THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE
ET DE L'UNIVERSITÉ DE SFAX

Délivré par *l'Université Toulouse III - Paul Sabatier*
et la Faculté des Sciences Économiques et de Gestion - Sfax

Discipline : Informatique

Présentée et soutenue par

Riadh BEN HALIMA

Le Mercredi 17 Juin 2009

Conception, implantation et expérimentation d'une architecture en bus pour l'auto-réparation des applications distribuées à base de services web

---

JURY

Présidente

Me.	Michelle SIBILLA	Professeur à l'Université Paul Sabatier, Toulouse III

Rapporteurs

Me.	Laurence DUCHIEN	Professeur à l'Université de Lille
M.	Ahmed HADJ KACEM	Professeur à la FSEG-Sfax, Tunisie

Examinateurs

Me.	Louise TRAVÉ-MASSUYÈS	Directeur de Recherche, LAAS-CNRS
M.	Christophe CHASSOT	Professeur à l'INSA de Toulouse
M.	Mohamed Karim GUENNOUN	Professeur Assistant à l'EHTP, Casablanca, Marroc

Directeurs de thèse

M.	Khalil DRIRA	Chargé de Recherche au LAAS-CNRS
M.	Mohamed JMAIEL	Professeur à l'ENI-Sfax, Tunisie

Laboratoire d'Architecture et d'Analyse des Systèmes
***
LAAS-CNRS

Unité de Recherche en Développement et Contrôle d'Applications Distribuées
***
ReDCAD

Résumé

Nos travaux de recherche abordent la problématique de l'auto-réparation dans les applications orientées service et plus spécifiquement celles basées sur la technologie des services web. Nous nous intéressons à un contexte impliquant des applications logicielles construites par composition de services web distribués et multipropriétaires. Notre but est de garantir la fiabilité et la qualité de service (QdS) de ces applications à travers des techniques d'auto-réparation. Si un service web (SW) montre des perturbations continues de la qualité offerte pour un problème de traitement ou de communication, ceci est considéré comme une dégradation. Il est alors nécessaire de remédier à une telle dégradation en substituant le service défaillant par un ou plusieurs autres services réalisant des fonctions équivalentes. Notre recherche vise à concevoir une architecture qui permet le contrôle, l'analyse des données de la QdS et la reconfiguration des applications à base de SW en cours d'exécution. Aucune hypothèse sur la logique interne des services n'est nécessaire pour l'applicabilité de notre approche. Celle-ci repose sur des moniteurs capables d'étendre les messages SOAP, échangés entre le client et le fournisseur du service, et des connecteurs capables de rediriger les requêtes à destination d'un service défaillant vers un autre, supposé efficace, implantant la même logique métier. Dans ce cadre, l'objectif principal est de fournir des mécanismes non intrusifs d'observation et de reconfiguration afin d'éviter le dysfonctionnement qui surviendrait et qui deviendrait perceptible par les clients. Nous définissons pour cela un cadriciel et des services logiciels couvrant toute la boucle de la gestion d'auto-réparation allant du monitoring de la QdS jusqu'aux actions de reconfiguration. Nous retrouvons, par conséquent, les quatre modules principaux suivants. En premier lieu, nous avons le monitoring qui correspond à la supervision de l'application. Il observe et stocke des valeurs des paramètres de QdS. En deuxième lieu, vient l'analyse. Il s'agit de la phase d'exploitation des valeurs obtenues par le monitoring permettant de s'assurer du bon fonctionnement de l'application et de prédire et détecter une éventuelle dégradation de la QdS. Cette détection utilise un algorithme basé sur des fonctions statistiques et des contraintes temporelles et produit, le cas échéant, des alarmes qui vont enclencher le diagnostic. Nous nous intéressons à la surveillance de l'évolution d'une caractéristique donnée de QdS plus qu'à ses valeurs absolues. En troisième lieu, nous trouvons le diagnostic. L'objectif de ce module est l'identification de l'origine de cette dégradation et l'élimination de l'effet de sa propagation. En quatrième lieu, intervient la réparation

qui correspond à la reconfiguration de l'application pour rétablir la QdS. L'exécution des actions de reconfiguration est réalisée à travers des *Connecteurs de Liaison Dynamique* qui seront générés, compilés et déployés automatiquement. Nous avons élaboré deux prototypes implantant ces différents modules sous forme de service web. Le premier prototype considère l'application de revue coopérative, qui est une coopération de services web gérant les conférences scientifiques. Le deuxième prototype représente un cas d'étude pour le commerce électronique. Il s'agit d'une application à base de services web orchestrés.

Dédicace

A tous ceux qui comptent pour moi...

Riadh BEN HALIMA

Remerciement

C'est avec un grand plaisir que je réserve cette page en signe de gratitude et de profonde reconnaissance à tous ceux qui ont bien voulu apporter l'assistance nécessaire au bon déroulement de ce travail.
Je veux d'abord remercier mes encadreurs Monsieur Khalil DRIRA et Monsieur Mohamed JMAIEL qui n'ont épargné aucun effort dans l'encadrement de cette thèse afin de me permettre de défier les entraves rencontrées et de travailler avec volonté. Ils ont été toujours disponibles pour m'orienter à prendre les bonnes décisions. J'espère être à la hauteur de leur confiance. Qu'ils trouvent dans ce travail le fruit de leurs efforts et l'expression de ma profonde gratitude.
Je tiens à exprimer mon profond respect et mes vifs remerciements envers les membres de ce jury : Madame Michelle SIBILLA, pour l'honneur qu'elle m'a fait en acceptant de le présider, Madame Laurence DUCHIEN et Monsieur Ahmed HADJ KACEM pour être rapporteurs et Madame Louise TRAVÉ-MASSUYÈS et Monsieur Christophe CHASSOT pour être examinateurs de mon travail.
J'adresse également, un remerciement ensoleillé à Monsieur Mohamed Karim GUENNOUN pour sa bienveillance, sa fraternité, sa gentillesse. Son soutien constant m'a encouragé durant l'élaboration de cette thèse. J'apprécie ses grandes qualités morales et son extrême modestie. Qu'il trouve dans ce travail l'expression de mon profond respect et mon infinie reconnaissance.
Je remercie aussi tous les membres du laboratoire LAAS-Toulouse, ainsi que les membres de l'unité de recherche REDCAD-Sfax pour l'ambiance chaleureuse qui règne au sein des deux équipes et pour leur bonne humeur.
Mes remerciements vont aussi à Monsieur René PEGORARO, à Monsieur German SANCHO, à Monsieur Ismail BOUASSIDA, à Madame Emna FKI, à Monsieur Mourid MARRAKCHI, et à Monsieur Ahmed JMAL qui n'ont jamais hésité à m'offrir leurs aides, leurs coopérations et qui ont su insuffler à mon travail un élan qui m'était indispensable.

Table des matières

1 Chapitre 1 : État de l'Art et Problématique **9**

1.1 Services web : historique, concept et technique 9
1.1.1 Introduction 9
1.1.2 Objet, composant et service 10
1.1.3 L'Architecture Orientée Service (AOS) 11
1.1.4 Services web : Définition et infrastructure 12
1.1.4.1 Transport : SOAP 13
1.1.4.2 Découverte : UDDI 14
1.1.4.3 Description : WSDL 15
1.1.4.4 Invocation d'un service web 16
1.1.4.5 Apports des services web 18
1.1.5 Composition des services web 18
1.1.5.1 BPEL 19
1.1.5.2 WS-CDL 19
1.1.5.3 BPML 19
1.1.5.4 BPSS 20
1.1.5.5 WSCL 21
1.2 La Qualité de Service dans les services web 21
1.2.1 Introduction 21
1.2.1.1 Modèles de QdS existants 22
1.2.1.2 QdS spécifiques 23
1.2.1.3 QdS génériques 24
1.2.2 QdS considérées 24
1.2.2.1 Performance des services web 25
1.2.2.2 Monitoring de QdS de point de vue client 27
1.2.2.3 Collection des mesures des côtés client et fournisseur 28
1.2.3 Techniques de mesure des QdS 28
1.2.3.1 Le *Timer* inséré dans le code 28

1.2.3.2 Utilisation de l'approche orientée aspect 30
1.2.3.3 Modification de la bibliothèque du SOAP 30
1.2.3.4 Approche du proxy 31
1.2.3.5 Approche basée sur le monitoring de paquets 31
1.2.3.6 Environnements d'expérimentation 32
1.2.3.7 Synthèse 33
1.3 L'auto-réparation 35
1.3.1 Approches existantes d'auto-réparation 35
1.3.1.1 Les approches basées Modèle 36
1.3.1.2 Les approches basées Middleware 38
1.3.1.3 Les approches basées Plate-forme 42
1.3.2 Approches existantes de monitoring 46
1.3.3 Approches existantes de substitution 48
1.4 La problématique 50
1.4.1 Système d'auto-réparation : Externe vs Interne 50
1.4.2 Niveau de gestion de la QdS : Instance vs Classe 50
1.4.3 Service cible par la gestion de la QdS : Sans-état vs Avec-état 51
1.4.4 Gestion du monitoring et du diagnostic : Local vs Global . . . 52
1.4.5 Gestion de la QdS : Pronostic vs Diagnostic 52
1.4.6 Gestion de la QdS : Orchestration vs Chorégraphie 53
1.5 Conclusion 54

2 Chapitre 2 : Un Middleware Auto-Réparable guidé par la QdS (*MARQ*) 55
2.1 Introduction 55
2.2 Approche Générale 56
2.3 Le Monitoring 57
2.3.1 Algorithmes et modèles sous-jacents 57
2.3.1.1 Le monitoring local des communications synchrones . 58
2.3.1.2 Le monitoring global des communications asynchrones 59
2.3.2 Fonctionnalités et architecture de mise en œuvre 60
2.3.2.1 L'interception niveau SOAP 61
2.3.2.2 L'interception niveau HTTP 62
2.4 L'Analyse 63
2.4.1 Algorithmes et modèles sous-jacents 63
2.4.2 Fonctionnalités et architecture de mise en œuvre 67
2.5 Le Diagnostic/Pronostic et la Décision 68
2.5.1 Algorithmes et modèles sous-jacents 68
2.5.1.1 Détection de la propagation de dégradation 68

2.5.1.2 Prévention de dégradation : Le modèle Markovien . 72
2.5.2 Fonctionnalités et architecture de mise en œuvre 76
2.6 Conclusion 76

3 Chapitre 3 : La Gestion de la Reconfiguration pour l'Auto-Réparation 77
3.1 Introduction 77
3.2 La Reconfiguration 78
3.2.1 Algorithmes et modèles sous-jacents 78
3.2.1.1 L'algorithme de substitution d'un service 78
3.2.1.2 L'algorithme de substitution d'une opération 79
3.2.2 La formalisation de la substitution 82
3.2.2.1 La formalisation de la substitution d'un service . . . 83
3.2.2.2 La formalisation de la substitution d'une opération . 85
3.2.3 Fonctionnalités et architecture de mise en œuvre 87
3.2.3.1 La reconfiguration par *Connecteur de Liaison Dynamique* 87
3.2.3.2 La reconfiguration par routage adaptatif au niveau HTTP 90
3.2.3.3 Les niveaux de gestion de la QdS : SOAP vs. HTTP 91
3.3 Le protocole d'échange entre les composants de *MARQ* 91
3.4 Le déploiement des composants de *MARQ* 94
3.5 La recherche et la sélection de service de substitution 95
3.6 La conception de *MARQ* 96
3.7 Règles de transformation en architecture auto-réparable en bus 103
3.8 Intégration de l'auto-réparation de niveau classe et de niveau instance 105
3.8.1 Intégration passive 105
3.8.1.1 La phase transitoire gérée par *SH-BPEL* (I1) 105
3.8.1.2 La phase transitoire gérée par *MARQ* (I2) 106
3.8.2 Intégration active (I3) 107
3.8.3 Illustration 108
3.9 Conclusion 109

4 Chapitre 4 : Expérimentations et Applications 111
4.1 Introduction 111
4.2 La revue coopérative : Un processus de coordination distribuée 112
4.2.1 Description du scénario « Recherche de conférences » 113
4.2.2 Hypothèses d'implantation 114
4.2.3 Environnement de développement 115
4.2.3.1 Les grilles de calcul 115
4.2.3.2 La plate-forme *Grid'5000* 116

4.2.3.3 Configuration et préparatifs de l'expérimentation . . 116
4.2.3.4 Résultats et analyses 119
4.3 Le *FoodShop* : Un exemple de chaîne de commandes automatisées . . 123
4.3.1 Hypothèses d'implantation 124
4.3.2 Implantation 125
4.3.3 La réparation 128
4.3.3.1 Le connecteur de reconfiguration niveau HTTP . . . 128
4.3.3.2 Le connecteur de reconfiguration niveau SOAP . . . 129
4.4 Conclusion 131

Publications 137

Table des figures

1.1 Le formatage visuel d'un message SOAP 14
1.2 Les étapes d'invocation d'un service web [99] 17
1.3 La classification des attributs de QdS [33] 22
1.4 La liste des QdS mesurées 25
1.5 Les différents niveaux d'interception 33

2.1 L'architecture du Middleware Auto-Réparable guidé par la QdS . . . 56
2.2 Le mode de fonctionnement de l'interception niveau SOAP 62
2.3 Le mode de fonctionnement de l'interception niveau HTTP 62
2.4 Un modèle de chronique de détection de dégradation de QdS 65
2.5 La propagation de la dégradation de la QdS 70
2.6 La matrice A de transition 73
2.7 Le degré d'appartenance de la logique floue 74
2.8 L'estimation du prochain état du système avec les CMC 75

3.1 Les étapes principales du processus de génération automatique du *CLD* 88
3.2 Les opérations du *Connecteur de Liaison Dynamique* 90
3.3 Le flux de messages échangés entre les services d'auto-réparation . . . 92
3.4 Des exemples de messages SOAP 94
3.5 Diagramme de composants : L'architecture d'auto-réparation 96
3.6 Diagramme de composants : Le monitoring impliquant deux clients . 97
3.7 Diagramme de Séquences : Les actions de monitoring 97
3.8 Diagramme d'activités : Les activités de monitoring 98
3.9 Diagramme de composants : Le Diagnostic/Pronostic et Décision . . . 98
3.10 Diagramme de Séquences : Les interactions entre les modules de *MARQ* 99
3.11 Diagramme de composants : L'Exécuteur de Reconfiguration 100
3.12 Diagramme de composants : La génération de connecteur 101
3.13 Diagramme de Séquences : Le processus d'auto-réparation avec *MARQ* 102
3.14 Un bus par couple client/service 103

3.15 Un bus par un service et ses clients 104
3.16 Un bus pour tous les services de l'application 104
3.17 La phase transitoire gérée par le *SH-BPEL* 106
3.18 La phase transitoire gérée par *MARQ* 107
3.19 L'intégration active entre *MARQ* et *SH-BPEL* 107
3.20 L'intégration de *MARQ* et *SH-BPEL* illustrée par le *FoodShop* 108

4.1 Le Système de Revue Coopérative avec les composants d'auto-réparation112
4.2 Diagramme de séquences : Recherche de Conférences 114
4.3 L'infrastructure de l'expérimentation 117
4.4 La variation des paramètres de QdS 120
4.5 La surcharge des moniteurs 122
4.6 L'instanciation de *MARQ* avec l'application du *FoodShop* 123
4.7 Le visualisateur graphique appliqué au *FoodShop* 126
4.8 L'historique des conversations 126
4.9 Les opérations statistiques 127
4.10 Le reroutage des requêtes 129
4.11 Le prototype de *FoodShop* 130
4.12 Le recouvrement de dégradation 131

Liste des tableaux

1.1 Le modèle de QdS des services web proposé par Araban et Sterling [5] 23
1.2 Le *Timer* de monitoring . . . 29
1.3 La synthèse des approches de mesure . . . 34
1.4 La comparaison des approches d'auto-réparation basées middleware . 41
1.5 La comparaison des approches d'auto-réparation basées plate-forme . 44

2.1 Le comportement du *MCF* envers les communications synchrones . . 58
2.2 Le comportement du *MCC* envers les communications synchrones . . 59
2.3 Le comportement du *MCC* envers les communications asynchrones . 59
2.4 Le comportement du *MCF* envers les communications asynchrones . 60
2.5 L'algorithme de détection (Moyenne pré-calculée). . . . 66
2.6 L'algorithme de détection (Moyenne calculée au cours de l'exécution). 67
2.7 L'algorithme de localisation de dégradation . . . 69

3.1 L'algorithme de reconfiguration d'un service (1/3) . . . 78
3.2 L'algorithme de reconfiguration d'un service (2/3) . . . 79
3.3 L'algorithme de reconfiguration d'un service (3/3) . . . 80
3.4 L'algorithme de reconfiguration d'une opération (1/3) . . . 81
3.5 L'algorithme de reconfiguration d'une opération (2/3) . . . 81
3.6 L'algorithme de reconfiguration d'une opération (3/3) . . . 81
3.7 Un exemple de grammaire de graphes . . . 83
3.8 GG1 : La substitution simple d'un service . . . 84
3.9 GG2 : La substitution composée sans surcharge . . . 84
3.10 GG3 : La substitution composée selon la disponibilité . . . 85
3.11 GG4 : La substitution composée avec balance de charge . . . 86
3.12 GG5 : La substitution simple d'une opération . . . 86
3.13 GG6 : La substitution selon la disponibilité de l'opération . . . 86
3.14 GG7 : La substitution avec balance de charge entre les opérations . . 87
3.15 Les composants architecturaux de la génération et du déploiement du *CLD* 89

3.16 La description des messages échangés entre les services d'auto-réparation 92
3.17 Les contraintes de déploiement des composants de gestion de la QdS 95

4.1 La configuration des nœuds de la plate-forme *Grid'5000* 117
4.2 Tableau récapitulatif des valeurs issues de l'expérimentation. 120
4.3 La scalabilité du service web *ConfSearch* 121

As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now that we have gigantic computers, programming has become a gigantic problem.

Edsger W. Dijkstra,
ACM Turing Award Lecture, 1972

Introduction Générale

Les travaux présentés dans cette thèse se situent dans le contexte des applications distribuées à base de services web. Un service web est une entité logicielle permettant la communication et l'échange de données entre applications et systèmes hétérogènes dans des environnements distribués. Il vise à assurer l'interopérabilité, et ce à travers une présentation standardisée des services offerts et d'un protocole standard de communication permettant de structurer les messages échangés entre les composants logiciels. Il offre aussi une spécification de publication et de localisation de services. La particularité des services web réside dans le fait qu'ils utilisent la technologie Internet comme infrastructure pour la communication entre les composants logiciels. Les architectures orientées service constituent un paradigme de conception et de réalisation applicable aux différents niveaux d'interaction d'un système communicant. Elles sont amenées à jouer un rôle de plus en plus important dans la conception des futurs systèmes de communication et leurs applications logicielles.

Les travaux élaborés dans le cadre de cette thèse visent à soutenir l'exécution des services web auto-réparables, guidée par la Qualité de Service (QdS). La problématique traitée concerne l'observation et l'analyse de la QdS, et l'implantation de l'auto-réparation au cours de l'exécution des applications coopératives distribuées. Cette problématique constitue un enjeu et un défi scientifique important dans la mesure où la réparation automatique d'une application est devenue impérative dans un contexte d'autonomie de systèmes fortement distribués et contenant un large nombre de services [53]. Dans ce contexte, notre objectif est d'offrir un cadre logiciel de plus en plus autonome et qui s'adapte aux différents profils et besoins des clients. Ces besoins peuvent être exprimés dans des contrats de niveau de service (*Service Level Agreement, SLA*) spécifiques tels que avec *WSLA* [52] et *WSOL* [97]. Cependant, l'existence des SLAs prédéterminés n'est pas applicable pour toutes les situations, comme dans le cas de coopération libre ou informelle entre les services web. Dans ce cas, notre approche analyse les valeurs observées des paramètres de

QdS, comme le temps d'exécution et le temps de réponse, et les compare au cours de l'exécution avec les valeurs obtenues des observations passées. Les actions de reconfiguration sont mises en œuvre à travers des entités intermédiaires entre les clients et les fournisseurs des services web, telles que les *médiateurs* dans [100], et les *communautés* dans [91]. Dans ce cas, l'ajout d'un service de substitution à la liste des services équivalents, nécessite une intervention minimale. Cependant, notre approche prend la description WSDL du service substituant comme paramètre d'entrée et établit automatiquement la liaison avec ce dernier grâce à un *Connecteur de Liaison Dynamique.*

La QdS est décrite par les paramètres standards représentant les temps de communication et de traitement, ainsi que d'autres paramètres plus spécifiques tels que la disponibilité et la « scalabilité ». L'observation de la QdS nécessite le monitoring et l'analyse des messages SOAP échangés. Nous l'avons réalisé au niveau communication de façon générique indépendamment de la logique métier des services web observés, et sans hypothèse de modification du code de l'appelant et de l'appelé. Ceci est mis en œuvre à travers le marquage et l'extension des messages échangés par des métadonnées décrivant la QdS. Les services web communiquent par échange de messages suivant deux types d'interaction, à savoir les interactions synchrones (requête-réponse) et les interactions asynchrones (requête). Dans le premier mode de communication, l'appelant est bloqué jusqu'au retour du résultat de sa requête. En revanche, dans le deuxième mode de communication, les requêtes et leurs réponses (sous forme de requêtes) sont échangées de façon symétrique sans blocage de l'appelant. Dans ce cas, le calcul de la QdS exploite des données de corrélation, *MessageId* et *RelatedTo*, dans les entêtes des messages SOAP indiquant l'association entres les deux messages de type requête. Des moniteurs sont mis en pratique de deux façons différentes. Le premier type de moniteur (niveau SOAP) est implanté en exploitant et étendant la technique d'interception offerte par les conteneurs de déploiement des services web. Ces moniteurs se restreignent à la gestion des communications synchrones vu qu'ils ne retiennent pas les données de corrélation au niveau du connecteur de reconfiguration. En résultat, les données recueillies à partir des moniteurs SOAP limitent le processus d'auto-réparation au niveau des gestions locales du monitoring et du diagnostic. Pour pallier ces défauts, nous avons élaboré un deuxième type de moniteur niveau HTTP. Ce moniteur ne manipule pas le contenu du message SOAP mais uniquement l'enveloppe HTTP englobante. Ceci permet la gestion des communications asynchrones en plus des communications synchrones. En utilisant d'autres données de corrélation telles que *source* qui indique le service source de l'invocation, les données de monitoring nous permettent de construire la

structure d'interaction (profil) entre les différents services web impliqués dans l'application. Ce profil enrichit nos connaissances sur les indépendances structurelles de l'application et permet, par la suite, un monitoring global des QdS.

L'étape suivante est la détection des dégradations de la QdS. Elle repose sur les données collectées de l'étape de monitoring. Deux approches ont été expérimentées. La première approche est discrète. Elle utilise les chroniques et les valeurs statistiques pour la détection de dégradation en se focalisant sur l'évolution du comportement du service tout en évitant la considération des violations transitoires. Une chronique définit un ensemble d'événements liés par des contraintes temporelles. Un événement (nommé aussi violation), correspond à la détection d'une valeur de QdS mesurée qui dépasse un seuil calculé statistiquement en fonction des valeurs de QdS déjà stockées. La succession non contrôlée de ces violations entraîne l'évolution du système vers l'état de dégradation. Le déclenchement d'alarmes par la chronique, signale la présence de violations. La deuxième approche est analytique. Elle se base sur les *chaînes de Markov cachées* pour prévenir des dégradations imminentes. En plus de ces deux approches, le diagnostic combine les QdS pour identifier l'origine de la dégradation et élimine le phénomène de propagation de dégradation qui peut conduire à des actions de reconfiguration inutiles.

Suite à un déclenchement d'alarmes, une opération de reconfiguration intervient par des actions de substitution élémentaires afin de permettre d'établir une meilleure QdS. Un cadre conceptuel de bus de service dynamiquement reconfigurable a été défini. Nous l'appelons dans la suite : **M**iddleware **A**uto-**R**éparable guidé par la **Q**dS (*MARQ*). Il intègre des *Connecteurs de Liaison Dynamiques* comme éléments clés pour le routage des requêtes vers différents fournisseurs de services web. Différents niveaux de reconfiguration sont possibles (opération, service, groupe de services). Le middleware *MARQ* gère la réification entre plusieurs services web –sans états– qui implantent la même logique métier. Deux techniques ont été élaborées. La première s'effectue au niveau SOAP. Son déploiement est adaptable aux différentes contraintes d'accessibilité et de ressources du côté client ou du côté fournisseur du service web. Cette technique a été illustrée avec une application choréographée à travers des expérimentations à grande échelle sur la plate-forme *Grid'5000*, et une application orchestrée avec BPEL. La deuxième technique agit au niveau HTTP. Elle est implantée comme un proxy HTTP qui intègre des composants de modification des adresses de routage. Cette technique a été illustrée avec une orchestration de services web. Elle a permis de raffiner la reconfiguration au niveau des opérations offertes par les services web.

La première expérimentation menée sur la plate-forme *Grid'5000*, a prouvé que la charge de nos moniteurs reste quasiment nulle ($\simeq 0s$) tant que le nombre des clients simultanés ne dépasse pas 50, et raisonnable jusqu'à 500 clients ($\leq 0,5s$). La deuxième expérimentation a validé avec succès le monitoring des communications synchrones et asynchrones, le processus de détection de dégradation ainsi que la mise en œuvre des connecteurs de reconfiguration.

Le rapport se compose de quatre chapitres :

Dans le premier chapitre, nous commençons par introduire les notions de service web et l'architecture orientée service. Ensuite, nous passons en revue les modèles de QdS existants, en présentant ceux adoptés par notre travail ainsi que l'ensemble des techniques utilisées dans la littérature pour les mesurer. Par la suite, une synthèse des travaux portant sur l'auto-réparation est présentée. Nous avons classifié les approches existantes en trois principales catégories, à savoir celles orientées modèle, middleware et plate-forme. La dernière partie aborde les problématiques confrontées tout au long de ce travail, telles que les propriétés des systèmes d'auto-réparation (interne vs externe), les services web ciblés par la reconfiguration (sans état vs avec état), le niveau de gestion de la QdS (local vs global), etc.

Nous introduisons, dans le deuxième chapitre, notre middleware *MARQ* sous forme d'un middleware auto-réparable guidé par la QdS pour les services web. Nous présentons les fonctionnalités offertes par le cycle d'auto-réparation, partant du monitoring jusqu'aux actions de reconfiguration. Nous commençons par détailler les algorithmes et les fonctionnalités offertes pour la gestion locale et globale du monitoring. Une comparaison entre les deux niveaux de mise en œuvre des moniteurs (SOAP et HTTP) est réalisée. Par la suite, nous expliquons les modèles de détection utilisés. La partie suivante traite la manière dont le diagnostic intervient pour identifier la source de la dégradation et éliminer l'effet la propagation de dégradation. La dernière partie est consacrée au pronostic à travers les *chaînes de Markov cachées*.

