ident* = *string* : *iconst* ;
| once tabc *ident* = *expr* ;
| once time_l *ident* = *expr* ;
| once time_g *ident* = *expr* ;

| once *ident* = *expr* ;
| once *ident* = *string* : *iconst* ;
| init *regle* ;
| time_g *ident* = *expr* ;
| always int *ident* = *expr* ;
| always bool *ident* = *expr* ;
| always time_l *ident* = *expr* ;
| always time_g *ident* = *expr* ;
| always tabc *ident* = *expr* ;
| always *ident* = *expr* ;

nodes : := ε
| *nodes node*

node : := node *iconst* : *inits_always regles*

regles : := ε
| *regles regle*

regle : := *garde* -> *actions* ;

expr_list : := ε
| *expr*
| *expr_list* ; *expr*

expr : := *expr* * *expr*
| *expr* / *expr*
| *expr* mod *expr*
| *expr* + *expr*
| *expr* - *expr*
| *expr* && *expr*
| *expr* || *expr*
| - *expr*
| *iconst*
| *bconst*

| *ident*
| *ident* (*expr_list*)

garde : := *entite*
| *garde* && *entite*

entite : := ? *ident*
| ? *ident* : *iconst*
| ? *ident* : *ident*
| *ident* == *expr*
| *ident* <> *expr*
| *ident* < *expr*
| *ident* <= *expr*
| *ident* > *expr*
| *ident* >= *expr*
| *ident*
| *bconst*
| before (*ident*)
| after (*ident*)

actions : := *action*
| *actions* ; *action*

action : := ! *ident dests*
| ! *ident* : *expr dests*
| ! *ident* : *expr*
| ! *ident*
| stop
| continue
| halt
| restart
| *ident* = *expr*
| goto *iconst*

dests : := (*ident*)

| (*ident* [*expr*])
| (*ident* [*muti*])

multi : := *expr* .. *expr*
| *multi* , *expr* .. *expr*