Le troisième chapitre présente l'architecture en bus reconfigurable implantée par le middleware *MARQ*. Il met l'accent sur l'étape de réparation. Dans ce chapitre, la reconfiguration est présentée sous forme d'algorithmes et puis formalisée à travers les modèles de graphes d'architecture. Le raffinement d'une action de reconfiguration varie de la substitution totale du service web à la substitution d'une ou plusieurs de ses opérations. Ensuite, nous présentons le protocole d'échange entre les composants de *MARQ* ainsi que sa conception réalisée en UML. La dernière partie considère l'intégration de l'auto-réparation de niveau classe (représentée par *MARQ*), et celle

de niveau instance (représentée par *SH-BPEL*).

La validation des résultats d'expérimentation fera l'objet du quatrième chapitre. Nous avons illustré notre approche par deux applications orientées service : un système de revue coopérative, implanté par une collection de services choréographés, et une application de commerce électronique (*FoodShop*) implantée par une orchestration de services. Une expérimentation à large échelle sur la plate-forme *Grid'5000* est menée pour mesurer la performance et vérifier la charge de nos moniteurs. Les détails de la mise en œuvre des expérimentations sur l'environnement *Grid'5000* sont présentés et les résultats sont analysés. Avec l'application du *FoodShop*, nous avons développé une interface de visualisation graphique de la gestion de la QdS. Ce module permet aussi de reconstruire les interactions entre les services web impliqués dans l'application et de pronostiquer l'état de toutes les opérations des services utilisés.

1

Chapitre 1 : État de l'Art et Problématique

1.1 Services web : historique, concept et technique

1.1.1 Introduction

Les applications logicielles d'entreprises deviennent de plus en plus distribuées, complexes et coûteuses en efforts de gestion. Parallèlement, l'espérance de vie de ces applications ne cesse d'être remise en question par le dynamisme qui caractérise, de nos jours, le monde de l'entreprise. Pour remédier à ces insuffisances, il faut développer de nouveaux concepts technologiques contribuant au développement d'applications plus étendues et plus complexes sur un marché en mutation constante. C'est dans ce contexte que se situent les architectures distribuées, partant de celles à 2–niveaux, passant par celles à 3–niveaux et aboutissant à celles à N–niveaux, et dont les objectifs sont les suivants :

- La simplification du fonctionnement du système (i.e. offrir une vue homogène d'un monde hétérogène, définir et respecter des normes et des standards) ;

- La favorisation de la réutilisation des composants logiciels ;
- L'augmentation de l'exigence non-fonctionnelle ;
- La garantie de l'évolutivité fonctionnelle et technique du système (i.e. préserver une certaine indépendance du système vis-à-vis des évolutions techniques potentielles de chacun de ses éléments).

1.1.2 Objet, composant et service

L'évolution des langages de programmation a amené de nouveaux outils aidant à la conceptualisation des problèmes en informatique. En effet, l'avènement de l'orienté objet a facilité l'abstraction du problème à résoudre en fonction des données du problème lui même (par l'utilisation de classes et d'objets). L'apparition de la programmation orientée objet a aussi donné lieu à de nouvelles technologies de distribution des applications [105] telles que RMI (*Remote Method Invocation*) et CORBA (*Common Object Request Broker Architecture*). RMI, middleware de Sun[1], assure la portabilité de l'exécution grâce à la machine virtuelle Java. CORBA, architecture et norme d'OMG (*Object Management Group*), est indépendante des plate-formes grâce au protocole de communication IIOP (*Internet Inter-ORB Protocol*).

L'objectif principal des modèles objet est d'améliorer la modélisation d'une application, et d'optimiser la réutilisation du code produit. Cependant, l'intégration d'entités logicielles existantes peut s'avérer difficile si leur modèle d'exécution est incompatible avec le modèle imposé par le langage objet choisi pour le développement de nouvelles entités. Par ailleurs, les modèles et les langages objets ne sont pas, en général, adaptés à la description des schémas de coordination et de communication complexes [64].

Pour pallier les défauts de l'approche objet, l'approche composant est apparue. Cette approche est fondée sur des techniques et des langages de construction des applications qui intègrent, d'une manière homogène, des entités logicielles provenant de diverses sources. Un composant est une boite noire, communicant avec l'extérieur à travers une interface dédiée, permettant la gestion du déploiement, de la persistance, etc [67]. En plus du concept d'objet, le composant se caractérise par la notion de déploiement, qui gère son cycle de vie de l'installation à l'instanciation. Cette notion permet aux développeurs de se focaliser sur la logique métier du composant, et délègue la gestion des propriétés non-fonctionnelles à l'environnement d'exécution l'hébergeant. L'enjeu est de faciliter la production de logiciels fiables, maintenables,

[1] http ://java.sun.com

évolutifs et toujours plus complexes. L'un des maîtres mots est la *réutilisation* par la disponibilité d'ingrédients logiciels, facilement composables et adaptables. Les technologies EJB (*Enterprise JavaBeans*) et CCM (*Corba Component Model*) sont les technologies les plus connues de l'architecture de composants logiciels pour la plate-forme J2EE (*Java 2 Enterprise Edition*) [105].

La distribution de composants fait naître de nouvelles difficultés qu'il convient de gérer efficacement afin de préserver la souplesse et de garantir l'évolutivité du système d'information. D'une part, l'interdépendance de composants distribués diminue la maintenabilité et l'évolutivité du système. Ainsi, on perçoit que, pour préserver son efficacité, une architecture distribuée doit minimiser l'interdépendance entre chacun de ses composants qui risque de provoquer des dysfonctionnements en cascade dont il est souvent complexe de détecter la cause et de déterminer précisément l'origine. D'autre part, la préservation de la qualité de service du système d'information, dans le cadre d'architectures distribuées, est une lourde tâche, souvent complexe, en particulier lorsque les composants techniques de l'architecture sont hétérogènes et qu'ils exploitent de multiples produits et standards. Ces difficultés imposent la nécessité d'une architecture plus flexible où les composants sont réellement indépendants et autonomes, le tout permettant de déployer plus rapidement de nouvelles applications. D'où l'apparition de l'architecture orientée service [67].

1.1.3 L'Architecture Orientée Service (AOS)

L'AOS est une approche architecturale permettant la création des systèmes basés sur une collection de services développés dans différents langages de programmation, hébergés sur différentes plate-formes avec divers modèles de sécurité et processus métier [49]. Chaque service représente une unité autonome de traitement et de gestion de données, communiquant avec son environnement à l'aide de messages. Les échanges de messages sont organisés sous forme de contrats d'échange. L'idée maîtresse de l'architecture orientée service est que tout élément du système d'information doit devenir un service identifiable, documenté, fiable, indépendant des autres services, accessible, et réalisant un ensemble de tâches parfaitement définies [15]. L'AOS est axée autour de trois concepts fondamentaux, à savoir, le fournisseur de services, le client de services, et l'annuaire de publication [48]. Le fournisseur permet l'accès à son service à travers une interface. Le client désigne une interface utilisateur, un serveur ou une autre application qui accède au service et l'invoque à travers son interface. L'annuaire joue le rôle d'intermédiaire entre le fournisseur et

le client. Les fournisseurs y enregistrent leurs services, et les clients y cherchent le service satisfaisant leurs besoins.

L'AOS présente plusieurs avantages bénéfiques pour le domaine de la technologie d'information et de communication. Elle se caractérise par la simplicité à travers les concepts de décomposition, de découplage et de réutilisation [102]. En plus, elle participe à la réduction du coût de développement des grands projets et rend le développement plus efficace.

L'architecture orientée service est apparue pour palier les limites des architectures distribuées. Cette architecture n'est pas simplement une mode. Elle se place, plutôt, dans la continuité logique des multiples tentatives de distribution de traitements, de répartition de données, d'intégration d'applications, d'homogénéisation du système d'information, etc. L'adoption de l'AOS a été énormément facilitée par l'émergence opportune de la technologie des services web et leurs standards bien définis.

La technologie des services web représente la technologie la plus utilisée pour migrer vers ce type d'architectures.

1.1.4 Services web : Définition et infrastructure

Selon Justin et al.[48] : *Un service web est une agrégation de fonctionnalités publiées pour être utilisées. Il utilise Internet comme conduit pour réaliser une tâche. Il est semblable à un processus métier virtuel qui définit des interactions au niveau application.*

Selon W3C[2] : *Un service web est un système logiciel conçu pour supporter les interactions entre applications à travers le réseau. Les services web offrent un moyen standard d'interopérabilité entre différentes applications qui s'exécutent sur une variété de machines/plate-formes. Ils sont caractérisés par leur grande interopérabilité et extensibilité, ainsi qu'une description interprétable/compréhensible automatiquement par la machine grâce au standard XML. Ils peuvent être combinés d'une façon faiblement couplée afin de réaliser des opérations complexes. Les programmes offrant des services simples, peuvent interagir ensemble afin de mettre en place des services sophistiqués avec des valeurs ajoutées.*

Les services web sont des compléments aux programmes et applications existants, développés dans différents langages de programmation, et servent de pont pour que ces

[2] World Wide Web Consortium, http ://www.w3c.org/

programmes communiquent entre eux [10]. Ainsi, les services web permettent d'interfacer des systèmes d'information hétérogènes. Ces derniers présentent les avantages suivants :

- Un faible couplage avec les technologies employées en interne ;
- Une grande flexibilité de mise à jour des systèmes employés de part et d'autre ;
- L'emploi de protocoles réseaux simples, répandus et bénéficiant d'implantations dans toutes les technologies majeures.

Les services offerts par l'infrastructure des services web couvrent essentiellement deux aspects fondamentaux [27] :

- Un service de communication qui permet l'échange de données entre les services web ;
- Un ensemble de services techniques destinés à automatiser le processus de localisation et d'invocation des composants.

L'originalité de l'infrastructure des services web consiste à les mettre en place en se basant exclusivement sur les protocoles les plus répandus d'Internet tels que HTTP (*HyperText Transfer Protocol*) et les formats standards d'échange de données tels que MIME (*Multipurpose Internet Mail Extensions*), XML, etc.

L'infrastructure des services web s'est concrétisée autour de trois spécifications considérées comme des standards, à savoir SOAP, UDDI et WSDL [20]. Nous les détaillons dans les parties suivantes.

1.1.4.1 Transport : SOAP

Simple Object Access Protocol appelé également *Service Oriented Access Protocol*[3]

SOAP est le protocole qui assure l'échange de messages dans les AOSs. Du fait qu'il est basé sur XML, il permet l'échange de données structurées indépendamment des langages de programmation ou des systèmes d'exploitation. SOAP permet l'échange d'informations dans un environnement décentralisé et distribué, comme Internet, indépendamment du contenu du message. Il peut être employé dans tous les styles de communication : synchrones ou asynchrones, point à point ou multi-points. SOAP utilise principalement les deux standards HTTP et XML :

- HTTP est un protocole de transport des messages SOAP. Il constitue, d'une part, un moyen efficace de transport et d'autre part, il est très utilisé sur le web.
- XML est un langage utilisé pour structurer les requêtes et les réponses et indiquer

[3]Soumis au W3C par UserLand, Ariba, Commerce One, Compaq, Developmentor, HP, IBM, IONA, Lotus, et Microsoft en May 2000

les paramètres des méthodes, les valeurs de retour, et les éventuelles erreurs de traitement.

Contrairement aux autres protocoles, IIOP pour CORBA, ORPC (*Object RPC*) pour DCOM, ou JRMP (*Java Remote Method Protocol*) pour RMI, qui sont des protocoles binaires [31], SOAP se base sur XML pour encoder les données. Les messages échangés via ce protocole jouissent donc des avantages que leur procure le langage XML pour structurer les données.

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  - <soapenv: Header/>
  - <soapenv:Body>
    - <Addition xmlns="http://doc">
          <a>4</a>
          <b>7</b>
      </Addition>
    </soapenv:Body>
  </soapenv:Envelope>
```

FIG. 1.1 – Le formatage visuel d'un message SOAP

La figure 1.1 illustre un exemple de message SOAP requête d'un service web qui additionne deux entiers. Le message est englobé dans une enveloppe et divisé en deux parties : l'entête et le corps. L'entête (*Header*) offre des mécanismes flexibles pour étendre un message SOAP sans aucune préalable connaissance des parties communicantes. Les extensions peuvent contenir des informations concernant l'authentification, la gestion des transactions, le payement, etc [41].

Le corps (*Body*) offre un mécanisme simple d'échange des informations mandataires destinées au receveur du message SOAP. Cette partie contient les paramètres fonctionnels tels que le nom de l'opération à invoquer, les paramètres d'entrée et de sortie ou des rapports d'erreur [41].

1.1.4.2 Découverte : UDDI

Universal Description, Discovery and Integration [4]

UDDI est une norme d'annuaire de services web appelée via le protocole SOAP.

[4]UDDI est le résultat d'un accord inter-industriel proposé par Dell, Fujitsu, HP, Hitachi, IBM, Intel, Microsoft, Oracle, SAP, Sun, etc. en 2001

Pour publier un nouveau service web, il faut générer un document appelé *Business Registry*. Il sera enregistré sur un UDDI *Business Registry Node*. Le *Business Registry* comprend trois parties [98] :

- Pages blanches : noms, adresses, contacts, identifiants des entreprises enregistrées ;
- Pages jaunes : informations permettant de classer les entreprises, notamment l'activité, la localisation, etc ;
- Pages vertes : informations techniques sur les services proposés.

Le protocole d'utilisation de l'UDDI contient trois fonctions de base :

- *publish* pour enregistrer un nouveau service ;
- *find* pour interroger l'annuaire ;
- *bind* pour effectuer la connexion entre l'application cliente et le service.

Comme pour la certification, il est possible de constituer des annuaires UDDI privés, dont l'usage sera limité à l'intérieur de l'entreprise.

1.1.4.3 Description : WSDL

Web Services Description Language [5]

WSDL, basé sur XML, permet de décrire le service web, en précisant les méthodes disponibles, les formats des messages d'entrée et de sortie, et comment y accéder.

L'élément racine d'une description WSDL est une *définition*. Chaque document définit un service comme une collection de points finaux ou *ports*. Chaque port est associé à un rattachement spécifique qui définit la manière avec laquelle les messages seront échangés. Chaque rattachement établit une correspondance entre un protocole et un *type de port*. Un *type de port* se compose d'une ou plusieurs opérations qui représentent une définition abstraite des capacités fonctionnelles du service. Chaque opération est définie en fonction des messages échangés au cours de son invocation. La structure du message est définie par des éléments XML associés à un schéma de type spécifique.

Ainsi, un document WSDL utilise les éléments suivants pour la définition des services [24] :

- *Types* : qui définissent des types de données échangées ;
- *Message* : qui définit d'une manière abstraite des données transmises ;
- *Operation* : qui décrit d'une manière abstraite les actions supportées par le service ;

[5] Proposé au W3C par Ariba, IBM et Microsoft en Mars 2001, la première version du standard a été proposé par le W3C en 2002

- *Port Type* (appelé *Interface* depuis WSDL2.0) : qui représente un ensemble d'opérations correspondant chacune à un message entrant ou sortant ;
- *Binding* (*Rattachement*) : qui est un protocole de communication et un format des données échangées pour un port ;
- *Port* : qui est une adresse d'accès au service ;
- *Service* : qui regroupe un ensemble de ports.

1.1.4.4 Invocation d'un service web

Le processus d'invocation d'un service web est similaire à celui de toute application distribuée utilisant la technologie CORBA ou RMI. Les étapes les plus importantes de l'invocation d'un service web sont les suivantes (voir figure 1.2) :

1. Le fournisseur de service se charge de l'enregistrement et de la publication des services auprès d'un serveur UDDI. Cette opération se fait par l'envoi d'un message (encapsulé dans une enveloppe SOAP) à l'annuaire UDDI. Ce message contient la localisation du service, la méthode d'invocation (et les paramètres associés) ainsi que le format de réponse. Toutes ces informations seront formalisées, par la suite, à l'aide de WSDL.

2. Un utilisateur désirant consulter un service interroge, en premier lieu, le serveur UDDI dont il possède l'adresse afin de connaître les services disponibles correspondant à ses besoins. Le serveur lui retourne la liste des possibilités parmi lesquelles il sélectionne une. A ce stade, l'utilisateur ne possède qu'une URL (*Uniform Resource Locator*) identifiant le service sélectionné.

3. L'utilisateur récupère ensuite une interface WSDL, accessible depuis l'URL, et qui lui permet de savoir comment utiliser le service. A partir de cette interface, il génère automatiquement le « proxy »du service. Il s'agit d'un objet local disposant des mêmes fonctions que le service distant et qui permettra à l'utilisateur d'accéder au service distant en toute transparence. Le « proxy »est créé grâce à un outil et peut être généré dans un grand nombre de langages de programmation différents. L'utilisation du service se fait simplement en invoquant la méthode du « proxy »qui correspond aux besoins de l'utilisateur.

4. Le proxy représente l'appel de la méthode distante sous forme d'une requête SOAP dans laquelle seront inclus les paramètres fournis par l'utilisateur. Ces paramètres seront empaquetés grâce à la méthode standard de présentation des données, ce qui permet d'assurer la compatibilité entre machines. Cette requête est, ensuite, émise

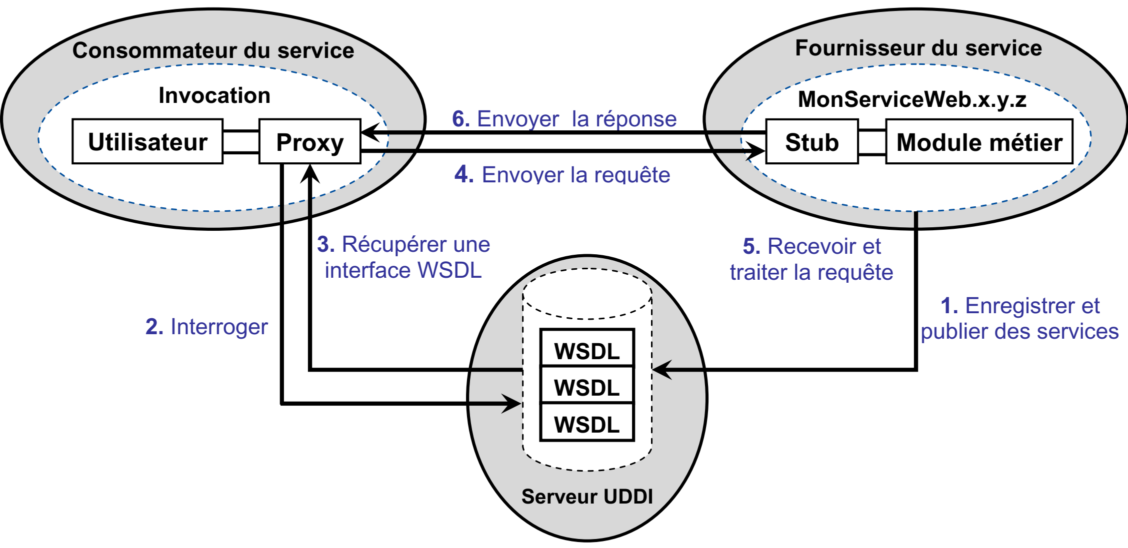

FIG. 1.2 – Les étapes d'invocation d'un service web [99]

vers l'URL désignant le service web.

5. Sur la machine hébergeant le service, la requête est reçue puis ouverte par la souche serveur (*stub*).

6. Une fois la requête est comprise, une réponse SOAP est formulée puis émise en direction de l'expéditeur initial.

Actuellement, SOAP, WSDL et UDDI sont les trois standards qui constituent l'architecture des services web. Il existe plusieurs environnements qui englobent ces trois standards et qui supportent, par la suite, l'implantation des services web. Parmi ces derniers, nous pouvons citer :

- *Axis* et le serveur *Tomcat* (http ://ws.apache.org/axis/) ;
- Serveurs HTTP *IIS* de Microsoft avec le framework .NET (http ://www.iis.net/) ;
- *Oracle WebLogic* de BEA et Oracle (http ://www.oracle.com/appserver/weblogic/weblogic-suite.html) ;
- *WebSphere* d'IBM (http ://www-01.ibm.com/software/websphere/) ;
- *JBoss* (http ://www.jboss.org/) ;
- *JAX-WS* (https ://jax-ws.dev.java.net/).

1.1.4.5 Apports des services web

L'utilisation des services web engendre plusieurs avantages dont nous pouvons citer [10] :

- L'interopérabilité : c'est la capacité des services web d'interagir avec d'autres composantes logicielles via des éléments XML en utilisant des protocoles de l'Internet.
- La simplicité : les services web réduisent la complexité des branchements entre les participants. Cela se fait en ne créant la fonctionnalité qu'une seule fois plutôt qu'en obligeant tous les fournisseurs à reproduire la même fonctionnalité à chacun des clients selon le protocol de communication supporté.
- Une composante logicielle légèrement couplée : l'architecture modulaire des services web, combinée au faible couplage des interfaces associées [9], permet l'utilisation et la réutilisation de services qui peuvent être recombinés facilement dans d'autres applications.
- L'hétérogénéité : les services web permettent d'ignorer l'hétérogénéité entre les différentes applications. En effet, ils décrivent la manière de transmettre un message (standardisé) entre deux applications, sans imposer la façon de le construire.
- L'auto-descriptivité : les services web ont la particularité d'être auto-descriptifs, c'est à dire, ils sont capables de fournir des informations permettant de comprendre la manière de les manipuler. La capacité des services à s'auto-décrire permet d'envisager l'automatisation de l'intégration des services.

1.1.5 Composition des services web

Généralement, un service web unique ne satisfait pas aux besoins des utilisateurs qui sont, de plus en plus, complexes. Pour fournir une solution à une tâche complexe, on peut combiner des services web pour n'en former qu'un seul ; on parle, alors, de composition de services web. La composition peut être élaborée soit d'une façon ad hoc, soit en utilisant un langage dédié. Pour la première, il s'agit d'un assemblage de plusieurs services web, dont les interactions sont codées manuellement par le développeur. Ce dernier prend en charge l'organisation du déroulement des processus métiers. Quant à la deuxième façon, elle est mise en œuvre à travers des langages tels que BPEL, WS-CDL, BPML, BPSS et WSCL.

1.1.5.1 BPEL

Business Process Execution Language[6]

Initialement connu sous le nom de *BPEL4WS*, renommé par la suite *WS-BPEL*, cette spécification est plus connue sous le nom de BPEL [4]. Ce dernier a succédé à XLANG (de Microsoft) et WSFL (d'IBM) comme un standard de spécification des flux entre les services web. Il s'agit d'un langage d'exécution des processus métiers qui permet la composition d'un ensemble de services web, et spécifie les règles de dialogue entre eux. Il définit un protocole d'interaction des services web (en se basant sur leurs WSDL) tout en spécifiant l'ordre d'invocation des opérations. Le process exécutable ressemble à une description d'un cadre de travail représentée par des activités basiques et structurelles spécifiant un modèle d'exécution des services web. Le document BPEL agit sur des éléments comme la transformation de données, l'envoi de messages ou l'appel d'opérations. Un processus BPEL peut être vu comme un service web autonome et son interface peut être représentée en utilisant WSDL.

BPEL est supporté par de nombreux éditeurs de logiciel comme Adobe, BEA Systems, HP, IBM, Oracle, JBoss, Sun, Tibco, Webmethods et Microsoft.

1.1.5.2 WS-CDL

Web Services Choreography Description Language[7]

Développé par le groupe de travail *Choreography* de W3C, WS-CDL [51] décrit le protocole métier de composition selon un point de vue globale. La description est implantée par un processus distribué individuel sans contrôle central [19]. Tout comme BPEL, il décrit les relations entre services composites mais, contrairement à BPEL, qui centralise le contrôle (*orchestration*), WS-Choreography s'intéresse à une description distribuée des messages échangés entre les partenaires.

1.1.5.3 BPML

Business Process Modeling Language[8]

[6]Proposé par BEA Systems, IBM, Microsoft, SAP et Siebel Systems en 2002, et standardisé par OASIS en 2007

[7] Proposé par Oracle, Commerce One, Novell, Choreology, W3C et Adobe Systems Incorporated en 2005

[8]Proposé par le consortium BPMI (Business Process Management Initiative) en 2001

C'est un méta-langage de modélisation [95] des processus métiers dont les premières spécifications sont apparues au printemps 2001 [8]. BPML fournit un modèle abstrait et une grammaire pour exprimer des processus métiers abstraits et exécutables. En utilisant BPML, les processus d'entreprise, les services web complexes et les collaborations multi-partenaires peuvent être définis et dirigés par des compositions d'activités qui exécutent des fonctions spécifiques. Un processus BPML peut être une partie d'une composition. Chaque activité dans le processus possède un contexte qui définit les comportements communs et les activités s'exécutant dans tel contexte. Ainsi, un processus peut être défini comme un type d'activité complexe qui déclare son propre contexte d'exécution. La spécification BPML définit dix-sept types d'activité et trois types de processus. La description WSDL d'un service web peut être importée ou référencée dans une spécification BPML. Une standardisation des documents BPML est proposée en utilisant RDF pour la sémantique des métadonnées, des métadonnées XHTML et *Dublin Core*[9] pour améliorer la lisibilité et le traitement de l'application.

1.1.5.4 BPSS

Business Process Specification Schema[10]

ebXML est un standard électronique basé sur XML qui permet aux entreprises de se retrouver et d'accomplir des affaires en utilisant des messages bien définis et des processus métiers standards [92]. Le schéma de spécification de processus métiers d'ebXML (BPSS) est une représentation des modèles de collaboration des processus métiers électroniques. En utilisant la syntaxe de XML, les parties impliquées dans une collaboration peuvent être modélisées et être d'accord sur le processus métier pertinent. Le standard BPSS d'ebXML peut être utilisé pour configurer les systèmes métiers afin de soutenir la collaboration commerciale. Ainsi, cette spécification détermine l'échange en cours (représenté par des modèles graphiques) des documents et des signaux métiers entre les partenaires. Une bibliothèque de modèles de processus peut être créée en utilisant les définitions de BPSS. Chaque modèle permet à l'utilisateur l'extraction des informations du BPSS correspondant et la configuration de son système au cours de l'exécution. Cependant, il n'y a pas un support explicite de description des flux de données durant les transactions. Mais il existe un support pour la spécification de la sémantique de la qualité de service pour les transactions

[9]Dublin Core est un modèle à base d'un ensemble de quinze propriétés utilisées pour la description des ressources

[10]C'est la spécification d'ebXML Business Process, standardisée par OASIS en 2001

telles que l'authentification et le dépassement du temps limite.

1.1.5.5 WSCL

Web Services Conversation Language[11]

WSCL [11] permet de définir le comportement externe visible des services web en spécifiant les conversations du niveau métier ainsi que les processus métiers publiques supportés par un service web. Les conversations sont définies en utilisant la syntaxe de XML. Un document WSCL spécifie les documents XML échangés comme une partie de la conversation ainsi que l'ordre dans lequel ils sont échangés. WSCL fournit un ensemble minimal de concepts nécessaires pour spécifier les conversations. La spécification déclare que la conversation est typiquement déterminée à partir de la perspective du fournisseur du service, qui peut aussi être utilisée pour déduire la conversation de la perspective du client. Bien que la conversation soit définie de la perspective du fournisseur du service, elle sépare la logique de la conversation de la logique de mise en œuvre ou des aspects d'implantation du service.

1.2 La Qualité de Service dans les services web

1.2.1 Introduction

Avec la prolifération des services web, la notion de QdS émerge aujourd'hui. Son intérêt pour les fournisseurs et les clients de service devient, de plus en plus, importante. Dans cette section, nous détaillons les paramètres de qualité de service pour les services web et présentons les différentes techniques de mesure existantes.

Il n'existe pas de consensus sur la définition de la qualité de service (QdS). La recommandation ITU-X.902[12] définit la QdS comme *un ensemble d'exigences dans le comportement collectif d'un ou plusieurs objets*. Dans le contexte des technologies de l'information et multimédia, la QdS a été définie par Vogel et al.[101] comme *l'ensemble des caractéristiques quantitatives et qualitatives d'un système multimédia, nécessaires pour atteindre la fonctionnalité requise par l'application*. Nous pouvons aussi dire que la qualité de service représente l'aptitude d'un service à répondre d'une

[11] Proposé par HP en 2002 et publié par W3C en 2005

[12] The International Telecommunication Union (ITU) standard X.902, Information technology - Open distributed processing - Reference Model.

manière adéquate à des exigences, exprimées ou implicites, qui visent à satisfaire ses usagers. Ces exigences peuvent être liées à plusieurs aspects d'un service, par exemple : sa disponibilité, sa fiabilité, etc.

Comme pour les exigences en QdS dans les services des couches basses des réseaux, il y a un besoin d'identifier et de décrire la QdS d'un service web. Les attributs de QdS peuvent être classifiés en deux parties : les QdS spécifiques, et les QdS génériques. Celles-ci peuvent être, également, divisées en des attributs mesurables et des attributs non mesurables. Cette classification est montrée par la figure 1.3.

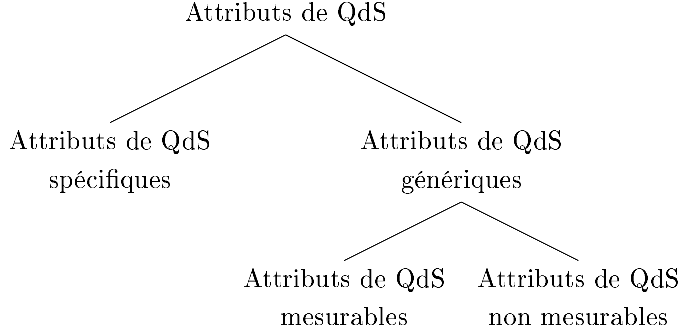

FIG. 1.3 – La classification des attributs de QdS [33]

1.2.1.1 Modèles de QdS existants

Le groupe de travail *Architecture des Services Web* du W3C, travaillant sur les architectures des services web, a identifié et décrit un ensemble de paramètres de QdS pour les services web [58], à savoir : la performance –qui englobe le débit (*throughput*), le temps de réponse et le temps d'exécution–, la fiabilité, la scalabilité ou l'adaptation au facteur d'échelle (*scalability*), la capacité, la robustesse, le traitement d'exception, l'exactitude, l'intégrité, l'accessibilité, la disponibilité, l'interopérabilité, la sécurité, et les exigences en QdS liées au réseau.

Il n'y a pas un consensus bien précis au sujet de l'ensemble des QdS importantes pour les services web, mais la plupart des travaux de recherche, qui ont essayé d'identifier et de classifier les paramètres de QdS, ont pris en considération les paramètres définis par le W3C auxquels sont associés, dans certains travaux, d'autres paramètres.

Le modèle de QdS pour les services web, proposé dans [5], suggère une classification principale des attributs de QdS, basée sur les attributs indépendants de l'environnement du service (partie fonctionnelle) et les attributs dépendants de l'environnement

du service (partie non fonctionnelle).

Facteurs QdS	Attributs internes (Métriques)	Attributs externes (Métriques)
Fiabilité	Correction (Exactitude, Précision)	(Disponibilité, Consistance)
Performance	Efficacité (Complexité temporelle et spatiale)	Gestion de la charge (Débit, Attente & Temps de réponse)
Intégrité		Sécurité
Utilisation	(paramètres d'entrée et de sortie)	

Tab. 1.1 – Le modèle de QdS des services web proposé par Araban et Sterling [5]

Bien que les métriques détaillées dans le tableau 1.1 soient moins bien définies que le modèle détaillé dans l'approche [58], le modèle de [5] donne une orientation générale que certains des attributs de QdS doivent être mesurés en examinant l'implantation du service (c-à-d les attributs internes).

Le travail mené par [82] a identifié et organisé les attributs de QdS des services web en catégories :

- Attributs liés à l'exécution (*Runtime Related QoS*) : scalabilité, capacité, performance (temps de réponse, temps de latence, et débit), fiabilité, disponibilité, robustesse/flexibilité, traitement d'exception, et exactitude ;
- Attributs liés au support de transaction : intégrité de transaction ;
- Attributs liés au prix et à la gestion de configuration : standard supporté, stabilité, prix et complétude ;
- Attributs liés à la sécurité : authentification, autorisation, confidentialité, traçabilité, cryptage de données, et non-répudiation.

1.2.1.2 QdS spécifiques

Les QdS spécifiques sont des qualités qui concernent une application particulière, et qui sont en relation avec sa logique métier. Par exemple, pour le processus de revue coopérative dans les conférences scientifiques (voir chapitre 4), nous pouvons identifier une liste de dysfonctionnements qui pourraient dégrader la QdS. En effet, il s'agit des événements à mesurer afin de diagnostiquer des problèmes éventuels

du système. Ces QdS spécifiques peuvent être des QdS liées aux arguments comme « le renvoi de conférence dont la date limite de soumission est dépassée suite à une requête de recherche de conférences par mot clé », ou « le renvoi de conférence dont le thème n'est pas pertinent ».

1.2.1.3 QdS génériques

Dans ce qui suit, nous présentons l'ensemble des attributs de QdS génériques. Nous distinguons, ici, les attributs mesurables et les attributs non mesurables.

Attributs mesurables :

Les attributs mesurables les plus communs sont décrits pas les paramètres liés à la performance.
- Le débit : le nombre de requêtes servies pendant un intervalle de temps ;
- Le temps de réponse : le temps requis pour compléter une requête du service web ;
- La fiabilité : la capacité d'un service d'exécuter correctement ses fonctions ;
- La scalabilité : la capacité du service de traiter le plus grand nombre d'opérations ou de transactions pendant une période donnée, tout en gardant les mêmes performances ;
- La robustesse : la probabilité qu'un service puisse réagir proprement à des messages d'entrée invalides, incomplets ou conflictuels ;
- La disponibilité : la probabilité d'accessibilité d'un service.

Attributs non mesurables :

Il y a des attributs de QdS qui ne sont pas mesurables, mais qui ont de l'importance pour les services web comme :
- Le prix d'exécution : c'est le prix qu'un client du service doit payer pour bénéficier du service ;
- La réputation : c'est une mesure de la crédibilité du service qui dépend, principalement, des expériences d'utilisateurs finaux ;
- La sécurité : c'est un regroupement d'un ensemble de qualités à savoir : la confidentialité, le cryptage des messages et le contrôle d'accès.

1.2.2 QdS considérées

Dans ce qui suit nous nous intéressons de près aux attributs de QdS que nous gérons dans notre travail.

1.2.2.1 Performance des services web

La performance des services web n'est pas un concept formellement défini. Elle est quantifiée à l'aide de différentes métriques. Nous allons adopter la définition fournie par le groupe travaillant sur les architectures des services web du W3C comme une fondation pour notre propre définition. Cette définition est composée du débit, du temps de réponse, du temps de communication et du temps d'exécution. Le temps d'exécution et le temps de communication sont deux dérivés de la définition du temps de réponse du W3C. Dans le cadre de cette thèse, nous considérons les paramètres inclus dans cette définition. En plus, nous tenons compte de la disponibilité et de la scalabilité.

Pour assurer le monitoring des paramètres de QdS considérés dans notre étude, quatre valeurs de temps sont mesurées, comme le montre la figure 1.4 :

t1 : Le temps d'envoi de la requête par le client ;

t2 : Le temps de réception de la requête par le fournisseur ;

t3 : Le temps d'envoi de la réponse par le fournisseur ;

t4 : Le temps de réception de la réponse par le client.

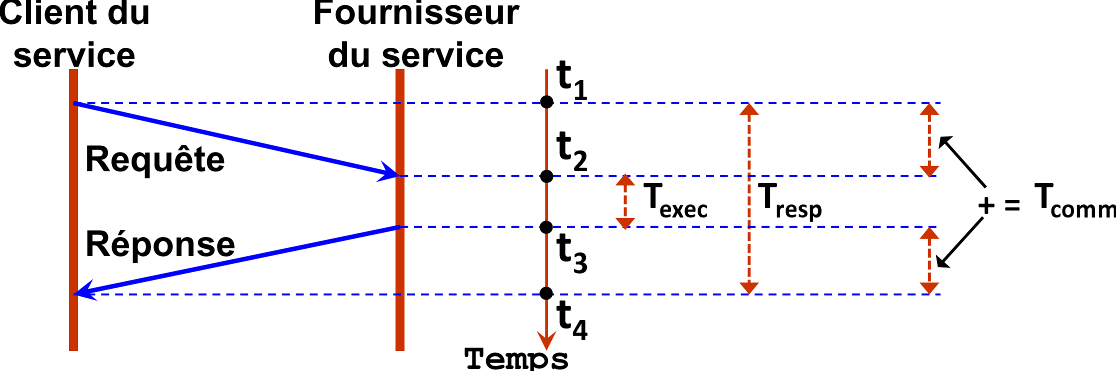

FIG. 1.4 – La liste des QdS mesurées

Dans la suite, nous énumérons l'ensemble des paramètres de QdS considérés en nous basant sur les mesures représentées par la figure 1.4.

1- Le **Temps de réponse :**

Défini comme	Le temps nécessaire pour traiter une requête dès l'instant de son envoi jusqu'au moment de la réception de la réponse. Le temps de réponse peut être divisé en temps de communication et en temps d'exécution [85, 86].
Formule	***Tresp = t4 - t1***
Quantification	Milliseconde
Type	Entier

2- Le **Temps d'exécution :**

Défini comme	Le temps mis par le service pour exécuter la requête.
Formule	***Texec = t3 - t2***
Quantification	Milliseconde
Type	Entier

3- Le **Temps de Communication :**

Défini comme	Le temps nécessaire pour le transport de la requête et de sa réponse.
Formule	***Tcomm = Tresp - Texec***
Quantification	Milliseconde
Type	Entier

4- Le **Débit :**

Défini comme	La capacité du fournisseur du service web de traiter les requêtes concurrentes. Il est mesuré en requêtes par seconde [66, 96].
Formule	***Débit = # requêtes/période de temps***
Quantification	Requête/unité de temps telles que minute, seconde, milliseconde, etc.
Type	Réel

5- La **Disponibilité :**

Définie comme	Le pourcentage de requêtes réussies par le fournisseur [66, 85]. Les réponses qui ont échoué, correspondent aux exceptions reçues du côté client.
Formule	***Disponibilité = Nombre de requêtes réussites/-Nombre total de requêtes***
Quantification	Pourcentage
Type	Réel

6- La **Scalabilité :**

Définie comme	La capacité de ne pas dégrader les performances offertes par le service web tout en augmentant le nombre de requêtes simultanées [86].
Formule	***Scalabilité = fn(Performance, Nombre de requêtes)***, où ***fn*** est une fonction qui représente la variation de performance pendant que le nombre de clients augmente (voir le tableau 4.3).
Quantification	Evolution au cours du temps exprimée à l'aide de plusieurs valeurs ou d'une courbe.
Type	Couples de réels ou pourcentage

1.2.2.2 Monitoring de QdS de point de vue client

La plupart des attributs des modèles de la QdS sont mesurés du côté fournisseur et ne nécessitent pas des mesures du côté client. Les mesures du côté client permettent de prendre en considération les caractéristiques de la connexion entre le client et le fournisseur. En effet, l'information de QdS est utilisée pour différencier les fournisseurs de service offrant des services similaires, ce qui est principalement à la charge du client plutôt que le fournisseur.

Considérons deux fournisseurs de service qui offrent les mêmes fonctionnalités, le premier assure un temps d'exécution minimal et un débit élevé (ici, le temps d'exécution et le débit sont mesurables du côté fournisseur), tandis que le deuxième offre un temps d'exécution et un débit acceptables. Le choix du fournisseur peut dépendre, dans cette situation, des QdS mesurables du côté client. Dans le cas où la connexion (le temps de communication) est meilleure avec le deuxième fournisseur du service, le client choisit de se connecter avec celui-ci. Ce scénario peut se produire si les fournisseurs en concurrence sont localisés sur deux réseaux différents : un premier réseau lent et avec des retards de connexion, et un deuxième réseau rapide et sans

délai.

1.2.2.3 Collection des mesures des côtés client et fournisseur

La performance pourrait être mesurée du côté serveur, et dans quelques cas, les mesures effectuées peuvent être différentes de celles effectuées du côté client. Les attributs de performance, de disponibilité et de fiabilité peuvent être collectés automatiquement du côté fournisseur du service ou du côté client. Les autres attributs (sécurité, précision, prix, etc.) ne sont pas dynamiques comme le cas des attributs liés à la performance qui varient dans le temps. Les mesures de ces attributs ont besoin d'être soumises manuellement lors du déploiement et à chaque mise à jour de version. Nous nous concentrons, dans notre travail, sur les QdS génériques et particulièrement sur les attributs de performance des services web, tout en essayant de couvrir ces paramètres des deux côtés : côté client et côté fournisseur du service pour être le plus objectif possible.

1.2.3 Techniques de mesure des QdS

Le monitoring est une étape primordiale dans le processus de mesure de la QdS (voir section 2.3). Il correspond à une étape d'observation du comportement du service et d'extraction des métriques nécessaires pour effectuer les mesures des QdS. La variété potentielle de paramètres de QdS rend cette tâche encore plus délicate. Différentes approches ont été proposées afin d'extraire l'information utile pour l'étude de QdS.

1.2.3.1 Le *Timer* inséré dans le code

Selon l'approche proposée dans [2], une méthode simple de mesurer les caractéristiques de performance des services web peut être développée en ajoutant des fonctionnalités dans le code client du service. Les étapes impliquées dans le développement de ce modèle, capable de mesurer le temps de réponse, se présentent comme suit :

1. Générer la souche logicielle cliente (*stub*) à partir du WSDL du service ;
2. Développer le code du client et ajouter le code de chronométrage du temps ;
3. Compiler le client du service modifié et invoquer les méthodes en question (voir tableau 1.2).

```
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.rpc.*;
import org.apache.soap.util.xml.*;
import mytimer.Timer;
public class EchoServiceClient
{
  private Call call = new Call();
  private URL url = null;
  private String SOAPActionURI = "";
  private SOAPMappingRegistry smr = call.getSOAPMappingRegistry();
  public EchoServiceProxy() throws MalformedURLException
  {
       .
       .
       .
  // Démarrer le Timer
    Timer timer = new Timer();
    timer.start();

    Response resp = call.invoke(url, SOAPActionURI); // Invocation de service

    // Arrêter le Timer
    timer.stop();
    // Afficher le temps de réponse en calculant la différence
    System.out.println("Response Time = " + timer.getDifference());

    // Vérifier la réponse
    if (resp.generatedFault())
    {
      Fault fault = resp.getFault();
      throw new SOAPException(fault.getFaultCode(), fault.getFaultString());
    }
    else
    {
      Parameter retValue = resp.getReturnValue();
      return (java.lang.String)retValue.getValue();
    }
  }
}
```

TAB. 1.2 – Le *Timer* de monitoring

1.2.3.2 Utilisation de l'approche orientée aspect

Le code de mesure de performance peut être implanté en utilisant la Programmation Orientée Aspect (POA) (*Aspect-Oriented Programming, AOP*). Celle-ci est un paradigme de programmation qui permet de réduire fortement les couplages entre les différents aspects techniques d'un logiciel [54]. L'aspect définit des points d'action qui représentent les points de jonction satisfaisant les conditions d'activation de l'aspect responsable de la mesure de performance. Par exemple, le temps de réponse est mesuré en définissant un point de mesure qui détecte le temps avant et après l'invocation d'une méthode du service. Ce type de solution est proposé par les auteurs de [85]. Cette approche permet le monitoring de QdS sans avoir besoin d'accéder à l'implantation du service du côté serveur. Mais ceci reste toujours dépendant de l'implantation, vu que le langage du codage de l'aspect est dépendant du langage de programmation choisi.

1.2.3.3 Modification de la bibliothèque du SOAP

Dans cette approche, la bibliothèque du parsing du message SOAP est modifiée pour enregistrer l'information nécessaire pour la mesure de performance. Cette approche n'engendre pas une grande surcharge sur le CPU comme le cas pour l'approche basée sur la capture de paquets (voir section 1.2.3.5). En plus, elle n'a pas besoin de configurer le code du client comme pour le cas dans l'approche basée sur le proxy (voir section 1.2.3.4). En adoptant cette approche, l'auteur de [96] a proposé un mécanisme pour la mesure automatique des valeurs de QdS pour les services web. Pour démontrer la collection des mesures de performance du côté client, un prototype a été implanté en utilisant la modification de la bibliothèque de gestion des messages SOAP. Le code permettant l'archivage (*logging*) est ajouté avant que le message SOAP ne soit envoyé, et au moment de la réception du message de réponse. Ces informations sont envoyées, par la suite, à une troisième entité responsable du stockage et de la mise à jour des mesures. Toutefois, cette approche souffre d'une limite qui réside dans la dépendance de l'implantation et de la plate-forme. La modification de bibliothèque a besoin d'être disponible sur les différentes implantations et plate-formes. C'est une modification qui doit être établie notamment dans les bibliothèques SOAP des clients ainsi que dans celles des fournisseurs.

1.2.3.4 Approche du proxy

Dans cette approche, un proxy est utilisé comme médiateur de communication entre le client et le serveur. Les messages échangés entre le client et le serveur seront, donc, visibles par le proxy, et les attributs de performance seront mesurés par ce dernier. Les avantages majeurs d'une telle approche sont que le programme de monitoring peut être mis en œuvre indépendamment du matériel et de la plate-forme. De plus, il entraîne moins de surcharge de CPU en comparaison avec l'approche de monitoring de paquets de bas niveau (voir section 1.2.3.5). Les inconvénients d'une telle approche sont dûs au fait que le code client a besoin d'être configuré pour utiliser le proxy. De plus, elle ne peut pas résoudre le problème de transformation directe des messages SOAP [96]. Bien qu'il y ait des travaux qui essaient de déplacer le processus de transformation de messages SOAP au proxy, mais ceci signifie que le code client existant doit être modifié de manière significative. Des outils, tels que *tcpmon*[13] et *wsmonitor*[14], peuvent être utilisés pour l'implantation d'un proxy.

1.2.3.5 Approche basée sur le monitoring de paquets

L'idée principale de cette approche est de capturer les paquets TCP composant les messages SOAP entrants et sortants. Il y a des outils disponibles pour assurer ce type de monitoring tels que *libpcap*[15], *windump*[16] et *winpcap*[17] qui sont utilisés fréquemment pour la surveillance du trafic du réseau et le filtrage de paquets. L'outil *windump* a été utilisé par [83] qui a adopté ce type de monitoring. En effet, il a décrit une approche pour l'extraction détaillée et en temps réel des informations concernant le comportement du service web et sa performance, en se basant sur l'analyse des protocoles TCP/IP et HTTP. L'approche proposée est basée sur la méthode de capture de paquets et puis l'analyse des données TCP. Cette approche utilise, particulièrement, les paramètres de la couche transport pour dériver les métriques et les mesures de la performance. Par exemple, les paramètres observés au niveau de cette couche peuvent être : La taille de la fenêtre d'anticipation et le temps d'aller-retour d'un message (*RTT, Round Trip Time*).

L'avantage principal de cette approche est que le programme de l'écoute peut être

[13]https ://tcpmon.dev.java.net/

[14]https ://wsmonitor.dev.java.net/

[15]"Tcpdump/libpcap project," http ://www.tcpdump.org

[16]http ://www.winpcap.org/windump/

[17]Viano and L. Degioanni, "WinPcap : the Free Packet Capture Architecture for Windows," http ://winpcap.polito.it

complètement indépendant du code de client, et en conséquence, il peut être fait sans accès au code client. Cependant, cette approche présente quelques limites. Premièrement, l'utilitaire de *libpcap* qui capture les paquets de bas niveau est dépendant du matériel et, par conséquent, le programme de monitoring doit être établi pour pouvoir identifier beaucoup de matériels reliés à un protocole (Ethernet, par exemple). Ceci signifie que de nombreuses ressources de CPU sont nécessaires pour décoder, filtrer et reconstruire les paquets. Deuxièmement, si les messages SOAP sont chiffrés ou comprimés, le programme de monitoring sera inefficace puisqu'il nécessite une étape de déchiffrement ou de décompression avant de procéder au monitoring de certaines QdS. D'énormes ressources de CPU seront alors nécessaires, et la complexité du programme augmente considérablement.

1.2.3.6 Environnements d'expérimentation

Pour montrer la faisabilité des différentes approches proposées, des expérimentations ont été menées. En général, les environnements de test choisis sont composés de deux machines pour déployer le service web et le programme client. Les deux machines peuvent être connectées via différents types de connexion réseaux : réseau Internet, réseau Ethernet, avec différents débits. Par exemple, le travail de [86] a exécuté un programme de test de service web en envoyant des requêtes SOAP au service *UbiLearn* (qui est un système d'enseignement à distance basé sur les services web) et a mesuré leur temps de réponse dans le but de tester l'adaptation au facteur d'échelle du système (scalabilité). Pour accomplir les mesures de performance, il a exécuté le même test avec 10, 100 et 500 différents clients concurrents à partir d'une seule machine en utilisant l'outil *TestMaker*, qui est fourni par *PushToTest* [25][18]. Grâce à cet outil, la machine cliente exécute plusieurs processus légers (*Threads*) concurrents. Elle a été placée, en premier lieu, dans le même réseau LAN que le service web avec un débit de 100 Mb/s. Elle a été connectée au serveur, en second lieu, à travers un réseau Internet dont le débit ne dépasse pas quelques centaines de Ko/s. Cependant, de tels environnements d'expérimentation ne permettent pas un passage à l'échelle des tests, même avec le réseau Internet puisque la tentative de tester le service avec plusieurs clients en parallèle se fait avec une seule machine qui simule le comportement de tous ces clients. De plus, on reste toujours dans le cadre de la simulation. En conclusion, l'auteur de [86] suggère le déploiement d'une entité logicielle (qu'il appelle *proxy*) permettant l'ordonnancement du flux massif

[18]C'est un outil à code libre qui envoie des agents de test continuellement pour invoquer le service web

de requêtes. Cependant, cette solution reporte le problème du niveau du serveur au niveau du *proxy*.

1.2.3.7 Synthèse

Une synthèse des travaux cités dans cette section est présentée dans le tableau 1.3. Nous comparons ces travaux par rapport à notre approche (MARQ) et à nos expérimentations présentées dans le chapitre 4.

Nous pouvons classifier les différentes approches de monitoring selon leurs niveaux d'action. Ces niveaux sont décrits par l'architecture présentée dans la figure 1.5. Nous remarquons que le coût engendré par l'interception augmente en descendant du niveau application vers la couche réseau. Par exemple, l'approche basée sur la capture des paquets TCP entraîne plus de surcharge pour le temps d'interception engendré par la décapsulation, le parcours des messages à la fouille des données et leur encapsulation. Par contre, la dépendance du code de l'application augmente en remontant vers le niveau applicatif. Par exemple, l'approche basée sur le *Timer* exige l'accès direct au code et l'utilisation du même langage que le client afin d'effectuer la mesure. Il en est de même pour l'approche qui se base sur l'aspect, elle est en relation directe avec le langage de programmation. L'aspect à développer doit se tisser avec le langage de programmation côté client. Notre approche de monitoring proposée, basée sur les intercepteurs (*handlers*) d'Axis, réussit à réaliser un compromis acceptable en essayant d'être la moins dépendante du code client et serveur.

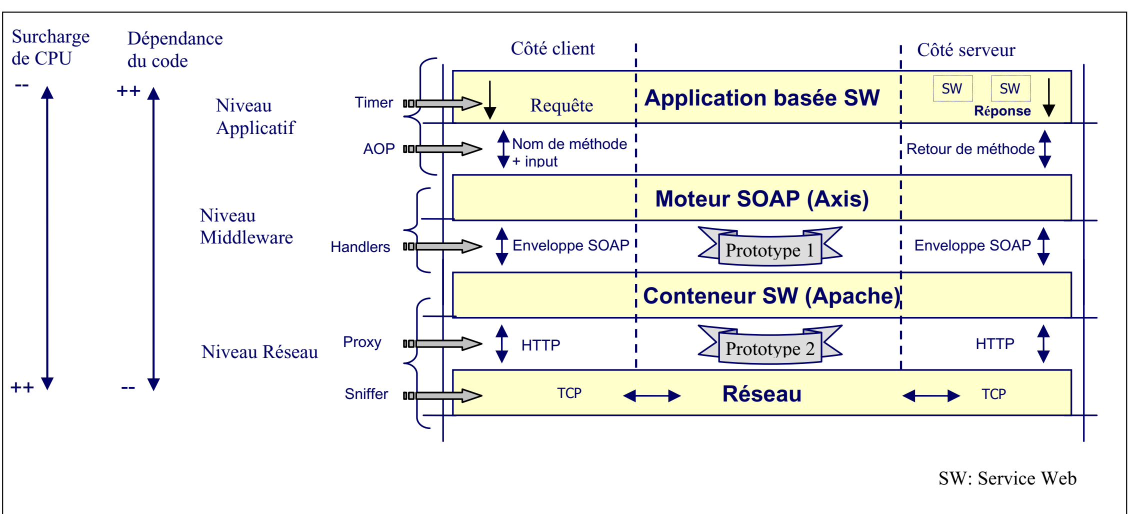

FIG. 1.5 – Les différents niveaux d'interception

		[96]	[86]	[83]	[85]	MARQ
QdS mesurées	Temps de Réponse	x	x	x	x	x
	Débit	x		x		x
	Scalabilité		x			x
	Disponibilité				x	x
Techniques		Modification de bibliothèque du moteur SOAP	Utilisation de l'outil *TestMaker* pour lancer plusieurs clients simultanés	Analyse des protocoles IP/TCP et HTTP	Programmation orientée aspect	Interception de message SOAP + proxy HTTP
Services web analysés		(i) API Google web , (ii) Amazon Box, (iii) WebserviceX.net (valider des adresses emails)	Application propriétaire : des services d'e-learning	Application propriétaire : non décrite	(i) API Google web , (ii) *CaribbeanT*, (iii) *Zip2Geo*	Notre application : SW pour la revue coopérative + Application externe : SW simulant un *FoodShop*
Déploiement		Internet : ADSL 512Kb/s + Cable	LAN 100 Mb/s + Internet ADSL 960Kb/s	LAN 100 Mb/s		*Grid'5000* + LAN 100 Mb/s
Points en faveur		Mesure automatique	Présentation de l'impact de deux méthodes de connexion sur la performance du service	Pas de modification du code du client et du service	Séparation de la mesure du code du client	Expérience à large échelle
Points contre		Dépendance de l'implantation	Les clients sont lancés d'une seule machine	Grande charge du CPU	Dépendance de l'implantation	La grille est un réseau fermé : absence d'interface avec d'autres applications

TAB. 1.3 – La synthèse des approches de mesure

Par ailleurs, les approches qui permettent l'extension de message SOAP sont celles basées sur le proxy et les intercepteurs. Cependant, comme nous l'avons signalé précédemment, l'approche proxy est gourmande du point de vue ressources afin d'analyser tous les paquets HTTP, et de choisir un emplacement convenable pour réaliser l'extension. Par contre, le message SOAP est bien approprié pour cette extension. En effet, l'entête SOAP (qui est basée sur XML) est destinée pour ce type d'enrichissement des messages échangés entre client/serveur [40].

1.3 L'auto-réparation

L'auto-réparation (*Self-Healing*) est la propriété qui permet à un système de percevoir qu'il ne fonctionne pas convenablement et d'assurer les ajustements nécessaires afin de se restituer à l'état normal, sans l'intervention humaine [39]. L'auto-réparation est une propriété importante pour les « systèmes autonomes ».

La notion de calcul autonome a été promue par IBM en 2001, avec l'objectif de réduire le coût total des systèmes et de simplifier la gestion de leur cycle de vie, en dépit de leur complexité croissante. Le développement d'applications auto-gérables, dotées de la robustesse, la proactivité et l'adaptation est le but des systèmes autonomes [53]. Les systèmes autonomes incluent des propriétés d'auto-réparation, d'auto-adaptation, d'auto-reconfiguration, d'auto-optimisation et d'auto-protection :

- L'auto-réparation : le système détecte, localise et répare automatiquement les problèmes logiciels et matériels [103, 79].
- L'auto-adaptation : c'est la capacité du système de se configurer dynamiquement à la volée. Un système logiciel peut être adapté immédiatement au déploiement d'un nouveau composant ou au changement de son architecture [103, 79].
- L'auto-reconfiguration : le système configure automatiquement ses composants tout en suivant des politiques de haut niveau. Le reste du système s'ajuste automatiquement et d'une façon transparente. [53].
- L'auto-optimisation : les composants d'un système cherchent continuellement les opportunités d'amélioration de leur performance et de leur efficacité [103, 53].
- L'auto-protection : le système se défend automatiquement contre les attaques malicieuses et les échecs en cascade. Ceci repose sur l'anticipation des dégradations et la prévention des échecs du système [103, 79].

1.3.1 Approches existantes d'auto-réparation

Nous distinguons trois catégories principales des solutions utilisées pour implanter les applications d'auto-réparation, à savoir la catégorie à base de modèle, la catégorie à base de middleware et la catégorie à base de plate-forme. Pour les approches à base de modèle [104, 22], les développeurs implantent à partir de zéro toutes les actions en relation avec le monitoring, le diagnostic et la réparation. Pour les approches à base de middleware [55, 71, 45], les développeurs construisent leurs solutions en adaptant les primitives d'auto-réparation offertes par les APIs du middleware à leur contexte d'application. La troisième catégorie se base sur les plate-formes qui fournissent des

composantes réutilisables pour mettre en œuvre des stratégies d'auto-réparation [23, 90, 6]. Dans ce qui suit, nous détaillons les différentes caractéristiques des approches à base de modèle, de middleware et de plate-forme pour l'auto-réparation. Nous étudions aussi les différents facteurs qui peuvent expliquer le choix d'une approche particulière.

1.3.1.1 Les approches basées Modèle

Avec ce type d'approche, l'utilisateur est mené à implanter tous les modules du cycle d'auto-réparation. Aucune primitive n'est offerte. Dans cette catégorie, nous nous concentrons particulièrement sur les approches architecturales.

Dans les applications d'auto-réparation basées modèle, les architectures offrent une abstraction des composants du système à un haut niveau, tout en permettant une analyse et une interprétation plus faciles. Généralement, les contraintes architecturales sont exprimées explicitement pour garantir l'évolution de l'application. Des modules de gestion sont proposés afin d'assurer le bon fonctionnement du processus d'auto-réparation. Ces modules se basent sur des entités logicielles mettant en œuvre les actions de monitoring, la stratégie du diagnostic et la réparation des dégradations.

Le travail décrit dans [104] présente *Kinesthetics eXtreme* (*KX*) qui est un cadre architectural générique des applications d'auto-réparation à base de middleware léger. C'est une architecture décentralisée qui repose sur un système à base d'événements. L'architecture présentée, peut être, soit implantée au niveau middleware, soit au niveau application. Le diagnostic se fonde sur des règles de détection de dégradation. Le modèle *KX* observe les propriétés fonctionnelles d'une application afin (i) d'analyser la sémantique des messages échangés (par exemple, la détection de SPAM dans un système de courrier électronique) (ii) et d'examiner le début et la fin de chaque invocation d'une opération pour détecter le service qui interrompt l'exécution. Selon ses auteurs, *KX* est scalable et peut être appliqué pour les systèmes distribués. Cependant, il souffre de quelques limites. D'une part, les entités logicielles assurant l'auto-réparation sont ajoutées manuellement au prototype présenté. D'autre part, les plans de réparation sont édités manuellement.

Rainbow [21, 22] est un cadre architectural pour les applications auto-adaptatives. Il contient deux couches d'action. La première couche est la couche système qui assure la collection d'information sur l'état du système et exécute les plans de réparation. La deuxième couche est la couche architecture qui dresse l'architecture courante du système, vérifie la violation des contraintes (en se basant sur la différenciation

de l'architecture réelle avec l'architecture conçue) et détermine les adaptations nécessaires. La couche système et la couche architecture interagissent à travers une *infrastructure de translation*. Cependant, une expérimentation, qui a porté sur cette approche externe d'auto-adaptation [29], a montré que sa mise en œuvre engendre un ralentissement considérable du comportement du système. Aussi, le plan de réparation est construit manuellement et puis inséré dans le code sans vérification de sa consistance.

D'autres approches basées modèle utilisent les styles architecturaux pour concevoir l'auto-réparation. Un style architectural représente une collection de décisions de conception qui sont déjà appliquées et peuvent être ré-utilisées. Il se compose de quelques caractéristiques clés et des règles pour les combiner afin que l'intégrité architecturale soit préservée. Dans cette catégorie, on mentionne Prism [65, 68] qui est un cadre architectural d'auto-réparation des applications orientées composant. Il est composé de deux couches. La première est la couche application qui inclut les composantes fonctionnelles et les messages échangés entre eux. La deuxième représente la couche d'auto-réparation, où les composants agissent comme des acteurs de reconfiguration. Ils facilitent le monitoring, la décision et le déploiement des plans. Ces composants (de la couche d'auto-réparation) sont conscients des composants de la couche application et peuvent initier des actions d'interactions avec eux, mais pas vice versa. Le style proposé, malgré son niveau très abstrait, semble être complet dans le sens où il couvre la plupart des exigences pour des applications d'auto-réparation comme l'adaptabilité, la dynamique, la conscience, l'observabilité, l'autonomie, la robustesse, la répartition et la mobilité. C'est une des rares approches qui considèrent la mobilité. Pourtant, ce cadre architectural ne vise pas de solution pour l'application à un domaine spécifique, mais c'est un cadre général pour concevoir des applications auto-réparables avec des exigences. Les problèmes spécifiques, comme la coordination et les politiques n'ont pas été considérés dans cette approche. La validité d'application du style a été démontrée.

Taylor et al. [76, 28] présentent une approche architecturale d'auto-réparation. Dans [76], Taylor et Oreizy proposent une approche basée sur les architectures pour l'auto-adaptation. L'approche n'a pas été implantée et n'a pas été appliquée à un domaine spécifique ou à une application. Elle insiste sur l'intégrité du système qui exige qu'on assure la consistance, l'exactitude et la coordination des changements. Dans [28], Taylor et Dashofy raffinent l'architecture proposée dans [76]. Ils exposent une approche à base de style pour les applications auto-réparables. Ils focalisent sur l'infrastructure afin de soutenir la création, la validation et l'exécution des stratégies de

réparation. L'architecture est décrite avec *xADL 2.0* qui est un langage de description des architectures à base de XML. L'infrastructure est principalement conçue pour les systèmes orientés événement. Ils utilisent la technique de *différenciation* des architectures pour diagnostiquer l'état du système et pour générer des plans de réparation.

1.3.1.2 Les approches basées Middleware

Les approches à base de middleware fournissent des primitives aidant les développeurs à implanter les applications auto-réparables. Dans ce qui suit, nous présentons ces approches. Les caractéristiques de chaque approche sont résumées dans le tableau 1.4.

DynamicTAO [55] est un middleware réflexif (extension de TAO [87], un ORB *open-source* de CORBA). Il permet la détection des changements de l'application et la recharge des nouvelles implantations des composantes qui peuvent être reliées au système au cours de l'exécution. Ces caractéristiques sont accomplies par l'utilisation d'une collection d'entités nommée *Configurators* qui maintiennent les informations de dépendance entre les composants qu'ils gèrent. L'entité *DynamicConfigurator* inspecte les implantations des composants du système et reconfigure l'application au cours de l'exécution à travers des actions de recharge ou de suppression des implantations à partir de la *Repository*. La scalabilité de *DynamicTAO* n'est pas prouvée. Elle est seulement évaluée avec un exemple simple (*getHello()*). L'infrastructure de *DynamicTAO* inclut deux services de gestion de la sécurité. Le premier est utilisé pour crypter/décrypter le contenu des messages et le deuxième authentifie les intervenants pour contrôler l'accès. La stratégie de sécurité peut être chargée et reliée dynamiquement au système au cours de son exécution.

Eternal [71] est une extension tolérante aux fautes du middleware CORBA en se basant sur la réplication des composants. Il est développé comme une couche externe pour la réparation au-dessous de la couche logicielle initiale de CORBA. Le monitoring est fondé sur les intercepteurs. *Eternal* intercepte les requêtes des clients et les transfère vers une réplique en cas de dégradation. Une simple application de test est réalisée pour mesurer la performance de *Eternal* en cas d'échec. Elle montre le temps de recouvrement nécessaire par la reconfiguration tout en variant la taille de l'état du composant à transférer à travers le réseau. La courbe montre que le temps de recouvrement augmente tant que la taille des données transférées accroît.

Eternal utilise le service de sécurité de CORBA (*SecIOP*[19]) et intègre SSL[20] pour protéger les messages échangés. Aussi, il exécute un pare-feu pour filtrer les requêtes et n'accepter que les clients authentifiés.

Le middleware **OpenORB** [14] est construit en se basant sur la programmation réflexive. Son architecture réflexive utilise des protocoles de méta-objet pour mettre en œuvre des intégrations qui supportent l'adaptation dynamique au cours de l'exécution. Les méta-modèles permettent le monitoring et la réparation des dégradations afin de préserver le style architectural. A travers le méta-modèle d'interception, il inspecte les requêtes échangées entre les clients et les composants. Le méta-modèle d'interface fournit un accès à l'implantation du composant alors que le méta-modèle d'architecture fournit un accès au graphe d'objets. Le prototype d'illustration traite la qualité d'une transmission continue d'un flux de média tout en inspectant et reconfigurant les ressources disponibles (l'unité centrale, le réseau, etc.) afin de maintenir la QdS [59]. Cette étude de cas démontre que *OpenORB* fournit un support suffisant en faveur des applications multimédia à petite échelle. Selon [14], la sécurisation des communications peut être réalisée aussi à travers les intercepteurs.

JavaPod [17] est un middleware réflexif qui focalise sur la séparation et la composition de propriétés fonctionnelles et non-fonctionnelles dans un cadre distribué. L'adaptation est assurée avec la composition d'objet. Elle est réalisée en étendant dynamiquement des méthodes avec de nouvelles implantations. Les auteurs développent une extension de Java pour simplifier l'implantation de leur approche. Ils fournissent de nouveaux protocoles pour gérer les connexions et gouverner les fautes. Pour la sécurité, *JavaPod* implante une liste de contrôle d'accès, qui permet de régir l'accès au niveau des méthodes pour chaque utilisateur. Pour l'évaluation, une application d'e-learning, appelée *Baghera*, a été développée. Le test de performance montre l'existence d'une surcharge considérable à cause du mécanisme de composition. Mais, cette surcharge est négligeable comparée au temps de communication qui est amélioré.

CME [45] (*Connection Management Engine*) est un middleware pour les applications réseaux. Il gère des connexions logiques (appelées *canaux*) entre les pairs échangeant des messages. Il contrôle les canaux pour identifier les intervenants communicants et déterminer le début et la fin des connexions et la quantité d'information échangée. *CME* utilise un mécanisme de politique pour faciliter l'administration du réseau. Les politiques représentent des exigences d'adaptation qu'il doit garan-

[19]SECure Inter-ORB Protocol
[20]Secure Socket Layer

tir pendant l'exécution de l'application. *CME* facilite le contrôle de la sécurité en enregistrant les pairs échangeant des requêtes. La scalabilité n'est pas vérifiée. Le middleware est seulement évalué avec un prototype sur PDA[21]. Il agit sur la couche réseau. Par conséquent, il est au-dessous du système d'exploitation et il peut supporter plusieurs types d'application.

Il existe d'autres middlewares tels que ceux proposés dans [61] et [107]. Les auteurs de [61] présentent un middleware pour gérer la QdS au sein d'un groupe de serveurs d'application. Cette approche propose trois modules, sous forme de composants, assurant l'adaptation à la QdS. Le premier est le module de *Monitoring* qui est responsable de l'observation et de la détection la dégradation de la QdS. Le deuxième est le module de *Configuration* qui se charge de garantir la disponibilité du service tout en aménageant le groupe de serveurs d'application afin de respecter la spécification du SLA[22] (niveau d'agrément du service). Le troisième est le module de *Load Balancing* qui agit comme un proxy HTTP routant les requêtes des clients selon la disponibilité du service. Les auteurs de [107] traitent la gestion de la QdS d'une composition de services web sur deux niveaux. Le premier niveau s'occupe de la sélection d'un service web qui satisfait les exigences de l'utilisateur spécifiées dans un modèle de QdS. Le deuxième niveau gère l'adaptation du processus en cours, suite à une dégradation de la QdS en substituant le service web déficient.

DynamicTAO, *Eternal*, *JavaPod* et *CME* sont développés sous forme d'une couche externe pour réparer les systèmes en cours d'exécution. Cependant, OpenORB peut être externe ou interne à l'application. Lorsqu'il est externe, le monitoring et la gestion de la réparation sont présentés à l'application/système sous forme d'un service externe. S'il est interne, les composants de monitoring et de gestion de la réparation sont insérés dans le code de l'application. Avec *Eternel*, les actions de réparation sont limitées à la réplication. *DynamicTAO* ne fournit pas des mécanismes de détection de dégradation et de diagnostic. C'est à l'utilisateur de se renseigner sur l'état du système et de prévoir une nouvelle version du composant en cas de dégradation. *JavaPod* n'assure pas le diagnostic et la planification. Toutefois, toutes les étapes sont automatisées à travers d'autres logiciels médiateurs.

[21] Personal Digital Assistant
[22] Service Level Agreement

Middleware	DynamicTAO [55]	Eternal [71]	OpenORB [14]	JavaPod [17]	CME [45]
Mécanismes de monitoring de la QdS	*Event Collector* : Observe le comportement des composantes et produit des événements pertinents à la QdS	*DynamicConfigurator* : Inspecte les implantations des composants	*Interceptor* : Utilise trois techniques : checkpoint, pinging (message *i-am-alive*) et logging	*Server Container* : Details non disponible	*Connection Monitor* : Observe les artifices de réseau (modem) et les entités de contrôle des protocoles (table de routage)
Mécanismes de détection de dégradation de la QdS	*Monitor* : Collecte les événements de QdS et reporte les comportements anormales	*L'utilisateur* : Interroge la base de données afin d'inspecter la QdS	*Fault Detector* : Communique l'occurrence des faults au *Fault Notifier*	Détails non disponible	*Adaptation Trigger* : Déclenche des adaptations basées sur un contexte avec critères pré-définis
Mécanismes de diagnostic et de décision	*Strategy Selectors* : Sélectionne une stratégie d'adaptation appropriée basée sur le feedback des *Monitors*	*L'utilisateur* : Choisit l'implantation appropriée de chaque catégorie.	*Fault Notifier* : Distribue les événements de notification de fautes au *Replication Manager*	Détails non disponible	*Adaptation Selector* : Choisit l'approche d'adaptation appropriée
Mécanismes de réparation	*Strategy Activators* : Implante une stratégie particulière, e.g. par manipulation du graphe de composant tout en préservant le style architectural	*DynamicConfigurator* : Charge et connecte la nouvelle implantation du service (*load*, *resume*, *suspend*, *remove*, *delete*, etc.)	*Replication Manager* : Transfère l'état d'un composant vers une réplique	*Composition* : Étend les composants par de nouvelles implantation et re-dirige les requêtes vers une nouvelle implantation	*Adaptation Executor* : Exécute des commandes (de type ouvrir et changer de canal) pour changer le comportement du système
Externe/Interne	Externe or Interne	Externe	Externe	Externe	Externe
Langage de programmation	Python	C++ / Java	C++	eJava (extension de Java)	Java
Scalabilité	Haut	Bas	Médium	Médium	Bas
Sécurité	Bas	Haut	Haut	Médium	Médium
Domaine d'application	Composant/Service web	Composant	Composant	Composant/Composant Mobile	Artifice réseau (Ordinateur, PDA,...) et protocole (LAN, GPRS,...)
Niveau de réparation	Structurel : Modifie l'architecture	Fonctionnel : Recharge une nouvelle implantation du composant	Structurel : Connecte les clients aux répliques	Fonctionnel : Étend les composants avec de nouvelles implantations	Communication : Maintient le niveau de communication tout en ajustant ou changeant de canaux

Tab. 1.4 – La comparaison des approches d'auto-réparation basées middleware

1.3.1.3 Les approches basées Plate-forme

Dans cette section, nous présentons les principales plate-formes suggérées et employées pour développer des applications auto-réparables. L'évaluation est fondée sur des critères tels que les fonctionnalités offertes de la programmation autonome, les composantes utilisées et les types d'architectures supportés par ces plate-formes.

Unity *[23, 94]* : Le projet *Unity* focalise ses efforts sur la manière dont les composants et les relations entre eux, assurent l'auto-réparation des systèmes informatiques. Le projet *Unity* implante un prototype d'un système autonome conçu pour supporter et valider un environnement auto-réparable.

Au cours de l'exécution, il réattribue et reconfigure les ressources du système pour optimiser son comportement selon des politiques spécifiées. Dans cette approche, chaque composant incorpore une partie d'auto-réparation de façon à ce qu'il devienne auto-réparable. Les composants fonctionnels de *Unity* sont :
- *Application Environment Manager* : il est responsable de la gestion des communications entre les composants de l'application et la prédiction de la disponibilité des ressources ;
- *Resource Arbiter* : il gère le partage et l'allocation de ressources ;
- *Registry* : il permet la localisation d'un composant ;
- *Policy Repository* : il représente l'interface d'administration ;
- *Sentinel* : il est utilisé par un composant pour observer le fonctionnement d'un autre ;
- *Solution Manager* : il est responsable de la maintenance et de la réparation.

Le monitoring est réalisé par tous les composants y compris les composants défectueux (s'ils existent) qui peuvent provoquer des dommages pour le système. Par conséquent, il est nécessaire de contrôler les données du monitoring. Les auteurs proposent l'ajout de politique pour filtrer les données rassemblées.

PAC-MEN[23] *[90]* : *PAC-MEN* fournit des concepts et des techniques d'auto-réparation pour une gamme de plate-formes incluant les ordinateurs, les ordinateurs portables, les PDAs etc. Il est fondé sur *la réaction réflexive* pour réagir aux dégradations de la QdS et aux *signes vitaux* pour évaluer l'état opérationnel (inspiré des mécanismes humains). Dans cette approche pair-à-pair, chaque pair dans le système est un élément auto-réparable :

- Chaque pair est responsable de son propre comportement intérieur d'auto-

[23] Personal Autonomic Computing Monitoring ENvironment

réparation ;
- Chaque pair peut être étendu pour inclure un monitoring partagé de l'environnement extérieur et puis il informe les membres du groupe d'événements qui nécessitent des actions individuelles.

L'approche *PAC-MEN* propose la mise en œuvre d'un serveur d'administration dans chaque pair appelé *Autonomic Manager* (*AM*), qui partage les données et les décisions de gestion. Cela permet une collaboration entre les *AMs* moins rigide que la collaboration courante des architectures d'information autonomes. Cependant, ce style est plus dynamique et plus centralisé.

CODA[24] *[84]* : *CODA* applique des concepts tels que l'auto-organisation, l'autorégulation et la viabilité pour tirer une architecture intelligente, qui peut réagir aux échecs de l'accomplissement des objectifs et qui recherche pro-activement des motifs (*patterns*) de comportements réussis. *CODA* est une approche en couche. Elle est composée de cinq couches :

- *Operations* : qui représente les opérations métiers du système ;
- *Monitor Operations* : qui assure le monitoring interne ;
- *Monitor of the Monitors* : qui assure le monitoring externe ;
- *Control* : qui appréhende les fautes et prédit la stratégies de réparation ;
- *Command* : qui identifie les menaces et prend les décisions.

Cactus *[44]* : Le système *Cactus* fournit un support d'adaptation dynamique et offre une solution potentielle pour construire des logiciels survivables -capables de tolérer les attaques intentionnelles et les échecs accidentels- dans les systèmes en réseaux. Un service dans *Cactus* est implanté comme un ensemble d'intercepteurs qui réagissent quand un événement se déclenche afin de garantir des attributs de la QdS (la fiabilité, la performance et la sécurité). En cas d'une application basée sur CORBA, le service est un protocole de communication qui réside dans la pile de protocoles au sommet du service de communication du plus bas niveau tel que UDP[25]. Les intercepteurs réagissent quand un message est échangé entre le client et le serveur. *Cactus* propose plusieurs types d'intercepteurs. Chaque intercepteur agit d'une façon indépendante des autres intercepteurs. *Cactus* offre la possibilité de choisir l'intercepteur à appliquer à chaque service. De ce fait, il est plus configurable et plus adaptable.

[24] Complex Organic Distributed Architecture
[25] User Datagram Protocol

Plate-formes	Unity [23, 94]	PAC-MEN [90]	CODA [84]	Cactus [44]	MAIS [6]	Jade [30]	MARQ
Architecture	Centralisée	Décentralisée	Centralisée	Décentralisée	Centralisée	Centralisée	Centralisée
Type d'architecture	Client/Serveur	pair-à-pair, Grille	Client/Serveur	Client/Serveur	Client/Serveur	Client/Serveur	Client/Serveur
Intervention Humaine	Minimale	Sans	Sans	Sans	Interaction	Sans	Sans
Propriétés Autonomes	Auto-configuration, auto-optimisation, auto-protection, auto-réparation	Auto-réparation, auto-monitoring	Auto-organisation, auto-monitoring	Auto-configuration, auto-adaptation	Auto-adaptation, auto-optimisation	Auto-réparation, auto-optimisation	Auto-réparation
Composants d'auto-réparation	Tous les composants	Tous les composants (collaboration)	Interne, externe et en couche	Externe : *Micro-protocole*	Interne, externe et en couche	Externe	Externe
Langage de programming	Java	Java	Java	C/C++/Java	Java	Java	Java
Interface de présentation	Interface graphique utilisateur	Sans	Sans	Interface de monitoring	Sans	Sans	Visualisateur de monitoring
Composant de monitoring	*Sentinel*	*Signes vitaux*	*Monitor operations, Monitor of the Monitors*	*Event Handlers*	*Diagnoser* et *Inspector*	*Managed Element* (Interface d'introspection)	*Moniteur*
Composant de réparation	*Solution manager*	*Reflex reaction*	*Command*	*Event Handlers*	*Recovery actions*	*Managed Element* (Interface de reconfiguration)	*Connecteur*
Niveau d'application	Application	Application et appareil (*device*)	Application	Réseau	Application, réseau et appareil (*device*)	Application et ressource	Application
Support des SW	Oui	Non	Oui	Non	Oui	Non	Oui

Tab. 1.5 – La comparaison des approches d'auto-réparation basées plate-forme

MAIS[26] *[6]* : Le projet *MAIS* étudie l'adaptabilité à tous les niveaux dans les systèmes d'information ; du niveau application au niveau réseau et aussi au niveau matériel (les ordinateurs, les portables, les téléphones cellulaires, etc.). Plusieurs niveaux d'adaptabilité sont considérés : le niveau supérieur (le niveau application), le niveau middleware (le niveau des services web) et le niveau inférieur (le niveau infrastructure). *MAIS* constitue un environnement pour invoquer les services web composites, mobiles et conscients du contexte d'une façon adaptable. L'architecture *MAIS* implante des services d'analyse de faute et de recouvrement au cours de l'exécution. Elle détecte les fautes en inspectant les requêtes et les réponses et en les analysant par un *diagnoser*. Cette architecture fournit quatre modules pour gérer les actions de recouvrement, à savoir : « la réallocation », « la substitution », « le générateur de *wrapper* »et « les modules de qualité ».

Jade *[30]* : C'est un environnement d'administration autonome d'infrastructures logicielles. Il permet d'avoir une vue abstraite sur les logiciels de l'application et d'intervenir en cas de panne d'une partie du système. Il se base sur le principe de duplication pour maintenir la disponibilité du service et la variation des ressources allouées selon la charge pour garantir la scalabilité. *Jade* est composé de deux parties :

- *Managed Element* : qui encapsule chaque logiciel et offre une interface d'administration ;
- *Autonomic Manager* : qui implante les politiques de gestion d'administration (réparation et optimisation). Il observer et agit sur le système à travers les interfaces des *Managed Elements*.

Il y a d'autres plate-formes dans la littérature qui s'intéressent à l'auto-réparation. C'est le cas des travaux de [74], [43], [56] et [50]. Le travail [56] traite la gestion des services et des ressources du réseau. Il est présenté en trois couches. La première couche inclut les *Reconfiguration Executors* qui adaptent les composants du réseau. La deuxième couche est responsable du monitoring et du diagnostic de la dégradation de la QdS. La dernière couche est une interface graphique de supervision utilisée pour administrer le système. L'abstraction des données échangées entre les couches est basée sur les capacités de XML. Le travail présenté dans [50] offre un système de monitoring composé de trois parties. La première partie génère le rapport de monitoring. La deuxième partie rassemble, centralise et analyse les données de monitoring. La troisième partie affiche graphiquement les données de monitoring à l'utilisateur. Afin d'améliorer leur approche, les auteurs proposent l'intégration automatique de

[26]Mobile Adaptive Information Systems

leur système de monitoring pour tous les types d'applications. Le travail décrit dans [74] présente un environnement d'exécution réflexif et adaptable dynamiquement. Il permet la construction dynamique d'applications et de middleware réflexif. Les auteurs de [43] proposent un mécanisme d'ajout et d'adaptation de service système en se basant sur un langage de reconfiguration adaptable.

Dans le tableau 1.5, nous présentons les propriétés de chaque plate-forme. A l'exception de *Unity* et de *MAIS*, la plupart des plate-formes ne permettent pas d'interaction entre l'utilisateur ou l'administrateur et l'application. De plus, *Unity* et *PAC-MEN* admettent que tous les composants sont auto-réparables, mais ce dernier (*PAC-MEN*) suppose que tous les composants collaborent pour s'observer. Le langage de programmation supporté par toutes les plate-formes est Java. Mais il y a une autre version de *Cactus* qui est développée en C/C++. *CODA*, *Unity*, *MAIS* et *MARQ* (notre approche) soutiennent les applications auto-réparables à base de services web. Les autres peuvent investir pour les soutenir. *Cactus* intègre des techniques d'auto-réparation au niveau communication. Afin de prendre en considération les services web, il peut étendre le protocole SOAP et prendre en considération les intercepteurs SOAP pour maintenir la QdS.

1.3.2 Approches existantes de monitoring

Plusieurs approches de monitoring existent dans la littérature. Généralement, et suite à une dégradation détectée, le monitoring est suivi par des actions de réparation [12, 13, 89] ou des pénalités en cas de contrat régissant l'interaction entre le client et le fournisseur du service [97]. Ces contrats sont établis sous forme d'accords qui expriment les capacités des fournisseurs de services et utilisent la sémantique des messages XML. Dans ce qui suit, nous détaillons un ensemble de ces approches.

Les auteurs de [12] présentent une approche de monitoring des applications à base de BPEL, et proposent deux niveaux de monitoring qui se présentent comme suit : le niveau instance, qui s'intéresse à une seule instance du processus BPEL et le niveau classe, qui collecte des informations concernant toutes les instances. Cette approche sépare la logique métier de la tâche de monitoring qui vont être gérées dans deux environnements s'exécutant en parallèle et communicant à travers un médiateur. Elle propose un nouveau langage de spécification des moniteurs, *RTML*[27], qui supporte les types boolean, statistique, et temps. Les moniteurs sont générés et déployés automatiquement à partir d'une spécification. Ils peuvent déclencher des

[27] Run-Time Monitoring specification Language

alarmes si le nombre d'interactions dépasse un certain seuil (contrainte sur le débit des requêtes traitées pour le cas de payement en ligne). Cette approche ne distingue pas les communications asynchrones et synchrones.

Le travail [13] présente les directives de monitoring sous forme de règles choisies et appliquées durant la phase de conception ou au cours de l'exécution d'un processus BPEL. Les moniteurs sont exprimés à travers le langage proposé *WSCoL*[28] et sont mis en œuve à travers le langage d'assertion *JML*[29] qui permet de réaliser des pré- et post-conditions et d'autres traitements à travers des annotations du code. Cette approche sépare entre la logique métier et la tâche de monitoring. En plus, elle permet d'activer et de désactiver les règles de monitoring en temps réel, ce qui permet l'ajustement du ratio de la charge de monitoring par rapport à la performance, tout en évitant les surcharges. Cette approche ne distingue pas les communications asynchrones de celles qui sont synchrones.

Les auteurs de [89] proposent un cadre formel pour le monitoring des exigences d'une composition de services avec BPEL. Leur approche utilise le calcul d'événements (*event calculus*) pour spécifier les propriétés du comportement du système à observer. Les événements sont alors observés et enregistrés au cours de l'exécution. Puis, les incohérences entre le fonctionnement attendu du système et le fonctionnement observé sont découvertes en utilisant des algorithmes de raisonnement déductifs. L'approche proposée est moins intrusive du fait que le monitoring est réalisé en parallèle avec l'exécution de la logique métier. Ceci a moins d'impact sur la performance, mais produit à un moindre degré de temps de réponse lorsque des anomalies surviennent. La validité d'application de l'approche proposée est entravée par sa complexité, particulièrement, pour les utilisateurs non familiers avec le calcul d'événements.

WSOL[30] [97] est un langage basé sur XML qui permet de spécifier des niveaux de QdS attachés à la description WSDL des services. *WSOL* s'appuie sur la notion de classes de services qui représentent des SLAs simples dans lesquels le client choisit, parmi les services proposés par les fournisseurs, celui dont la QdS est convenable à ses besoins. Le langage *WSOL* permet d'exprimer des contraintes fonctionnelles, de QdS, de droit d'accès et de prix. Une offre de service *WSOL* peut aussi spécifier des entités responsables de surveiller une contrainte particulière sur un service. Ces entités sont appelées *SOAP intermediary* et elles se basent sur la notion d'intercepteur.

[28]Web Service Constraint Language
[29]Java Modelling Language : http ://www.eecs.ucf.edu/~leavens/JML/
[30]Web Service Offerings Language

L'approche WSLA[31] [52] propose, à la fois, un langage de spécification de SLA ainsi qu'une infrastructure pour établir et surveiller les contrats lors de l'exécution des services. Le langage proposé [62] est basé sur XML *Schema* et décrit les parties en présence, la description des services (caractéristiques et paramètres observables) et les obligations. Le langage WSLA est riche et il introduit en particulier (i) les parties impliquées dans l'agrément, (ii) les métriques de la QdS ainsi que les directives de mesures et (iii) les actions à exécuter dans les cas particuliers. La phase de négociation s'appuie sur des échanges successifs de gabarits de contrats WSLA (*WSLA template*) entre les fournisseurs et les clients des services. Le système de gestion s'intéresse plus à la surveillance des contrats et à la réservation des ressources nécessaires du côté fournisseur pour supporter les niveaux de services offerts.

SLAng [57] est aussi un langage de définition de SLA qui considère la définition de la QdS avec une focalisation sur les services réseaux et de middleware et sur le maintien de la QdS de bout en bout. Chaque architecture de mesure est décrite avec un schéma *SLAng*. *SLAng* gère, aussi, la QdS des services web, mais il ne donne pas des détails techniques sur la mise en œuvre. Dans [47], la sémantique de SLAng est définie en UML[32] et OCL[33] et des notions de compositionnalité de SLAs sont introduites.

Généralement, ces approches se limitent à la mesure de la QdS pour vérifier un contrat entre le client et le fournisseur. En cas de violation de ce contrat, le fournisseur doit payer une pénalité en contre partie de la QdS non assurée [97]. Notre approche procède par des actions de reconfiguration afin de remédier aux dégradation de la QdS. En plus, elle est générique. D'une part, elle est indépendante de la logique métier des services et de l'application. D'autre part, elle est applicable dans les cas des architectures orchestrées et choréographées. Cependant, on considère les dégradations de la QdS et non les fautes fonctionnelles comme le cas de [89].

1.3.3 Approches existantes de substitution

Dans la littérature, il existe plusieurs approches qui se sont intéressées à la réparation. Plusieurs techniques sont proposées telles que la *substitution* [88, 14], le *wrapping* [6, 16] et la *redondance* [32, 38]. Pour les services web, la réparation repose, généralement, sur une entité déployée entre le client et le fournisseur du service telle

[31] Web Service Level Agreement
[32] Unified Modeling Language
[33] Object Constraint Language

que le *médiateur* dans [100], et la *communauté* dans [91].

Dans [91], les auteurs fournissent une solution pour la substitution. Ils proposent le concept de communauté des services web (*WS community*) qui regroupe les services web ayant les mêmes fonctionnalités. Cette communauté est représentée par une interface abstraite (*Abstract WS Interface*) qui est une interface commune pour tous les services web équivalents. La cohérence de projection (*mapping*) entre l'interface du service concret et celle de l'abstrait est assurée dynamiquement tout en respectant les règles de projection.

Le papier de [100] propose de grouper un ou plusieurs services web avec leurs caractéristiques de QdS dans une interface médiatrice et de publier le résultat comme un service web standard. Les clients invoquent cette enveloppe (appelée aussi *service web virtuel*) qui est responsable de l'invocation du fournisseur réel. A chaque invocation, le médiateur choisit le service qui offre la meilleure QdS, en se basant sur l'historique de son invocation.

Le travail décrit dans [63] décrit la substitution d'un service web avec état et faisant partie d'une orchestration. Le nouveau service de substitution est choisi parmi une liste de services découverte suite à une compatibilité sémantique avec le service actuel devenu indisponible. Ce nouveau service doit pouvoir synchroniser son état de ressource avec l'état de ressource du service défaillant, et ce avant de lancer l'opération de substitution. Ce transfert d'état est réalisé grâce au message *SetResourceProperties*. Toutefois, les services supportant ce type de message de synchronisation d'état sont limités. Il faut avoir les droits d'accès nécessaires pour mettre à jour l'état d'un service.

Contrairement à l'approche [63], notre approche gère les services web sans état. Elle connecte les clients à un service, et ne déploie un nouveau connecteur qu'en cas de dégradation. Les auteurs de [100] proposent de changer la connexion à chaque requête, ce qui peut engendrer des charges additionnelles et affecter l'application. Notre approche réalise la reconfiguration en se basant sur une liste de services équivalents comme l'approche [91], sauf qu'elle se connecte à un seul service à la fois, et la connexion vers un autre service équivalent nécessite une action de reconfiguration pour la mettre en œuvre.

1.4 La problématique

Cette partie présente les problématiques confrontées tout au long de la réalisation de ce travail.

1.4.1 Système d'auto-réparation : Externe vs Interne

Les actions d'auto-réparation peuvent être réalisées d'une manière *interne* ou *externe* par rapport à l'application. Dans une stratégie interne, le code d'auto-réparation est fusionné avec le code de l'application. Cependant, dans une stratégie externe, le code d'auto-réparation est séparé du code de l'application [80].

Pour l'auto-réparation *interne*, l'insertion d'un code ou d'une nouvelle stratégie dans un composant logiciel (généralement perçu sous forme de boîte noire) est difficile et peut générer des problèmes d'incohérence. En plus, on doit avoir un minimum d'informations sur la conception du composant afin de régir son comportement. Néanmoins, l'auto-réparation *interne* reste généralement plus rapide en terme de temps, et nécessite moins de ressources que l'externe. Les projets *Unity* [23] et *AC-MEN* [90] présentent un prototype de système autonome basé sur des stratégies d'auto-réparation internes.

L'auto-réparation *externe* est appropriée quand il est difficile de fusionner les politiques de reconfiguration avec le code de l'application. Les mécanismes d'auto-réparation internes sont réutilisables et mis à jour facilement tant qu'ils sont localisés [37]. La diversité des sources de provenance des entités logicielles composant l'application, favorise les mécanismes externes. Cette technique partage la tâche d'implantation de l'application entre le développeur et le gestionnaire de l'auto-réparation. Les approches *Kinesthetics eXtreme* [104] et *Rainbow* [21] offrent des techniques d'auto-réparation externes.

1.4.2 Niveau de gestion de la QdS : Instance vs Classe

La QdS peut être gérée au niveau *instance*, c'est à dire traiter la QdS en relation avec la requête ou l'instance du processus en cours d'exécution. Généralement, la gestion de la QdS au niveau *instance* correspond à la gestion des propriétés fonctionnelles. En d'autres termes, on traite tout ce qui est en relation avec la logique métier du service. Les opérations de réparation proposées sont du type : *Redo-Retry*, *Compensate*,

Resume, *Adjust*, *Inform*, *Skip* et *ValChange* [7].

La QdS peut être aussi gérée au niveau *classe*. Dans ce cas, on s'intéresse aux QdS en relation avec toutes les requêtes servies. Par exemple, un monitoring niveau *classe* est équivalent à un monitoring des requêtes de tous les clients qui invoquent ce service. On traite, ici, les propriétés non-fonctionnelles en relation avec la QdS et le contexte en général, telles que les QdS liées au temps, à la réputation et au prix. Les actions de réparations proposées sont du type : *Substitution* et *Duplication* [70].

L'approche présentée dans [12] traite le monitoring des services web sur les deux niveaux : instance et classe. Il présente un nouveau langage d'assertion qui permet d'assurer le monitoring soit d'une instance du processus BPEL, soit de l'ensemble des informations agrégées concernant toutes les instances du processus BPEL.

1.4.3 Service cible par la gestion de la QdS : Sans-état vs Avec-état

Un service sans état est un service qui ne possède aucun état entre différents appels de ses méthodes. Un service web ne doit pas maintenir un état d'invocation avec un client (qu'il soit client final, ou un autre service web) [35, 73]. Si une interaction nécessite un état, alors toutes les informations en relation avec l'état doivent faire partie du message échangé [36]. De cette façon le service est plus fiable tout en simplifiant la procédure de sa réparation en cas de défaillance. Aussi, la scalabilité d'un tel service est approuvée vu que le service ne nécessite pas un état persistant et par la suite, il consomme moins de ressource mémoire. La réparation est plus facile, car on n'a pas à gérer (1) les données persistantes au niveau du service, (2) les dépendances de ce service avec d'autres services, (3) les sessions qui sont en cours d'exécution entre ce service et ses clients.

Un service avec état contient un état de conversation retenu et sauvegardé entre les appels de méthodes et les transactions. La gestion de l'état se fait à travers des informations contenues dans les messages échangés. Les messages sont étendus par des données qui gèrent l'état [60].

Durant une opération de reconfiguration, aucun problème n'est détectable avec les services sans états. Pour un service avec état, si les données de l'état sont enregistrées dans le conteneur du service, une opération de reconfiguration qui ne prend pas en considération l'état du service en panne, et qui dirige les requêtes vers un nouveau service, provoquera des incohérences vu la perte des données enregistrées dans le

conteneur de l'ancien service.

1.4.4 Gestion du monitoring et du diagnostic : Local vs Global

Dans cette partie, nous nous intéressons aux aspects local/global du composant monitoring du processus d'auto-réparation. Dans un scénario local, le monitoring des valeurs de la QdS considère uniquement une paire client/service. Avec une telle vue limitée du système, il ne permet que le monitoring des communications synchrones. Le monitoring global s'intéresse à plusieurs couples client/service. Il permet le monitoring des communications asynchrones et synchrones. Pour une vision globale, le monitoring doit surveiller les paramètres globaux de la QdS. Ces paramètres sont généralement liés à plusieurs requêtes provenant de plusieurs clients. L'analyse est davantage plus sophistiquée dans le cas d'un raisonnement global entraînant plusieurs couples client/service, et surtout en connaissant l'architecture de composition et les interactions entre tous les services.

Cette vue globale du système nous permet l'identification de la source de dégradation ainsi que l'optimisation de l'effort de réparation tout en évitant les actions inutiles de reconfiguration. Une telle situation se produit suite à une propagation de la dégradation de la QdS, telle que la propagation du retard.

1.4.5 Gestion de la QdS : Pronostic vs Diagnostic

En adoptant une politique d'auto-réparation réactive, les services de monitoring coopèrent avec les services d'analyse à détecter et identifier la dégradation de la QdS et à réagir convenablement à travers des actions de réparation. En adoptant une politique d'auto-réparation prédictive, les services de monitoring coopèrent avec les services d'analyse à détecter et identifier la dégradation de la QdS et à réagir convenablement à travers des actions de reconfiguration. Les plans de réparation et de reconfiguration agissent par adaptation d'une composition de services en commutant localement la connexion entre différentes instances du service web. Ils peuvent, également, inclure la connexion à distance à d'autres services web présentant la même interface (WSDL) ou les mêmes fonctionnalités. De tels plans peuvent également inclure l'activation, la désactivation et le déploiement dynamique de nouveaux services web. Ces actions peuvent être déclenchées en tant que réaction à une dégradation de la QdS ou en tant que prévention d'une telle dégradation de se produire. Pour

manipuler une dégradation prédite ou détectée liée à un service web donné, la réparation et les plans de reconfiguration agissent en prévenant -voir même empêchant- les futures requêtes des clients d'utiliser les services concernés par la dégradation. Dans certains cas, la planification doit traiter des situations où plusieurs clients utilisent le même service web ou utilisent différents services web qui partagent des ressources de communication ou de calcul. Les politiques réactives de réparation et les politiques prédictives (préventives) de reconfiguration peuvent reposer sur la redirection des requêtes à une nouvelle instance du service web. Le choix de redirection des clients dépend de leur classification selon différents contrats de souscription. Le groupe de clients souscrit à une faible priorité est pénalisé. Tandis que le groupe des clients souscrit à une haute priorité est privilégié. Selon la source de dégradation, le routage d'une requête vers une nouvelle instance du service web peut être plus approprié que le maintien de la connexion avec l'instance en cours. Pour les applications qui n'ont aucune hiérarchie de privilège des requêtes, la décision repose sur des algorithmes standards d'équilibrage de charge. Dans certains cas, l'analyse des valeurs enregistrées dans le log, participe à améliorer une telle décision. D'autres actions d'adaptation peuvent être planifiées quand la substitution traditionnelle (un à un) n'est pas suffisante ou n'est pas possible, et ce en routant les nouvelles requêtes vers plusieurs instances du même service web, ou vers d'autres services implantant les mêmes fonctionnalités. Les plans de réparation et de reconfiguration agissent par le reroutage des requêtes vers un ou plusieurs services web implantant partiellement ou totalement le WSDL offert par le service dégradé. Par exemple, la requête peut être routée vers un service offrant une interface avec plus d'opérations que le service défaillant. Ceci est appelé substitution. En suivant ce principe, un service peut être remplacé par un ou plusieurs services implantant chacun une partie de l'interface WSDL.

1.4.6 Gestion de la QdS : Orchestration vs Chorégraphie

La majorité des travaux s'intéresse soit à la choréographie [106] soit à l'orchestration [13]. Pour l'orchestration, l'intelligence de la composition est centralisée. En d'autres termes, il existe une entité centrale qui gère les invocations des services et qui les coordonne entre eux. Le service n'a qu'à répondre aux requêtes. Tandis que pour la choréographie, l'intelligence est distribuée sur les services intervenants [78]. En d'autres termes, c'est le service qui prend l'initiative pour satisfaire le besoin de l'invocateur tout en appelant les autres services nécessaires pour compléter la chaîne de composition.

Pour l'orchestration, la gestion de la QdS est généralement centralisée et réalisée au niveau de l'entité centrale de composition. Cependant, pour la chorégraphie, la gestion de la QdS est généralement répartie. Deux applications sont traitées dans ce travail, à savoir : le *FoodShop* qui est une orchestration BPEL de service web et la *Revue Coopérative* qui est une chorégraphie (voir chapitre 4).

1.5 Conclusion

Dans ce chapitre, nous avons présenté une synthèse de l'état de l'art sur les travaux portant sur les différentes étapes du cycle de l'auto-réparation. Nous avons, également, détaillé la problématique confrontée tout au long de la réalisation de ce travail. Le chapitre suivant présentera notre approche de conception d'un système auto-réparable guidé par la QdS.

2

Chapitre 2 : Un Middleware Auto-Réparable guidé par la QdS (*MARQ*)

2.1 Introduction

Dans ce chapitre, nous présentons l'architecture que nous avons définie pour la gestion de l'auto-réparation, partant du monitoring, jusqu'à l'étape de diagnostic/-pronostic. Cette architecture, appelée *MARQ* (**M**iddleware **A**uto-**R**éparable orienté **Q**dS), est décrite par la figure 2.1.

Dans ce qui suit, nous entamons la description des modules intégrés avec *MARQ*. Nous présentons les modèles, les fonctionnalités offertes, ainsi que les détails de l'implantation de chaque module.

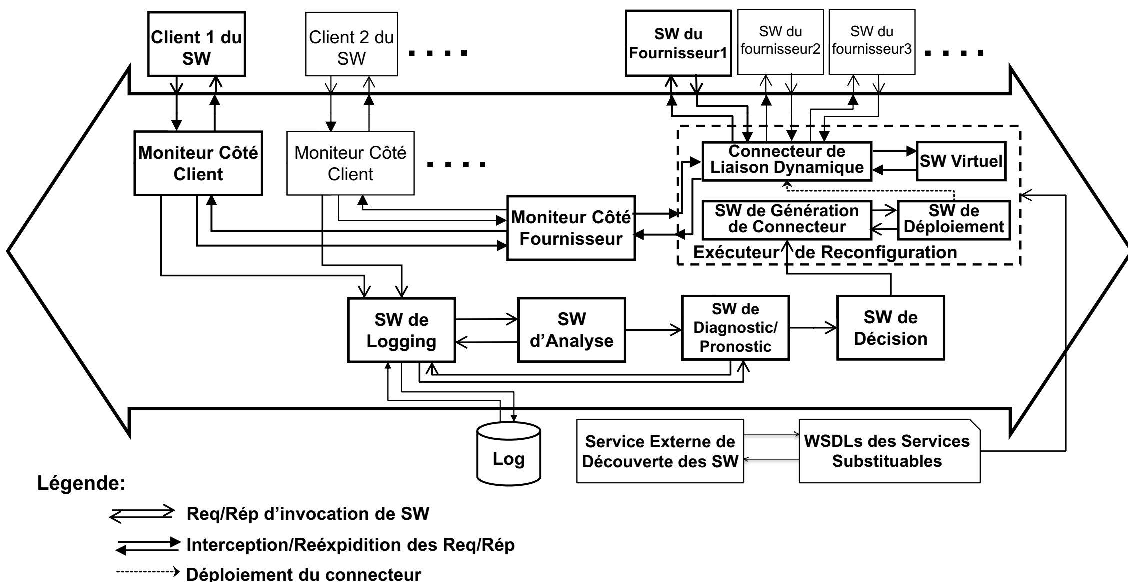

FIG. 2.1 – L'architecture du Middleware Auto-Réparable guidé par la QdS

2.2 Approche Générale

L'objectif principal est de fournir des mécanismes non intrusifs d'auto-réparation afin de fournir une qualité de service meilleure tout au long de l'exécution. Nous définissons pour cela un cadriciel et des dispositifs logiciels couvrant toute la boucle de la gestion d'auto-réparation allant du monitoring de la QdS jusqu'aux actions de reconfiguration. Nous retrouvons, par conséquent, les quatre modules principaux suivants (voir figure 2.1) :

Le monitoring : il correspond à la supervision de l'application. Nous nous focalisons, dans ce cadre, sur l'observation et le stockage des paramètres de QdS obtenus pendant l'exécution.

L'analyse : elle correspond à la phase d'exploitation des valeurs obtenues par le monitoring permettant de s'assurer du bon fonctionnement de l'application, de prédire et de détecter une éventuelle violation de la QdS. Cette détection produit, le cas échéant, des alarmes qui vont enclencher le diagnostic.

Le diagnostic : il est exécuté suite à la détection d'une dégradation de la QdS. L'objectif, ici, est d'identifier et de localiser l'origine de cette dégradation.

La réparation : elle correspond à la reconfiguration de l'application pour rétablir une QdS acceptable pour le système. Les actions qui lui sont associées se basent sur le diagnostic obtenu lors de la phase précédente. Nous considérons

dans notre approche, des réparations comprenant l'exécution d'une séquence d'actions élémentaires de reconfiguration architecturale. La mise en pratique des actions de reconfiguration est réalisée à travers des *Connecteurs de Liaison Dynamique* qui sont générés, compilés et déployés automatiquement après une décision de réparation.

2.3 Le Monitoring

Le module de monitoring intercepte les messages SOAP (requête/réponse), et les étend avec des métadonnées décrivant les paramètres de QdS impliqués et les valeurs mesurées au cours de l'exécution. Ces paramètres nécessitent un traitement du côté du fournisseur (comme le temps d'exécution), ou du côté du client (comme le temps de réponse), ou des deux côtés (comme le temps de communication). Par conséquent, nous déployons un *Moniteur Côté Fournisseur* (*MCF*) pour intercepter tous les messages entrants/sortants du *SW du fournisseur* et un *Moniteur Côté Client* (*MCC*) pour chacun de ses clients. Notre approche cible seulement les interactions du service et n'a pas besoin d'accéder à l'état interne du service, qui est généralement caché par les fournisseurs.

2.3.1 Algorithmes et modèles sous-jacents

Le module de monitoring inclut l'observation et le stockage des valeurs de paramètres de QdS. Il agit au niveau communication. Il intercepte les messages SOAP échangés et les étend par des métadonnées de QdS y incluant les valeurs correspondantes. Il est implanté à travers des *intercepteurs* (voir section 2.3.2.1) qui sont déployés automatiquement représentant le *MCF* et le *MCC*. Le service de *Logging* est implanté comme un service web qui reçoit et enregistre les données dans une base de données dédiée.

Les services web communiquent par échange de messages suivant deux modes de communication, à savoir le synchrone (requête-réponse) et l'asynchrone (requête). Dans le premier mode de communication, l'appelant est bloqué jusqu'au retour du résultat de sa requête. Par contre, dans le deuxième mode de communication, les requêtes et leurs réponses (sous forme de requêtes) sont échangées de façon symétrique sans blocage de l'appelant. Dans ce cas, le calcul de la QdS exploite des données de corrélation, *MessageId* et *RelatedTo*, dans les entêtes des messages SOAP indi-

quant l'association entres les deux messages de type requête. Nous présentons, dans la partie suivante, les algorithmes élaborés pour le monitoring des deux modes de communication.

Dans les algorithmes qui suivent, *SOAPEnvelop* dénote l'enveloppe SOAP contenant une requête ou une réponse, *QoSMetadata* dénote les métadonnées décrivant des paramètres de QdS, et *QoSParamValues* dénote les valeurs des paramètres de QdS.

2.3.1.1 Le monitoring local des communications synchrones

Comme le montre le Tableau 2.1, le *MCF* intercepte les messages SOAP entrants. Il ajoute les métadonnées de QdS (e.g. le temps *t2* d'arrivée de la requête, ligne 2) et transfère la requête pour qu'elle soit exécutée (ligne 3). Le message de réponse est intercepté (ligne 4), étendu par des valeurs de paramètres de QdS (e.g. le temps *t3* d'émission de la réponse, ligne 5) puis envoyé au client.

```
loop
    1   OnRequest( SOAPEnvelop)
        begin
    2       SOAPEnvelop.AddInHeader(QoSMetadata, QoSParamValues)
    3       SOAPEnvelop.release();
        end
    4   OnResponse( SOAPEnvelop)
        begin
    5       SOAPEnvelop. AddInHeader(QoSMetadata, QoSParamValues)
    6       SOAPEnvelop.release()
        end
endloop
```

TAB. 2.1 – Le comportement du *MCF* envers les communications synchrones

Le *MCC* intercepte le message SOAP sortant (ligne 1), comme le décrit le Tableau 2.2. Il ajoute les métadonnées de QdS (e.g. le temps *t1* d'envoi de la requête, ligne 2) et transfère la requête au *MCF* (ligne 3). Le message de réponse est intercepté (ligne 4), étendu par des valeurs de paramètres de QdS (e.g. le temps *t4* de réception de la réponse, ligne 5), et envoyé (ligne 7) après l'extraction de la liste des paramètres de QdS (e.g. *t1*, *t2*, *t3* et *t4*, ligne 6). Par la suite, l'algorithme calcule les valeurs des paramètres de QdS (à savoir, le *temps d'exécution* : *t3 - t2* et le *temps de réponse* : *t4 - t1*, ligne 8) et les stocke (ligne 9). Nous n'avons pas besoin de synchroniser les horloges du client et du fournisseur, parce que le *temps de réponse* (respectivement le *temps d'exécution*) représente la différence entre deux

```
loop
    1   OnRequest( SOAPEnvelop)
        begin
    2       SOAPEnvelop.AddInHeader(QoSMetadata, QoSParamValues)
    3       SOAPEnvelop.release();
        end
    4   OnResponse( SOAPEnvelop)
        begin
    5       SOAPEnvelop. AddInHeader (QoSMetadata, QoSParamValues)
    6       <Li>=ExtractQoSParamValues(SOAPEnvelop)
    7       SOAPEnvelop.release()
    8       <QoSPi>=ComputeQoSValues(<Li>)
    9       Log(<QoSPi>)
        end
endloop
```

TAB. 2.2 – Le comportement du *MCC* envers les communications synchrones

valeurs mesurées auprès du site client (respectivement site fournisseur), en utilisant la même horloge.

2.3.1.2 Le monitoring global des communications asynchrones

Le monitoring est plus complexe pour les communications asynchrones (appelées également *one way message*). Tous les messages sont envoyés via des requêtes, et les réponses éventuelles sont exprimées également sous forme de requête. Nous recourons à la spécification de *WS-Addressing* [102] pour identifier et relier les requêtes lors des communications asynchrones, grâce aux messages d'entête *MessageId* et *RelatesTo*.

```
loop
    1   OnRequest( SOAPEnvelop)
        begin
    2       SOAPEnvelop.AddInHeader(QoSMetadata, QoSParamValues)
    3       SOAPEnvelop.release()
        end
endloop
```

TAB. 2.3 – Le comportement du *MCC* envers les communications asynchrones

Dans le Tableau 2.3, le *MCC* intercepte les messages SOAP sortants (ligne 1), ajoute les métadonnées de QdS (e.g. le temps *t1* de départ de la requête, ligne 2) et transfère la requête au *MCF* (ligne 3).

Comme l'illustre le tableau 2.4, le *MCF* intercepte la requête asynchrone (ligne 1). Il

```
loop
    1   OnRequest( SOAPEnvelop)
        begin
    2       SOAPEnvelop.AddInHeader(QoSMetadata, QoSParamValues)
    3       <Li>=ExtractQoSParamValues(SOAPEnvelop)
    4       <WS-Adrs>= ExtractWSAddressingData(SOAPEnvelop)
    5       SOAPEnvelop.release()
    6       if ("RelatesTo" ∉ <WS-Adrs>)   /*Handle as a request*/
    7           Local_Log (<Li>, "MessageId")
    8       else  /*Handle as a response*/
    9           <Li.new>= Find_in_Local_Log_MessageId_EqualTo("RelatesTo")
    10          <QoSPi>=ComputeQoSValues(<Li>,<Li.new>)
    11          Log(<QoSPi>)
    12      Endif
        End
endloop
```

Tab. 2.4 – Le comportement du *MCF* envers les communications asynchrones

ajoute les métadonnées de QdS (e.g. le temps *t2* de l'arrivée de la requête, ligne 2) et envoie la requête (ligne 5). Suite à l'extraction de la liste des paramètres de QdS, (e.g. *t1* et *t2*, ligne 3), l'algorithme extrait les données concernant le *WS-Addressing* (ligne 4). Si ces données ne contiennent pas le message d'entête *RelatesTo* (ligne 6), il s'agit d'une requête. Dans ce premier cas, les valeurs des paramètres de QdS sont stockées dans un log local (ligne 7). Autrement, la requête est traitée comme réponse (e.g. *t1* représente *t3*, et *t2* représente *t4*). Dans ce deuxième cas, on cherche les valeurs des paramètres de QdS en relation dans le log local de façon que le champ *RelatesTo* de la requête courante soit égal au champ *MessageId* d'une requête précédente (ligne 9). Puis, on calcule les valeurs de QdS (à savoir, le *temps exécution* : *t3* - *t2* et le *temps de réponse* : *t4* - *t1*, ligne 10) et on les stocke au log (ligne 11).

2.3.2 Fonctionnalités et architecture de mise en œuvre

L'intercepteur est le mécanisme de base du monitoring des systèmes distribués. Il est proposé pour différentes technologies de systèmes distribués.

- Dans CORBA, les intercepteurs sont utilisés afin de changer et de contrôler le comportement de l'application [72]. Deux types d'intercepteur sont proposés [75]. Le premier intercepteur est *niveau requête*. Il réagit à la requête (ou réponse) par des pré- ou post-actions. Son comportement ressemble au comportement de celui utilisé par l'orienté aspect. Le deuxième intercepteur est *niveau message*. Il prend en paramètre le message issu de la transformation de la requête par l'ORB. Son comportement se rapproche du comportement de celui appliqué à l'orienté service.
- L'approche orientée *aspect* utilise le terme *Composition Filter* [1] pour exprimer

des fonctionnalités d'interception. Ce dernier permet de spécifier des points de jonction dans le programme qui représente les moments de déclenchement de l'*aspect*.

– Dans l'approche orientée service, les termes *Handler* [93] et *Intermediary* [97] sont utilisés pour référer aux intercepteurs. Ils sont une sorte d'hameçon attaché aux messages échangés entre un client et un serveur. Ils permettent la supervision et la modification des requêtes et des réponses. Afin de mettre en œuvre nos moniteurs, nous avons utilisé et étendu des *Handlers* offerts par les APIs d'*Axis* de la version 3.1.

Handler, *Composition-Filter*, *Intermediary*, et *Intercepteur* sont tous utilisés pour exprimer le même sens : une entité qui assure le contrôle et la supervision du flux échangé entre les composants d'un système.

2.3.2.1 L'interception niveau SOAP

Dans l'approche orientée service, l'intercepteur agit au niveau SOAP. Il intercepte le message SOAP au niveau du conteneur de déploiement, du côté serveur et au niveau de la souche cliente, du côté client. Il intercepte le message, extrait l'enveloppe SOAP et la manipule comme une arborescence XML.

Limite de l'API d'Axis pour les communications synchrones et la solution apportée

Lors de la réception de la requête, le conteneur de service web (Axis) récupère les données fonctionnelles du corps du message SOAP, qui sont nécessaires pour l'exécution du service et ignore les informations stockées dans l'entête SOAP. La réponse préparée pour cette requête contient un nouveau entête vide. Toutes les informations stockées dans l'entête SOAP sont perdues. De ce fait, il en résulte une perte des paramètres de QdS -en relation avec la requête- du côté fournisseur.

Pour remédier à cette limite, nous avons procédé à une extension de l'API *axis.jar*, et plus spécifiquement, la classe *SOAPService.java*, se trouvant sous l'arborescence *org/apache/axis/handlers/soap*. Dans cette dernière, nous avons ajouté le code qui garde une copie de l'entête SOAP de la requête, et qui l'ajoute à l'entête SOAP de la réponse[1].

Limite de l'interception niveau SOAP

Comme le montre la figure 2.2, le connecteur responsable du routage (pour la recon-

[1] La nouvelle API est disponible à l'adresse suivante : http ://www.laas.fr/~rbenhali/Axis-modifie/axis.jar

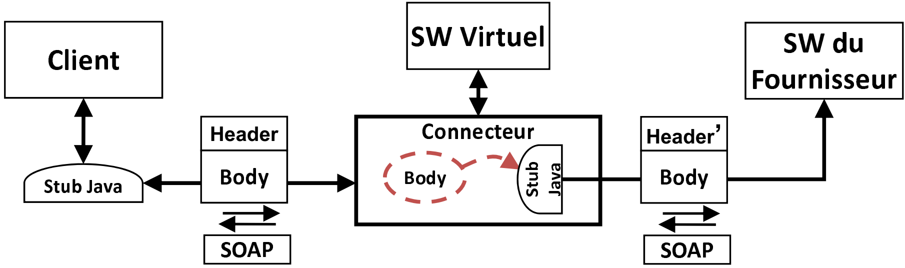

Fig. 2.2 – Le mode de fonctionnement de l'interception niveau SOAP

figuration) de la requête vers le service web concret (*SW du fournisseur*), extrait les paramètres fonctionnels à partir du message SOAP du client, et crée localement une nouvelle requête (*stub java*) qui est envoyée vers le *SW du fournisseur*. Le nouveau message créé contient un nouvel entête SOAP tout en omettant l'entête envoyé par le client. Le transfert des données de l'entête vers le nouveau message SOAP est une tâche techniquement difficile, vu que les données de l'entête ne peuvent être manipulées qu'à travers un intercepteur, ce qui n'est pas le cas pour le nouveau message créé. Ce dernier contient un nouvel entête SOAP (**Header'**) comme le montre la figure 2.2. Il en résulte une perte d'information dans le nouveau message créé. Ceci limite la gestion des communications asynchrones, dont le monitoring repose sur des informations portées par l'entête SOAP.

2.3.2.2 L'interception niveau HTTP

Pour pallier les limites de l'interception niveau SOAP, nous avons développé une nouvelle version permettant la sauvegarde des informations échangées dans l'entête SOAP, et supportant, ainsi, la gestion du monitoring global.

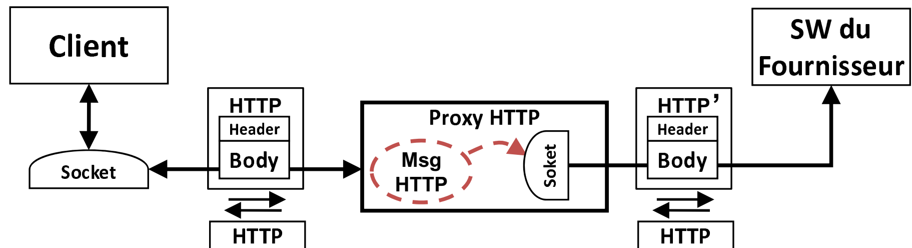

Fig. 2.3 – Le mode de fonctionnement de l'interception niveau HTTP

L'interception niveau HTTP, se base sur la programmation des sockets. Le message SOAP est encapsulé dans un entête HTTP. L'extension du message par des valeurs

de paramètres de QdS, est réalisée en parcourant le message -manipulation de chaîne de caractères- et en l'étendant par des balises XML au niveau de l'entête SOAP ou des données textuelles au début de l'enveloppe HTTP.

Le niveau HTTP (pour la reconfiguration) agit sur l'enveloppe HTTP, en laissant intact le contenu du message SOAP encapsulé. La figure 2.3 montre la création d'une nouvelle enveloppe HTTP (**HTTP'**), tandis que l'entête SOAP reste inchangé. Ceci permet le monitoring global qui se base en partie sur ces informations.

2.4 L'Analyse

Le module d'analyse exploite les valeurs des paramètres de QdS pour s'assurer du bon fonctionnement de l'application. Il permet de détecter les éventuelles violations de la QdS, et produit, le cas échéant, des alarmes qui enclencheront le diagnostic.

2.4.1 Algorithmes et modèles sous-jacents

Notre approche est prédictive. Elle est basée sur l'observation, en temps réel, de l'évolution des valeurs de QdS afin de détecter des violations considérées comme symptômes d'une dégradation imminente. Dans le processus de détection, nous ciblons les situations où les valeurs de QdS mesurées dépassent continuellement le seuil des valeurs acceptables (seuil absolu ou relatif) plus que les discordances transitoires. Une telle situation correspond à une dégradation de QdS, qui est un indicateur de détérioration de l'état du système. Afin de prendre en considération un comportement de référence, nous utilisons d'une part, des indicateurs statistiques pré-calculés et/ou calculés en temps réel au cours de l'exécution. Les événements considérés peuvent correspondre, par exemple, à détecter qu'une valeur de QdS mesurée dépasse un seuil donné. D'autre part, nous utilisons les chroniques temporelles afin de surveiller l'évolution de ces valeurs de QdS pour éviter de considérer les violations transitoires de QdS. Une chronique temporelle est un ensemble d'événements liés par des contraintes temporelles, et dont l'occurrence dépend du contexte [34]. Les événements temporels correspondent principalement à l'occurrence ou à l'absence d'un événement durant une période de temps.

L'analyse de la QdS a pour but l'évaluation de l'état d'un service donné, et non seulement une interaction spécifique dans une conversation spécifique. La détection de dégradation inclut toutes les interactions entre les clients et les fournisseurs.

Dans notre contexte, le fait d'observer N violations au niveau du temps de réponse, lorsqu'on traite plusieurs requêtes provenant de clients distincts, est considéré comme une dégradation de la QdS, de la même façon que pour N violations au niveau du temps de réponse de requêtes, provenant du même client. N s'exprime en fonction de la disponibilité du service ainsi que la probabilité de sa transition entre les états *Violation* et *OK* (Non-Violation). Il représente le nombre de violations successives avant que la dégradation aurait lieu.

Par exemple, pour chaque valeur de *Tresp* mesurée, l'état de service peut être *TrespV*, correspondant à une valeur de temps de réponse mesurée qui soit supérieure au seuil (la valeur maximale acceptable), où *TrespOK*, correspondant à une valeur mesurée au-dessous de ce seuil. Le seuil des valeurs acceptées de chaque paramètre de QdS est égal à la valeur moyenne (*Moyenne*) et un retard toléré (RT). Ce RT est proportionnel à l'*écart type*, et peut être calculé en se basant sur l'inégalité de *Chebyshev* ([3], pages 80-81) qui s'exprime par l'inéquation suivante :

$$P[Tresp < (Moyenne + k\sigma)] \geq 1 - \frac{1}{k^2} \tag{2.1}$$

Où : σ dénote l'*écart type* et k est un nombre réel strictement positif représentant la constante qui définit la proportion de dépassement.

Ce seuil $(Moyenne + k\sigma)$ peut être pré-calculé, à partir des valeurs de QdS d'une expérience déjà réalisée. Il peut également se calculer en temps réel à partir des valeurs mesurées au cours de l'exécution. Par exemple, pour $k = 3$, la probabilité de non-violation du temps de réponse est supérieure à 0.89.

N est égal au nombre de successions de violation menant à un état d'échec d'invocation. Afin de déterminer une valeur pertinente de N, on utilise l'expression suivante, qui correspond à avoir la probabilité de N violations successives (*TrespV*) inférieure à la probabilité -mesurée expérimentalement- des requêtes échouées $(1 - Disponibilité)$:

$$P[N\ TrespV\ successive] \leq (1 - Disponibilité)$$

En plus, la probabilité de « l'état suivant est *TrespV* », est égale à la probabilité partant d'un état *TrespOk*, multipliée par la probabilité de transition de cet état vers l'état *TrespV* et la probabilité d'être dans un état *TrespV*, multipliée par la probabilité de rester dans cet état pour la prochaine invocation :

$$P[Tous\ état \rightarrow TrespV] =$$
$$P[TrespOK] \times P[TrespOK \rightarrow TrespV] + P[TrespV] \times P[TrespV \rightarrow TrespV]$$

La probabilité d'obtention de N violations successives est égale à la probabilité d'atteindre l'état *TrespV*, partant de n'importe quel état, multipliée par la probabilité de rester dans l'état *TrespV* pour *(N-1)* fois :

$$P[N\ TrespV\ successive] = P[Tous\ état \rightarrow TrespV] \times P[TrespV \rightarrow TrespV]^{N-1}.$$

Quand on remplace la probabilité $P[N\ TrespV\ successive]$ dans la première inégalité, on obtient l'expression suivante fixant la limite inférieure de N :

$$P[Tous\ état \rightarrow TrespV] \times P[TrespV \rightarrow TrespV]^{N-1} \leq (1 - Disponibilité)$$
$\Leftrightarrow N \geq \lambda$, où :

$$\lambda = 1 + \frac{\ln\left(\frac{1-Disponibilité}{P[tous\ état \rightarrow TrespV]}\right)}{\ln(P[TrespV \rightarrow TrespV])}$$

La plus petite valeur que N peut prendre, est égale à la partie entière de $(\lambda + 1)$. Lorsqu'on choisit une valeur de N très supérieure à cette dernière, on augmente la précision dans le pourcentage des dégradations détectées. Cependant, ceci peut empêcher la détection de quelques dégradations.

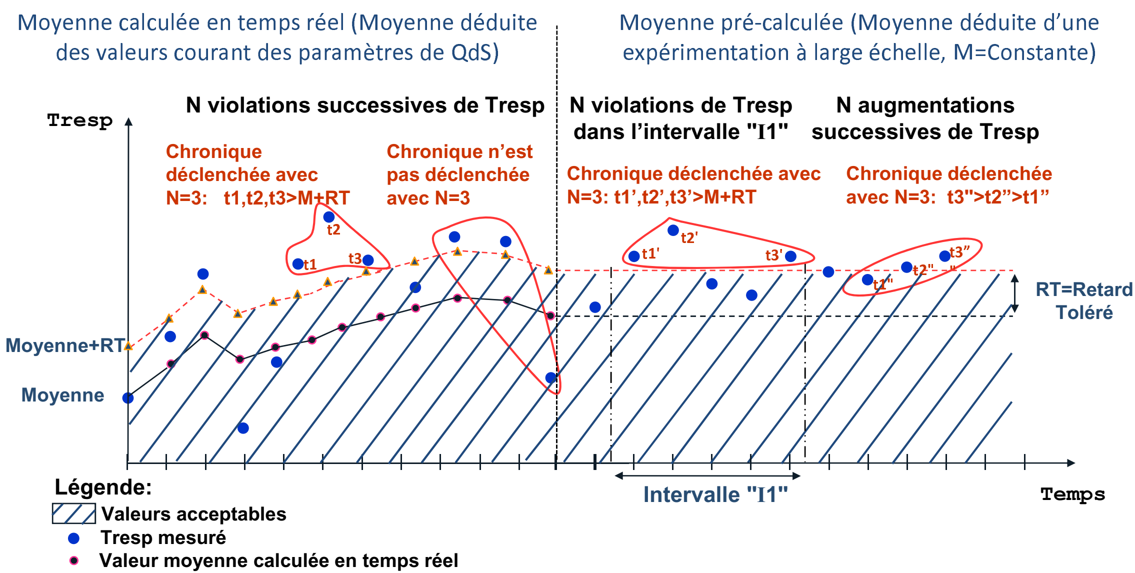

FIG. 2.4 – Un modèle de chronique de détection de dégradation de QdS

La figure 2.4 présente des exemples de chroniques temporelles impliquant le *temps de réponse*. Ces chroniques montrent deux façons pour calculer le seuil de la valeur maximale acceptable : soit pré-calculé, soit calculé à la base des valeurs mesurées en temps réel. Ces chroniques examinent l'évolution des événements *TrespOK* et *TrespV*. Elles peuvent concerner le comportement de la QdS durant un intervalle fixe. La première chronique est déclenchée suite à l'observation de trois occurrences successives de l'événement *TrespV* (*t1*, *t2* et *t3* dans la figure 2.4) sans aucune occurrence de l'événement *TrespOK*. La deuxième chronique est détectée suite à l'observation, dans l'intervalle de temps *I1*, de trois occurrences de l'événement *TrespV* (*t1'*,*t2'* et *t3'* dans la figure 2.4). La troisième chronique est déclenchée suite à l'observation de trois augmentations successives du *temps de réponse* (*t1"*, *t2"* et *t3"* dans la figure 2.4).

La chronique basée sur un seuil des valeurs acceptables calculé au cours de l'exécution, semble ne pas être appropriée au cas d'une augmentation progressive du *Tresp*. Cette augmentation affecte le seuil, et l'accroît progressivement tout en dépassant le niveau d'agrément. D'autre part, la chronique basée sur le pré-calcul du seuil, dans un environnement dédié, ne prend pas en considération le contexte de l'exécution de l'application et les valeurs mesurées du *Tresp* seront insatisfaisantes. Toutefois, la combinaison des deux chroniques est plus solide grâce aux valeurs mesurées au cours de l'exécution. De plus, elle est plus robuste pour faire éviter le cas de l'augmentation progressive grâce à des valeurs déjà mesurées par une expérimentation à grande échelle.

```
1   begin
2       Moyenne= Calcule_Moyenne() // Constante
3       RT= Calcule_Retard_Tolere()
4       Idle: Nbr_Violation_Recue=0
5       loop
6               Pour_chaque_Nouv_Tresp_Mesure()
7                 if (Tresp> Moyenne+RT) then
8                     Nbr_Violation_Recue++;
9                 else goto: Idle
10                endif
11                if (Nbr_Violation_Recue≥ Nbr_Max_Violation_Succ)  then
12                    Notifie_Degradation_Au_Service_De_Diagnostic()
13                endif
15      endloop
16  end
```

TAB. 2.5 – L'algorithme de détection (Moyenne pré-calculée).

Nous avons décrit ces chroniques par des algorithmes de détection de dégradation,

qui déclenchent des alarmes en cas de succession de violations N fois (représenté par *Nbr_ Max_ Violation_ Succ* dans le tableau 2.5 et le tableau 2.6).

Comme c'est illustré dans le tableau 2.5, pour chaque paramètre de QdS mesuré (ligne 6), l'algorithme de détection de dégradation du temps de réponse augmente le nombre des violations reçues (*Nbr_ Violation_ Recue*, à la ligne 8) si la nouvelle valeur dépasse le seuil de la valeur maximale acceptable (ligne 7). Dans le cas de réception d'une non-violation (ligne 9), l'exécution saute vers l'initialisation située à la ligne 4. Autrement, le nombre des violations soulevées sera augmenté jusqu'au déclenchement d'alarmes (ligne 12). Dans cet algorithme, la moyenne (*Moyenne*) et le retard toléré (RT) sont pré-calculés (lignes 2 et 3) et déduits d'une expérimentation faite à grande échelle (voir section 4.2.3.4).

```
1   begin
2       Idle: Nbr_Violation_Recue=0
3       loop
4               Pour_chaque_Nouv_Tresp_Mesure()
5               Moyenne= Calcule_Moyenne() // Variable
6               RT= Calcule_Retard_Tolere()
7                 if (Tresp> Moyenne+RT) then
8                     Nbr_Violation_Recue++;
9                 else goto: Idle
10                endif
11                if (Nbr_Violation_Recue≥ Nbr_Max_Violation_Succ)  then
12                    Notifie_Degradation_Au_Service_De_Diagnostic()
13                endif
14      endloop
15  end
```

TAB. 2.6 – L'algorithme de détection (Moyenne calculée au cours de l'exécution).

Dans l'algorithme de détection de dégradation du temps de réponse présenté au tableau 2.6, la moyenne (*Moyenne*) et le retard toléré (RT) sont calculés en temps réel (lignes 5 et 6), et mis à jour au cours de l'exécution (ligne 4). Une non-violation réinitialise l'exécution (ligne 9). En cas de dégradation, des alarmes sont envoyées vers le service de diagnostic.

2.4.2 Fonctionnalités et architecture de mise en œuvre

L'analyse vise à évaluer l'état d'un service en observant ses interactions avec ses clients. Ainsi, la détection de dégradation concerne les interactions entre un service fournisseur et ses clients. La gestion des violations provenant d'un même client est

la même que celle provenant de plusieurs clients distincts.

L'analyse est implantée en tant que service web. Deux politiques sont utilisées pour entamer un processus d'analyse : la première repose sur une interrogation périodique du log, et la seconde repose sur des notifications reçues –du service web de logging– à propos de la disponibilité de nouvelles valeurs de QdS. Durant le processus d'analyse, des alarmes sont déclenchées en cas de détection dégradation.

2.5 Le Diagnostic/Pronostic et la Décision

Selon la politique adoptée, deux modèles peuvent être utilisés : *Diagnostic* et *Pronostic*. Les algorithmes implantant ces modèles inspectent, ré-activement ou pro-activement, le comportement du service web à la base des valeurs de paramètres de QdS stockées au log. Ceci permet d'identifier/prédire la dégradation passée/imminente. La décision est basée sur des actions de reconfiguration architecturale.

2.5.1 Algorithmes et modèles sous-jacents

Cette partie traite l'identification de la propagation d'une dégradation ainsi que la prévention avec le modèle *Markovien*.

2.5.1.1 Détection de la propagation de dégradation

La tâche de diagnostic repose sur le monitoring et l'analyse des motifs (*patterns*) de défaillance. En effet, une dégradation possède plusieurs sources, et peut être déclenchée par un ou plusieurs services de l'application. Afin d'implanter un système auto-réparable efficace, nous avons besoin de localiser le service défaillant et de raisonner sur la source de dégradation. Par exemple, la combinaison de différents paramètres de QdS comme le temps de réponse et le temps d'exécution, permet de discriminer entre les défaillances du niveau exécution et celles de celui de la communication. C'est aussi le cas du raisonnement sur les dépendances structurelles qui permet d'identifier les propagations de dégradation et d'éviter les actions de réparation inutiles.

Considérons une paire client/fournisseur pour laquelle une alarme révélant une dégradation du temps de réponse ($Tresp$) est déclenchée, comme le montre le ta-

```
1   begin
2       if (Tresp > TrespMoy + TrespRT) then
3           if (Texec ≤ TexecMoy + TexecRT) then
4               Niveau_Degradation ="Communication"
5           else    Retard_Texec= Texec – (TexecMoy+ TexecRT)
6                   if (Tresp – Retard_Texec ≤ TrespMoy + TrespRT) then
7                       Niveau_Degradation="Execution"
8                   else
9                       Niveau_Degradation="Execution & Communication"
10                  endif
11          endif
12      enif
13  end
```

Tab. 2.7 – L'algorithme de localisation de dégradation

bleau 2.7 à la ligne 3. Cet algorithme discrimine entre les défaillances au niveau de la communication et au niveau de l'exécution. On distingue trois cas :

Dans le premier cas (lignes 3 et 4), la valeur du $Texec$ ne dépasse pas la valeur maximale acceptable (*Moyenne* + *Retard Toléré*). Étant donné que le temps de réponse est composé du temps d'exécution et du temps de communication, on déduit que la dégradation est localisée au niveau de la communication. Dans le second cas (lignes 6 et 7), c'est seulement le $Texec$ qui excède le seuil d'acceptation et son retard ($Retard_{Texec}$) est l'origine du déclenchement de dégradation du $Tresp$. La dégradation provient du niveau d'exécution. Dans le troisième cas (lignes 8 et 9), le temps d'exécution et celui de communication dépassent les valeurs maximales acceptables et la dégradation provient des deux niveaux : exécution et communication.

Après la localisation du niveau de la dégradation, on entame le raisonnement sur sa source. Le diagnostic n'est pas limité à une interaction entre une seule paire client/fournisseur. Il inclut les interactions du service web considéré avec les autres services web de l'application. La vue globale du système nous donne la possibilité d'identifier la source de la dégradation, et d'optimiser les efforts de réparation tout en évitant les actions de reconfiguration inutiles. Telle situation se produit dans le cas de la propagation de la dégradation de la QdS.

Considérons le scénario d'une conversation imbriquée, comme l'illustre la figure 2.5. Le service de diagnostic, lié au service web WS2, détecte que son temps d'exécution dépasse le seuil acceptable. Il est défaillant. Pareil pour le service web WS1, son service de diagnostic détecte une dégradation de son temps d'exécution due au phénomène de propagation. Si on ne détecte pas que la dégradation de WS1 est

une simple propagation de celle de WS2, ça mènera à des actions de reconfiguration inutiles.

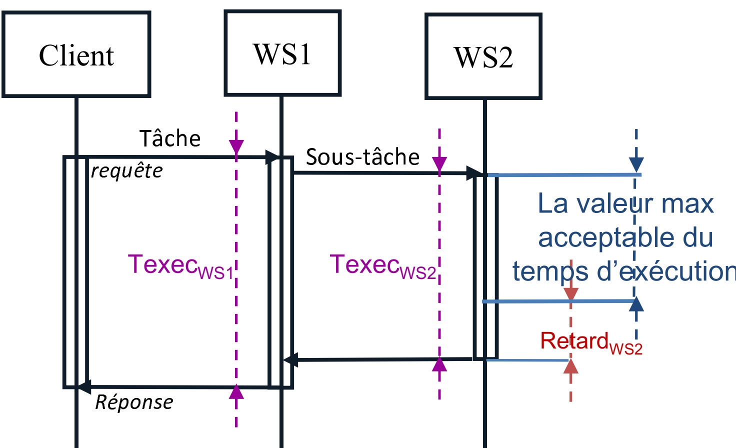

FIG. 2.5 – La propagation de la dégradation de la QdS

La requête du client est exécutée par WS1, qui fait appel à WS2 afin de réaliser une partie de la tâche accordée, comme le montre la figure 2.5. $Texec_{WS1}$ représente le temps d'exécution de la paire Client/WS1 et $Texec_{WS2}$ représente le temps d'exécution de la seconde paire WS1 (comme client)/WS2. WS2 génère un retard important ($Retard_{WS2}$) durant l'exécution de la requête, tout en engendrant un dépassement des valeurs acceptables de $Texec_{WS1}$ et de $Texec_{WS2}$. Les deux services de diagnostic liés à $Texec_{WS1}$ et $Texec_{WS2}$ déclenchent des alarmes.

Dans le cas d'un diagnostic « naïf », deux processus de diagnostic locaux seront envisagés. Le premier concerne le service web WS2. Il compare le temps de réponse et celui de communication avec les seuils d'acception correspondants, et déduit que le problème provient du niveau exécution. Le service de décision choisit de substituer WS2 par un autre service équivalent, qui offre les mêmes fonctionnalités. D'une façon similaire, le diagnostic local lié à WS1 détecte une dégradation de la QdS, et la décision substitue WS1 par un service web équivalent.

Lorsque le diagnostic est établi localement, WS1 et WS2 seront examinés séparément, et chaque processus de diagnostic déclare un verdict local de défaillance :

$$Local_diag(WS2, dégradation) \Rightarrow WS2\ défaillant.$$

et

$$Local_diag(WS1, dégradation) \Rightarrow WS1\ défaillant$$

Lorsque ce verdict de diagnostic est à gérer avec les fonctionnalités de réparation, les actions de reconfiguration doivent substituer chacun des deux services :

$Substituer(WS2, WS2')$, Où WS2' est équivalent à WS2.
et
$Substituer(WS1, WS1')$, Où WS1' est équivalent à WS1.

Une vue globale des dépendances structurelles rend le processus de diagnostic plus précis. Elle permettra d'identifier que WS2 est la source de la dégradation. La dégradation détectée de WS1 est proprement diagnostiquée comme une manifestation de propagation :

$$\begin{aligned}
&Global_diag(WS1, WS2, dégradation, dégradation) \equiv \\
&\quad Local_diag(WS1, dégradation) \wedge Local_diag(WS2, dégradation) \\
&\quad \wedge (Texec_{WS1} - Retard_{WS2} \leq TexecMoy_{WS1} + TexecRT_{WS1}) \\
&\Rightarrow WS2\ défaillant \wedge WS1\ utilise\ un\ service\ web\ défaillant
\end{aligned}$$

Nous notons que la dégradation de WS1 est due à la propagation d'une autre dégradation.

La séquence de reconfiguration correspondante est plus efficace. Elle nécessite seulement la substitution de WS2 :

$Substituer(WS2, WS2')$ Où WS2' est équivalent à WS2.

Différents verdicts de diagnostic peuvent être établis dans le cas d'imbrication d'invocations illustrée par la figure 2.5. Les quatres cas possibles sont comme suit :

1- Premier cas : La réponse du WS2 arrive avec retard, et la réponse de WS1 arrive aussi en retard après l'élimination du retard propagé provenant de WS2.
Les deux services sont défaillants. Dans ce cas, le résultat du diagnostic global est équivalent au résultat des diagnostics locaux de WS1 et de WS2. Les deux services doivent être substitués.

$$\begin{aligned}
&Global_diag(WS1, WS2, dégradation, dégradation) \equiv \\
&\quad Local_diag(WS1, dégradation) \wedge Local_diag(WS2, dégradation) \\
&\quad \wedge (Texec_{WS1} - Retard_{WS2} \geq TexecMoy_{WS1} + TexecRT_{WS1}) \\
&\Rightarrow WS1\ défaillant\ \wedge WS2\ défaillant
\end{aligned}$$

2- Deuxième cas : La réponse de WS2 arrive avec retard, et si on élimine le retard propagé de WS2, la réponse de WS1 ne vient pas en retard.
Les deux services semblent être défaillants, mais WS2 est la source de la dégradation. Le retard généré par ce service ($Retard_{WS2}$) s'est propagé et a affecté WS1. Le diagnostic global identifie la source de la dégradation, et le service de décision demande la substitution de WS2 :

$$\begin{aligned}&Global_diag(WS1, WS2, dégradation, dégradation) \equiv\\&\quad Local_diag(WS1, dégradation) \wedge Local_diag(WS2, dégradation)\\&\quad \wedge(Texec_{WS1} - Retard_{WS2} \leq TexecMoy_{WS1} + TexecRT_{WS1})\\&\Rightarrow WS2\ défaillant \wedge WS1\ utilise\ un\ service\ web\ défaillant\end{aligned}$$

3- Troisième cas : La réponse de WS2 arrive avec retard, et pas de retard pour celle de WS1.
Seul WS2 semble être défaillant, et l'exécution très rapide de WS1 absorbe le retard engendré pas WS2 ($Retard_{WS2}$) :

$$\begin{aligned}&Global_diag(WS1, WS2, \neg dégradation, dégradation) \equiv\\&\quad Local_diag(WS1, \neg dégradation) \wedge Local_diag(WS2, dégradation)\\&\Rightarrow WS2\ défaillant\end{aligned}$$

4- Quatrième cas : La réponse de WS1 arrive avec retard. Quant à celle de WS2, elle arrive à temps. WS1 est le seul service défaillant :

$$\begin{aligned}&Global_diag(WS1, WS2, dégradation, \neg dégradation) \equiv\\&\quad Local_diag(WS1, dégradation) \wedge Local_diag(WS2, \neg dégradation)\\&\Rightarrow WS1\ défaillant\end{aligned}$$

2.5.1.2 Prévention de dégradation : Le modèle Markovien

Les méthodes traditionnelles de détection de dégradation sont généralement limitées à un diagnostic simple (réactif) du comportement du système. Cependant, pour une stratégie préventive, les valeurs des paramètres de QdS doivent être collectées et filtrées au cours du temps afin d'obtenir une vue prédictive et réaliste sur l'évolution de l'état du système. Nous avons implanté un modèle de prédiction d'état du système en utilisant les *Chaînes de Markov Cachées* (CMC).

Les CMC [81] ont pour but l'estimation du prochain état du système, à travers des observations partielles de l'état actuel, et des hypothèses stochastiques. On utilise les CMC pour pronostiquer l'état des applications à base de service web. Les

CMC notifient la dégradation de la QdS afin de décider de l'action de réparation appropriée.

Les CMC suivent un modèle stochastique à temps discret, défini comme un processus de *Markov* avec des états non-observables (cachés). Compte tenu du temps d'exécution (*Texec*) comme paramètres de QdS à pronostiquer, les CMC sont formalisées par le cinq-uplet $< S, A, V, B, \Pi >$, où :

S est l'ensemble des états :

S = {*Working, Partially Working, Not Working*}, ou :

- *Working* (W) : Le service est fonctionnel de façon efficace.
- *Partially Working* (PW) : Le service est fonctionnel, mais il montre quelques régressions de la QdS comparé aux valeurs attendues.
- *Not Working* (NW) : Le service n'est pas fonctionnel, il montre une évolution inacceptable des valeurs de QdS.

A représente la probabilité de transition entre les états. Un exemple de cette matrice est montré dans la figure 2.6. Elle est calculée à la base d'une expérimentation réalisée sur la grille *Grid'5000* (voir chapitre 4). Cette matrice peut être aussi calculée en se basant sur les valeurs de QdS recueillies en temps réel au cours de l'exécution. Mais dans ce cas, il faut attendre le stockage des valeurs de QdS liées à l'exécution d'un nombre suffisant de requêtes, afin qu'elle se stabilise.

A (t \ t + 1)	**W**	**PW**	**NW**
W	**0.7**	**0.2**	**0.1**
PW	**0.3**	**0.5**	**0.2**
NW	**0.1**	**0.2**	**0.7**

FIG. 2.6 – La matrice A de transition

Par exemple, la valeur *0.3* (figure 2.6) représente la probabilité du passage du service d'un état PW à l'instant t à l'état W à l'instant $t+1$.

V est la variable à observer : $V = \{Texec\}$.

B est la distribution de probabilité actuelle d'observer *Texec* à l'état W, PW, et NW à l'instant t : $B_t = \{P(W), P(PW), P(NW)\}$.

Nous utilisons la logique floue pour fixer la distribution d'appartenance de la valeur

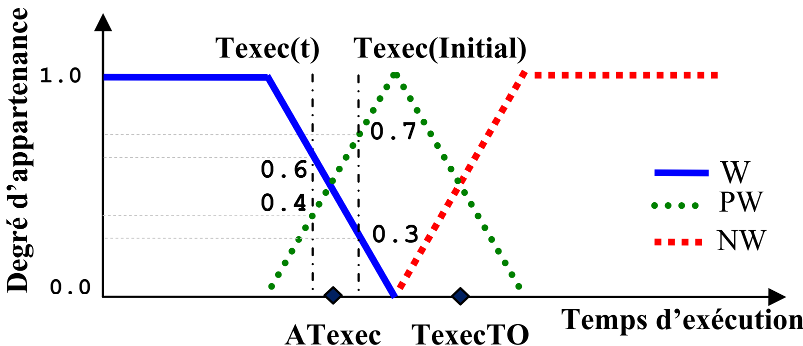

FIG. 2.7 – Le degré d'appartenance de la logique floue

de *Texec*, comme l'illustre la figure 2.7. Nous avons utilisé les algorithmes de Karn [77] et Jacobson [46] pour calculer le *TimeOut* du temps d'exécution (*TexecTO*) ainsi que l'acceptation du temps d'exécution (*ATexec*). Le *TexecTO* représente le temps maximal d'exécution attendu. Il est calculé en utilisant l'équation (2.2).

$$TexecTO_i = STexec_i + K.\sqrt{\sigma_i^2} \tag{2.2}$$

Où : i dénote le numéro de la dernière exécution, K dénote la constante qui définit la proportion de dépassement prématuré du temps limite ; σ_i^2 dénote la variance définie à l'équation (2.4) ; et $STexec$ dénote le temps d'exécution lissé (*Smoothed*) à l'équation (2.5).

Le *ATexec* définit si la valeur du temps d'exécution du service web courant est en pertinence avec son historique. Ainsi, si le *Texec* est supérieur à *ATexec*, alors le service web est considéré se diriger vers l'état de dégradation. Il est calculé à l'aide de l'équation (2.3).

$$ATexec = (STexec_i + TexecTO_i)/2 \tag{2.3}$$

$$\sigma_i^2 = (1-\alpha)\sigma_{i-1}^2 + \alpha.(STexec_i - Texec_i)^2 \tag{2.4}$$

Où : α correspond à ce qui a été décrit dans l'équation (2.6).

$$STexec_i = (1-\alpha).STexec_{i-1} + \alpha.Texec_i \tag{2.5}$$

$$\begin{cases} \alpha = 1/i; & i \leq Q \\ \alpha = 1/Q; & i > Q \end{cases} \quad (2.6)$$

Où : Q dénote une constante qui contrôle la rapidité de *STexec* à s'adapter aux changements.

Avec chaque nouvelle invocation, on recalcule les nouvelles valeurs de *ATexec* et *TexecTO*.

A l'instant t, B_t = *{0.6, 0.4, 0}*.

Π représente la distribution de probabilité initiale des états. Comme le montre la figure 2.7, Π = *{0.3, 0.7, 0}*.

Essayons d'estimer le prochain état du système (à l'instant *t+1*) en partant d'une observation de l'état courant de *Texec*. A l'instant t, la distribution de la probabilité observée est B_t=*{0.6, 0.4, 0}*. Avec les CMC, on utilise B_t et la matrice de transition A afin d'estimer la distribution de probabilité à l'instant *t+1* qui est égale à *{0.54, 0.32, 0.14}*, comme c'est illustré dans la figure 2.8. On prédit que le service réussira sa prochaine invocation.

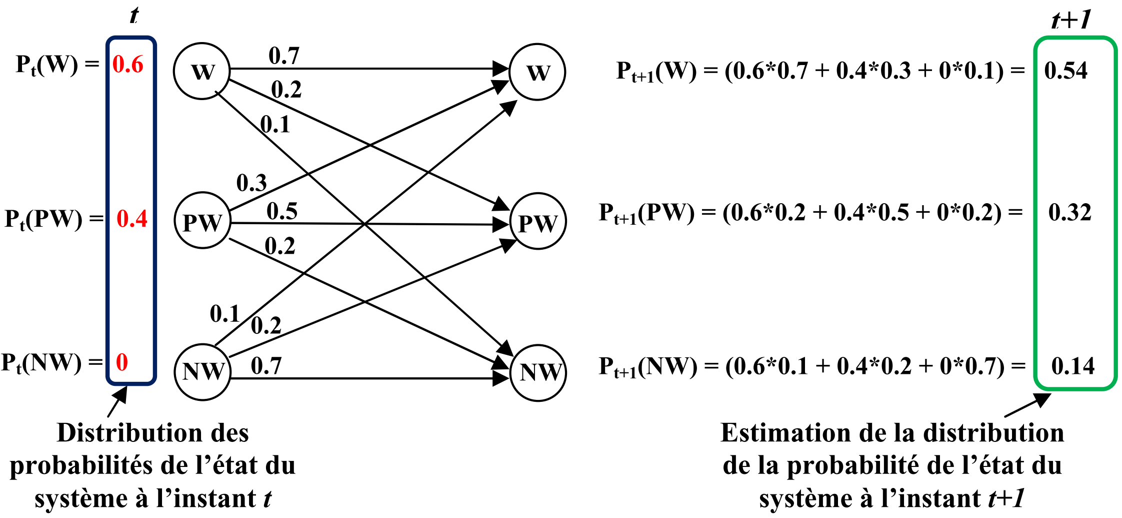

FIG. 2.8 – L'estimation du prochain état du système avec les CMC

2.5.2 Fonctionnalités et architecture de mise en œuvre

Les services de diagnostic et de pronostic raisonnent sur la dégradation tout en se basant sur les valeurs de QdS déjà enregistrées par le monitoring, et identifient la source de la dégradation. Ensuite, ils génèrent un rapport sur l'état du système. Basé sur ce rapport, le service de décision produit des plans et les envoie au module de recouvrement pour demander des réparations (en cas de diagnostic) et des reconfigurations (en cas de pronostic). Chacun de ces trois services est implanté sous forme de service web.

Le rapport de décision, résultant du diagnostic (ou du pronostic) d'une chaîne imbriquée de services web, distingue deux différents cas, à savoir, le raisonnement *local* et le raisonnement *global*. Pour un raisonnement local, nous considérons chaque couple de client/service séparément. La décision s'intéresse à la substitution des fonctionnalités assurées par le service défaillant ainsi que l'ensemble de services qu'il utilise dans le cadre du processus en cours. En d'autres termes, nous substitutions les fonctionnalités offertes par l'ensemble des services connectés dont la façade est représentée par le service défaillant, par un autre service (ou une composition) garantissant ces mêmes fonctionnalités. Pour un raisonnement global, les instances de *MARQ* sont déployées entre tous les couples (client/service), et une action de substitution d'un service défaillant est suivie par un ensemble d'actions de coordination avec les autres instances de *MARQ*. En effet, ces derniers reconnectent le nouveau service utilisé (en tant que client) avec les services utilisés précédemment par le service défaillant. Le raisonnement global informe, aussi, toutes les instances de *MARQ* qui ne sont pas incluses dans ce processus pour ne plus utiliser le service identifié défaillant.

2.6 Conclusion

Dans ce chapitre, nous avons proposé un middleware auto-réparable pour les services web guidé par la QdS : *MARQ*. Ce middleware intervient au niveau communication, entre le client et le fournisseur du service web. Il observe la QdS, l'analyse, et prévoit des actions de réparation en cas de dégradation du service. *MARQ* assure le monitoring des communications synchrones ainsi que celles asynchrones. Son pronostic permet la prévention des défaillances. Il permet, aussi, d'identifier la source d'une dégradation et de reconnaître les phénomènes de propagation. La reconfiguration fera l'objet du chapitre suivant.

3

Chapitre 3 : La Gestion de la Reconfiguration pour l'Auto-Réparation

3.1 Introduction

Notre architecture de gestion de la QdS est implantée sous forme de bus logiciel. Elle constitue un middleware de couplage faible qui implante la reconfiguration selon le principe de la réification du concept de *Connecteur de Liaison Dynamique*. Dans ce chapitre, nous présentons les actions de reconfiguration sous forme d'algorithmes et puis nous les formalisons avec les grammaires de graphes. Ensuite, nous présentons le protocole d'échange entre les composants de *MARQ* ainsi que sa conception réalisée en UML. La dernière partie considère l'intégration de l'auto-réparation de niveau classe (représentée par *MARQ*) avec celle de niveau instance (représentée par *SH-BPEL*).

3.2 La Reconfiguration

Notre approche d'auto-réparation est basée sur la reconfiguration architecturale. Elle substitue un service défaillant soit par un autre équivalent, soit par une composition de plusieurs services, selon l'algorithme décrit aux tableaux 3.1, 3.3 et 3.3. La mise en œuvre de la reconfiguration est garantie à travers la réification d'un *Connecteur de Liaison Dynamique* (*CLD*) qui déconnecte la connexion courante et reroute les requêtes vers les nouveaux services sélectionnés d'une façon complètement transparente par rapport aux clients. Le *Connecteur de Liaison Dynamique* est généré et déployé automatiquement en utilisant la compilation au cours de l'exécution, et la programmation réflexive.

3.2.1 Algorithmes et modèles sous-jacents

Cette partie présente les algorithmes de substitutions.

3.2.1.1 L'algorithme de substitution d'un service

La substitution d'un service web défaillant peut être réalisée par un ou plusieurs autres services. Nous distinguons trois différents cas, comme le montre l'algorithme présenté dans les tableaux 3.1, 3.2 et 3.3.

```
/*L'interface WSDL du service défaillant à substituer*/
WSDL_Interface(S_défaillant) = {Op_1, ... ,Op_N}
/*Parcourir les services disponibles*/
Analyse_Disponible_WS_Interfaces();
/*Les services web considérés pour la substitution*/
S ← {S_1, …, S_M}
(1) Substitution_Simple ← ∃ S_t such that WSDL_Interface(S_t) ⊇ WSDL_Interface(S_défaillant)
 if (Substitution_Simple)
        then      for each (Op_i)-requete
                       Reroute_request_to S_t;
                  endfor
endif
```

TAB. 3.1 – L'algorithme de reconfiguration d'un service (1/3)

Le premier cas traite les substitutions simples. Il est présenté dans le tableau 3.1. Toutes les requêtes sont redirigées vers le nouveau service web, qui offre les mêmes

opérations du service défaillant. Dans un tel cas, $S_{défaillant}$ est entièrement remplacé par S_t.

(2) Substitution_Composée_SansSurcharge $\leftarrow \exists$ *{S_h,.., S_k} tel que:*

$$\left(\left(\bigcup_{i=h}^{k} WSDL_{Interface(S_i)}\right) \supseteq WSDL_{Interface(S_{défaillant})}\right)$$
$$\wedge$$
$$\left(\left(\forall_{x,y}\left(WSDL_Interface(S_x) \cap WSDL_Interface(S_y)\right) \cap WSDL_Interface(S_{défaillant})\right) = \emptyset\right)$$

if *(Substitution_Composée_SansSurcharge)*
then ***for each*** *(Op_i)-requete*
Reroute_requete_vers S_j, tel que $Op_i \in$ WSDL_Interface(S_j)
endfor
endif

TAB. 3.2 – L'algorithme de reconfiguration d'un service (2/3)

Le deuxième cas traite la substitution composée par deux ou plusieurs services, dont l'union couvre les opérations offertes par le service défaillant. Aucune surcharge n'est détectée parmi les opérations offertes par la composition. Il est présenté dans le tableau 3.2. Dans un tel cas, $S_{défaillant}$ est remplacé par un ensemble de services et les requêtes qui lui sont adressées, seront fractionnées sur l'ensemble $S_h, .., S_k$.

Le troisième cas est similaire au deuxième, mais certaines opérations offertes apparaissent plusieurs fois dans les services sélectionnés pour la substitution. Il est présenté dans le tableau 3.3. Deux politiques sont adoptées. Dans la première politique, le service de substitution peut être utilisé par d'autres clients qui contournent notre middleware de gestion de la QdS. Dans ce cas, nous suivons la disponibilité la plus élevée pour rerouter les requêtes des clients. Dans la deuxième politique, on gère des services qui ne sont accessibles qu'à travers notre middleware. Dans ce cas, le partage de la charge équitablement entre les services parait la solution la plus réaliste.

3.2.1.2 L'algorithme de substitution d'une opération

La substitution simple peut être considérée comme une substitution totale de toutes les opérations du service. Toutefois, il arrive des cas ou seulement quelques opérations présentent des défaillances, et la substitution du service entier peut être observée comme étant une perte d'opérations qui fonctionnent parfaitement. Dans ce cas, on procède par des actions de substitution au niveau opération. Les tableaux 3.4, 3.5 et 3.6 montrent un algorithme de substitution plus raffiné que celui présenté

(3) *Substitution_Composée_AvecSurcharge* ← ∃ $\{S_h,.., S_k\}$ *tel que:*

$$\left(\left(\bigcup_{i=h}^{k} WSDL_{Interface(S_i)} \right) \supseteq WSDL_{Interface(S_{défaillant})} \right)$$

$$\wedge$$

$$\left(\begin{array}{c} \exists Op_i \in WSDL_Interface(S_{défaillant}), \\ \exists (x,y) \text{ tel que } Op_i \in \left(WSDL_Interface(S_x) \cap WSDL_Interface(S_y) \right) \end{array} \right)$$

if *(Substitution_Composée_AvecSurcharge)*
then ***for each*** *(Op_i)-requete*
Reroute_requete_vers S_j,
tel que $Op_i \in WSDL_Interface(S_j)$, $\nexists k: Op_i \in WSDL_Interface(S_k)$
and
*/*Gestion des Opérations Surchargées ; Première Politique: Suivre la disponibilité du service*/*
if *(Politique_Service_Disponibilité)*
for each *(Op_i)-requete, tel que* $Op_i \in \{Sj_1, \dots, Sj_a\}$
Reroute_requete_to Sj_k,
tel que Sj_k ← *Disponibilité_élevée(*$\{Sj_1, \dots, Sj_a\}$*,*Op_i*)*
endfor
/ Gestion des Opérations Surchargées ; Deuxième Politique :*/*
/ Equilibrer la charge équitablement entre les services*/*
else
for each *(Op_i)-requete, tel que* $Op_i \in \{Sj_1, \dots, Sj_a\}$
Reroute_requete_to Sj_k
j_{k++}
endfor
endif
endif

TAB. 3.3 – L'algorithme de reconfiguration d'un service (3/3)

dans la section précédente. Il gère la substitution au niveau opération. En effet, un service est composé de plusieurs opérations, et c'est possible de remplacer seulement l'opération défaillante. Nous fractionnons les requêtes des clients à un niveau plus granuleux, à savoir l'opération.

Comme le montre le tableau 3.4, le premier cas substitue une opération par une autre équivalente.

Dans le deuxième (tableau 3.5) et le troisième cas (tableau 3.6), nous notons l'existence de plusieurs opérations équivalentes pouvant remplacer l'opération défaillante.

Dans le tableau 3.5, l'algorithme reroute les requêtes adressées à l'opération défaillante, vers l'ensemble des opérations équivalentes, en partageant la charge équitablement.

L'algorithme du tableau 3.6 reroute les requêtes adressées à l'opération défaillante, vers l'ensemble des opérations équivalentes, en suivant la meilleure disponibilité.

```
/*L'opération défaillante à remplacer*/
Op_défaillante ∈ WSDL_Interface(S)
/*Parcourir les services disponibles*/
Analyse_Disponible_WS_Interfaces();
/*Les services web considérés pour la substitution*/
S ← {S1, ..., SM} tel que ∀i , ∃Op_sub ∈ WSDL_Interface(Si),
                  tel que Op_défaillante ≡ Op_sub, ou  i ∈ {1..M}

/* Substitution d'une opération par une équivalente */
(1) Substitution_Directe ,t ∈ {1..M} et t=constance
if (Substitution_Directe)
        then    if  ((Op_sub)-requete  )
                        then  Reroute_requete_vers St
                endif
endif
```

Tab. 3.4 – L'algorithme de reconfiguration d'une opération (1/3)

```
/* Substitution d'une opération par plusieurs équivalentes en équilibrant les charges */
 (2) Substitution_Equilibre_Charge, i ∈ {1..M}

if (Substitution_Equilibre_Charge)
        then    if ((Op_sub)-requete)
                            Reroute_requete_to Si
                          i++
                endif
endif
```

Tab. 3.5 – L'algorithme de reconfiguration d'une opération (2/3)

```
/* Substitution d'une opération par celle la plus disponible */
(3) Substitution_Disponibilité_Opération, Sj ← Disponibilité_élevée({S1, ..., SM},Op_sub)
if (Substitution_Disponibilité_Opération),
        then    if ((Op_sub)-requete)
                            Reroute_requete_to Sj
                endif
endif
```

Tab. 3.6 – L'algorithme de reconfiguration d'une opération (3/3)

3.2.2 La formalisation de la substitution

Nous utilisons les grammaires de graphes pour la description des opérations de substitution des services web défaillants. Le formalisme adopté a été élaboré dans la thèse de K. Guennoun [42]. Les grammaires de graphes peuvent représenter l'ensemble de toutes les configurations possibles d'une application où les opérations de reconfiguration architecturale sont considérées comme des règles de ré-écriture de graphes. L'une de nos perspectives est d'utiliser cet aspect génératif des grammaires pour caractériser rigoureusement les instances consistantes de l'architecture. Le moteur de transformation de graphes (MTG) [42] peut déterminer si un invariant est préservé par toutes les architectures possibles et peut détecter si un graphe (configuration architecturale) viole ou pas l'invariant. Le MTG permet aussi de détecter l'inconsistance de l'application en se basant sur la différenciation de l'architecture. Ceci repose sur la comparaison de l'architecture modèle avec une reconstruction de l'architecture en cours d'exécution. Dans notre cas, la reconstruction de l'architecture d'interaction entre les services web de l'application est réalisable à travers *L'interface graphique de monitoring et d'analyse de la QdS* de *MARQ* version HTTP (voir section 4.3).

Le formalisme utilise plusieurs types d'étiquettes pour décrire les contraintes et les besoins des instances de l'architecture. Les nœuds terminaux (T) et les nœuds non-terminaux (NT) admettent plusieurs étiquettes et les arcs considèrent plusieurs étiquettes. Une grammaire de graphes est décrite par le quadruplet (G, NT, T, P) (voir table 3.7) où G, NT, T, P représentent respectivement le graphe décrivant l'application, l'ensemble des non-terminaux, l'ensemble des terminaux et l'ensemble des productions de la grammaire. Les nœuds terminaux sont ceux appartenant à un graphe terminal, et les nœuds non-terminaux sont ceux permettant le passage d'un graphe intermédiaire vers un autre graphe intermédiaire. P est de la forme $p[(X_1, ..., X_i), (D_1, ..., D_i)]$ où $X_1, ..., X_i$ sont des étiquettes variables de nœuds appartenant au graphe père ou au graphe fils de la production p tandis que $D_1, ..., D_i$ correspondent respectivement aux domaines de définition de ces variables. L'affectation d'un paramètre X_j de cette production de grammaire par une valeur V_j appartenant au domaine D_j implique l'affectation de toutes les étiquettes des nœuds correspondant à X_j par V_j.

Une instance appartenant à la grammaire de graphes est un graphe contenant des nœuds terminaux obtenus en partant du graphe initial de l'application et en appliquant une séquence de productions de P.

$GG = (G, NT, T, P)$ avec : $T = \{A(a,1,m)\}$, $NT = \{\ \}$ and $P = \{p\}$
$p = (L = \{A(a,1,m1), A(b,2,m2), A(a,1,m1) \xrightarrow{r} A(b,2,m2)\}\,;$ $K = \{A(b,2,m2)\}\,;$ $R = \{A(c,1,m3), A(d,2,m4), A(c,1,m3) \xrightarrow{q} A(d,2,m4)\}\,;$ $C = \{\}$

TAB. 3.7 – Un exemple de grammaire de graphes

P spécifie un ensemble de productions de la grammaire de la forme $(L; K, R; C)$ ou L, R et K sont des sous-graphes et C est un ensemble d'instructions de connexion. Une telle production est considérée applicable sur un graphe donné G s'il existe un homomorphisme de graphes de L à G. Si une production est applicable, il résulte de son application la suppression de l'occurrence du sous-graphe $Del = (L \setminus K)$, l'insertion d'une copie isomorphisme du sous-graphe $Add = (R \setminus K)$ et l'exécution des instructions de connexion déclarées dans C.

C contient l'ensemble des instructions de connexion de la grammaire. Ces instructions sont communes à toutes les productions de grammaire : après chaque application d'une des productions appartenant à P, toutes les instructions de connexion de C, qui sont applicables, seront exécutées [42].

Dans la spécification suivante, on s'interesse à l'application en tant qu'ensemble de couples de client (*Client*) et service web (SW). Si un service web invoque un autre service web, alors dans ce cas, le premier service est traité une fois –par un ensemble de règles– en tant que *Client* et une deuxième fois –par un autre ensemble de règles– en tant que *SW*.

3.2.2.1 La formalisation de la substitution d'un service

Le tableau 3.8 représente la formalisation du premier cas de substitution décrit par le tableau 3.1 : la substitution simple d'un service web par un autre équivalent. L'application de la production p de *GG1* supprime le service défaillant décrit par le sous-graphe *Del*, et ajoute le nouveau service remplaçant décrit par le sous-graphe *Add*.

$GG1 = (G, NT, T, P)$ avec : $T = \{N_1(Id_1, Client), N_2(Id_2, SW, Etat)\}$, $NT = \{\ \}$ and $P = \{p\}$
$p = (L = \{N_1(Id_1, Client), N_2(Id_2, SW, Défaillant),$ $N_1 \rightarrow N_2\}$; $K = \{N_1(Id_1, Client)\}$; $R = \{N_3(Id_3, SW, \neg Défaillant), N_1 \rightarrow N_3\}$;)

Tab. 3.8 – GG1 : La substitution simple d'un service

Le tableau 3.9 représente la grammaire de graphes *GG2* formalisant le deuxième cas de substitution décrit par le tableau 3.2 : la substitution d'un service web par une composition de services sans qu'elle contienne des méthodes surchargées. L'application de la production *p1* de la grammaire *GG2* redirige la requête d'une opération Op_i du service défaillant vers une opération équivalente appartenant à un autre service non défaillant. La production *p1* est appliquée *N-1* fois, (où *N* représente le nombre d'opérations du service défaillant). La dernière opération est traitée par la production *p2* de la grammaire *GG2* qui permet la substitution de la dernière opération et la suppression du service défaillant.

$GG2 = (G, NT, T, P)$ avec : $T = \{N_1(Id_1, Client), N_2(Id_2, SW, Etat)\}$, $NT = \{\ \}$ and $P = \{p1, p2\}$
$p1(Op_i) = (L = \{N_1(Id_1, Client), N_2(Id_2, SW, Défaillant),$ $N_1 \xrightarrow{\{Op_1,\ldots,Op_n\}} N_2\}$; $K = \{N_1(Id_1, Client), N_2(Id_2, SW, Défaillant)$ $N_1 \xrightarrow{\{Op_1,\ldots,Op_n\}\backslash Op_i} N_2\}$; $R = \{N_3(Id_3, SW, \neg Défaillant), N1 \xrightarrow{Op_i} N_3\}$;) $p2(Op_i) = (L = \{N_1(Id_1, Client), N_2(Id_2, SW, Défaillant),$ $N_1 \xrightarrow{\{Op_i\}} N_2\}$; $K = \{N_1(Id_1, Client)\}$; $R = \{N_3(Id_3, SW, \neg Défaillant), N1 \xrightarrow{Op_i} N_3\}$;)

Tab. 3.9 – GG2 : La substitution composée sans surcharge

Les tableaux 3.10 et 3.11 représentent la formalisation du troisième cas de substitution décrit par le tableau 3.3 : La substitution d'un service web par une composition de services qui contiennent des opérations redondantes.

Le tableau 3.10 traite le cas où le choix de l'opération cible repose sur la disponi-

bilité la plus élevée du service. Pour caractériser ce paramètre, nous avons ajouté l'étiquette des arcs d qui représente la disponibilité du service contenant l'opération choisie. Les productions *p1* et *p2* de la grammaire *GG3* choisissent la disponibilité la plus haute afin de router la requête au nouveau service.

$GG3 = (G, NT, T, P)$ avec : $T = \{N_1(Id_1, Client), N_2(Id_2, SW, Etat)\}$, $NT = \{\ \}$ and $P = \{p1, p2\}$
$p1(Op_i) = (L = \{N_1(Id_1, Client), N_2(Id_2, SW, Défaillant), N_1 \xrightarrow{\{Op_1,\ldots,Op_n\}} N_2\}$; $K = \{N_1(Id_1, Client), N_2(Id_2, SW, Défaillant)\ N_1 \xrightarrow{\{Op_1,\ldots,Op_n\}\setminus Op_i} N_2\}$; $R = \{N_3(Id_3, SW, \neg Défaillant), N1 \xrightarrow{Op_i, d} N_3\}$;) $p2(Op_i) = (L = \{N_1(Id_1, Client), N_2(Id_2, SW, Défaillant), N_1 \xrightarrow{\{Op_i\}} N_2\}$; $K = \{N_1(Id_1, Client))\}$; $R = \{N_3(Id_3, SW, \neg Défaillant), N1 \xrightarrow{Op_i, d} N_3\}$;)

TAB. 3.10 – GG3 : La substitution composée selon la disponibilité

Les productions *p1* et *p2* de la grammaire *GG4*, montrées dans le tableau 3.11, traitent le cas de partage des requêtes équitablement entre les services contenant des opérations surchargées. Pour spécifier ce paramètre, nous avons ajouté l'étiquette aux arcs p qui représente le poids de l'opération Op_i. Cette opération est invoquée p fois avant de balancer la charge à une opération équivalente. Si $p=1$, alors la charge est distribuée équitablement entre les opérations redondantes.

3.2.2.2 La formalisation de la substitution d'une opération

Le tableau 3.12 représente la grammaire *GG5* qui formalise le premier cas de substitution décrit par le tableau 3.4 : la substitution simple d'une opération défaillante Op_i d'un service web par une autre équivalente. L'application de la production p de la grammaire *GG5* supprime l'opération défaillante et sa connexion qui sont décrites par le sous-graphe *Del*, et ajoute la nouvelle opération ainsi qu'un nouvel arc la liant au client qui sont décrits par le sous-graphe *Add*.

Le tableau 3.13 représente la formalisation du deuxième cas de substitution décrit par le tableau 3.5 : la substitution d'une opération par une autre possédant la disponibilité la plus élevée parmi les opérations équivalentes. L'application de la

$GG4 = (G, NT, T, P)$ avec : $T = \{N_1(Id_1, Client), N_2(Id_2, SW, Etat)\}$, $NT = \{\ \}$ and $P = \{p1, p2\}$
$p1(Op_i) = (L = \{N_1(Id_1, Client), N_2(Id_2, SW, Défaillant), N_1 \xrightarrow{\{Op_1,\ldots,Op_n\}} N_2\}$; $K = \{N_1(Id_1, Client), N_2(Id_2, SW, Défaillant)\ N_1 \xrightarrow{\{Op_1,\ldots,Op_n\}\backslash Op_i} N_2\}$; $R = \{N_3(Id_3, SW, \neg Défaillant), N1 \xrightarrow{Op_i,p} N_3\}$;)
$p2(Op_i) = (L = \{N_1(Id_1, Client), N_2(Id_2, SW, Défaillant), N_1 \xrightarrow{\{Op_i\}} N_2\}$; $K = \{N_1(Id_1, Client)\}$; $R = \{N_3(Id_3, SW, \neg Défaillant), N1 \xrightarrow{Op_i,p} N_3\}$;)

TAB. 3.11 – GG4 : La substitution composée avec balance de charge

$GG5 = (G, NT, T, P)$ avec : $T = \{N_1(Id_1, Client), N_2(Id_2, SW, Etat)\}$, $NT = \{\ \}$ and $P = \{p\}$
$p(Op_i) = (L = \{N_1(Id_1, Client), N_2(Id_2, SW, Défaillant), N_1 \xrightarrow{\{Op_1,\ldots,Op_n\}} N_2\}$; $K = \{N_1(Id_1, Client), N_2(Id_2, SW, Défaillant)\ N_1 \xrightarrow{\{Op_1,\ldots,Op_n\}\backslash Op_i} N_2\}$; $R = \{N_3(Id_3, SW, \neg Défaillant), N1 \xrightarrow{Op_i} N_3\}$;)

TAB. 3.12 – GG5 : La substitution simple d'une opération

production p de la grammaire *GG6* redirige la requête vers une opération Op_i non défaillante dont la disponibilité d est la plus élevée.

$GG6 = (G, NT, T, P)$ avec : $T = \{N_1(Id_1, Client), N_2(Id_2, SW, Etat)\}$, $NT = \{\ \}$ and $P = \{p\}$
$p(Op_i) = (L = \{N_1(Id_1, Client), N_2(Id_2, SW, Défaillant), N_1 \xrightarrow{\{Op_1,\ldots,Op_n\}} N_2\}$; $K = \{N_1(Id_1, Client), N_2(Id_2, SW, Défaillant)\ N_1 \xrightarrow{\{Op_1,\ldots,Op_n\}\backslash Op_i} N_2\}$; $R = \{N_3(Id_3, SW, \neg Défaillant), N1 \xrightarrow{Op_i,d} N_3\}$;)

TAB. 3.13 – GG6 : La substitution selon la disponibilité de l'opération

La grammaire *GG7*, montrée par le tableau 3.14, traite le cas de partage des requêtes entre les opérations équivalentes disponibles. Ce cas est décrit par le tableau 3.6. Pour spécifier la disponibilité, nous avons ajouté l'étiquette a aux arcs qui représente le poids de l'opération Op_i. Chaque opération est invoquée a fois avant de balancer la charge à une opération équivalente. Si a=*1*, alors la charge est distribuée équitablement entre les opérations surchargées.

$GG7 = (G, NT, T, P)$ avec : $T = \{N_1(Id_1, Client), N_2(Id_2, SW, Etat)\}$, $NT = \{\ \}$ and $P = \{p\}$
$p(Op_i) = (L = \{N_1(Id_1, Client), N_2(Id_2, SW, Défaillant), N_1 \xrightarrow{\{Op_1,...,Op_n\}} N_2\}$; $K = \{N_1(Id_1, Client), N_2(Id_2, SW, Défaillant)$ $N_1 \xrightarrow{\{Op_1,...,Op_n\}\backslash Op_i} N_2\}$; $R = \{N_3(Id_3, SW, \neg Défaillant), N1 \xrightarrow{Op_i,a} N_3\}$;)

TAB. 3.14 – GG7 : La substitution avec balance de charge entre les opérations

3.2.3 Fonctionnalités et architecture de mise en œuvre

Les fonctionnalités de reconfiguration de notre middleware, sont implantées par le module *Exécuteur de Reconfiguration*. Dans ce qui suit, nous détaillons son fonctionnement interne.

3.2.3.1 La reconfiguration par *Connecteur de Liaison Dynamique*

Le module *Exécuteur de Reconfiguration* est en charge de : 1) offrir l'interface du service web virtuel (*SW Virtuel*) et exécuter le service web du fournisseur (*SW du fournisseur*), 2) mettre en œuvre la reconfiguration suivant le plan décrit par le service de décision (*SW de Décision*). Le service virtuel est un service *blanc* qui affiche la même interface que celle du *SW du fournisseur*, sans logique métier (le corps est vide). Du point de vue du client, –avec abstraction faite– les requêtes sont adressées au fournisseur à travers notre middleware et -avec réification faite- c'est au *SW Virtuel* que ces requêtes sont adressées. Le *Connecteur de Liaison Dynamique* les intercepte, et duplique les paramètres fonctionnels pour le *SW du fournisseur*. Il intercepte aussi la réponse, et remplace celle du *SW Virtuel* par celle reçue de la

part du *SW du fournisseur*. De cette façon, l'interaction est assurée entre client et fournisseur de service.

La description du processus de génération du *Connecteur de Liaison Dynamique* est illustrée par la figure 3.1. Le tableau 3.15 décrit les étapes du processus de génération.

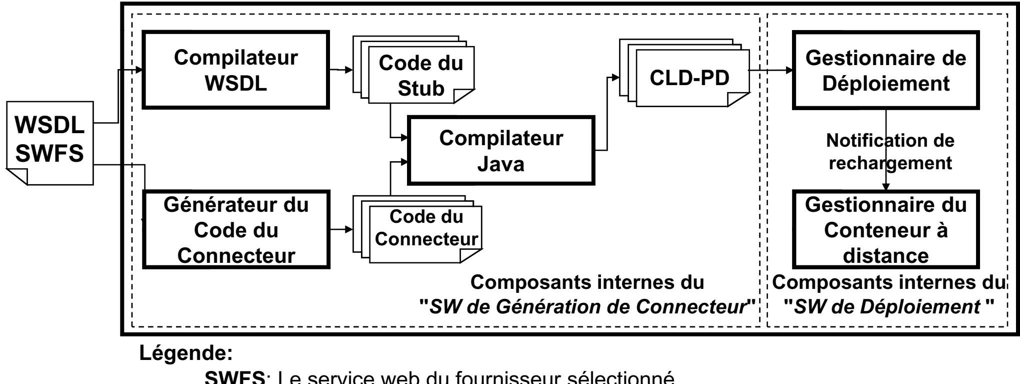

FIG. 3.1 – Les étapes principales du processus de génération automatique du *CLD*

Dans la figure 3.1, le *SW de Génération de Connecteur* 1) reçoit le plan de réparation du *SW de Décision*, 2) génère automatiquement le *stub* du nouveau *SW du fournisseur* sélectionné en utilisant son WSDL, 3) génère automatiquement le code du *CLD* et le compile, et 4) re-déploie le nouveau *CLD* en utilisant le *SW de Déploiement*. Le compilateur WSDL initial, offert par Axis, ne permet des compilations que sur la ligne de commande. Nous étions amenés à modifier son API pour lever la protection, et l'intégrer dans ce processus. La classe modifiée est *WSDL2Java.java*, se trouvant dans le jar *axis.jar* sous l'arborescence *org/apache/axis/wsdl*. Nous avons utilisé *Tomcat* (version 5.5) comme serveur web, *Axis* (version 1.4) comme conteneur de services web, et *Java* (version 1.5) comme langage de programmation.

Le fonctionnement interne du *Connecteur de Liaison Dynamique* est divisé en deux parties dont chacune est représentée par une méthode comme le montre la figure 3.2. La première, intitulée *Init*, est exécutée une seule fois. Elle permet d'initialiser le nouveau connecteur généré. Elle crée le *stub*, et analyse la description WSDL (avec l'API Jdom[1]) du nouveau *SW du fournisseur*, afin d'extraire les types des paramètres fonctionnels, qui seront nécessaires plus tard pour l'invocation réflexive de celui-ci. La deuxième méthode, intitulée `Bind`, est exécutée une fois par chaque requête. Elle analyse le message SOAP envoyé au *SW Virtuel* (en utilisant les APIs d'*Axis* et

[1]http ://www.jdom.org/ : *parseur* XML en Java

Les composants internes	**Description**
SWFS : Le service web du fournisseur sélectionné	Le nouveau service sélectionné pour substituer le service défaillant
Compilateur du WSDL	Le compilateur adapté de WSDL. Il compile le WSDL du service web du fournisseur, et génère une souche cliente (*stub*) en Java
Compilateur Java	Il compile en temps réel le code Java généré
Générateur du Code du Connector	Il génère les classes Java du nouveau *Connecteur de Liaison Dynamique*
Gestionnaire de Déploiement	Il déploie les fichiers du *Connecteur de Liaison Dynamique* dans le conteneur
Gestionnaire du Conteneur à distance	Il invoque le conteneur à distance afin de recharger le contexte et mettre en œuvre le nouveau connecteur. C'est un service web que nous avons développé. Il se connecte au *Tomcat Manager* et exécute des commandes telles que *deploy*, *reload*, etc.
Le code généré pour créer les entités de reconfiguration	**Description**
Code du Connecteur	Les classes Java implantant la logique métier du connecteur
CLD-PD : Connecteur de Liaison Dynamique - Paquetage à Déployer	Les classes Java et les descripteurs de déploiement du *Connecteur de Liaison Dynamique* qui sont compilés et emballés (dans un *Jar*)
Code du *Stub*	La souche cliente du nouveau service web du fournisseur qui est généré dynamiquement

TAB. 3.15 – Les composants architecturaux de la génération et du déploiement du *CLD*

de *Jdom*) et extrait les valeurs des paramètres fonctionnels. En utilisant le *stub* - créé par la méthode `Init-` , elle envoie dynamiquement une requête vers le *SW du fournisseur* avec lequel le *CLD* est lié en se basant sur la programmation réflexive de Java[2]). Par la suite, la réponse sera récupérée et copiée dans le message SOAP réponse envoyé au client.

[2]http ://java.sun.com/j2se/1.3/docs/guide/reflection/

Le comportement interne du Connecteur de Liaison Dynamique

```
Init()
{
 Stub= BuildStubforNewWSProvider();
 // Créer le stub permettant la connexion au nouveau service web du fournisseur

 ParamTypes[]=GetParamTypefromWSDLProvider(WSProviderWsdlUrl);
 // Extraire le type des paramètres fonctionnelles du  WSDL à l'aide de l'API Jdom
}

Bind()
{
 Param[]= GetParamfromSOAPRequest(SOAPEnvelopReq);
 // Extraire les paramètres fonctionnelles de la requête interceptée à l'aide des APIs Jdom et Axis

 ResWSProvider= InvokeWSProvider(Stub, Param[], ParamTypes[]);
 // Invoquer le service du fournisseur  à l'aide de l'API reflection de Java

 SetResult2SOAPResponse(ResWSProvider, SOAPEnvelopResp);
 // Substituer la réponse du service web virtuel par celle du service web du fournisseur
}
```

FIG. 3.2 – Les opérations du *Connecteur de Liaison Dynamique*

3.2.3.2 La reconfiguration par routage adaptatif au niveau HTTP

La reconfiguration niveau HTTP, considère seulement l'enveloppe HTTP qui encapsule le message SOAP. Par la suite, une reconfiguration est égale à une opération de routage d'un message HTTP.

Dans ce cas, le module *Exécuteur de Reconfiguration* est en charge de : 1) offrir le proxy HTTP (qui remplacera le *SW Virtuel* du routage niveau SOAP) et rerouter les requêtes vers le service web du fournisseur (*SW du fournisseur*), 2) mettre en œuvre la reconfiguration suivant un plan établi par le *SW de Décision*.

Dans le prototype implanté (voir section 4.3.3.1), un plan de réparation correspond à une requête SQL paramétrable qui spécifie le service à substituer et le service remplaçant. On peut descendre dans la granularité pour substituer seulement l'opération défaillante. Dans ce cas, on doit spécifier, en plus des noms des deux services, le nom de l'opération à substituer, ainsi que le nom de l'opération remplaçante. Suite à une décision, un nouveau plan doit être chargé par le proxy à partir de la base de données et mis en œuvre.

Du point de vue du client, en utilisant l'abstraction, il envoie ses requêtes au fournisseur à travers notre middleware. En utilisant la réification, les requêtes du client sont adressées au proxy qui s'occupe du routage vers le fournisseur cible. Aussi, il (le proxy) reçoit la réponse du service du fournisseur, et la fait suivre au client.

3.2.3.3 Les niveaux de gestion de la QdS : SOAP vs. HTTP

L'interception des messages au niveau SOAP est une manipulation de haut-niveau des données qui se base sur le *parsing* de structure XML. Cependant, l'interception au niveau HTTP est une manipulation de bas niveau qui se base sur les sockets et le traitement de données sous forme textuelle

Le proxy HTTP est plus approprié pour traiter le monitoring des communications asynchrones, qui se base sur l'échange d'informations dans l'entête du message SOAP. Comme nous l'avons montré dans la figure 2.3, le routage sera réalisé sans toucher au contenu du message SOAP, ce qui n'est pas le cas pour l'interception niveau SOAP.

Pour le déploiement des moniteurs côté client, les deux méthodes sont semblables. Il exige une modification mineure du code client. Côté fournisseur, l'interception niveau SOAP se met en œuvre grâce à une extension du descripteur de déploiement du service web, et elle ne touche que le service concerné. Cependant, l'interception niveau HTTP exige la configuration du serveur afin d'acheminer toutes ses réponses vers le proxy (y compris les autres services hébergés sur ce serveur), ce qui peut représenter un handicap pour les autres services non-inclus dans le processus de gestion de la QdS de *MARQ*. Une meilleure solution est de combiner les deux solutions afin de bénéficier des apports des deux techniques : permettre le monitoring global (du niveau HTTP), manipuler les données de la QdS en tant que structure XML et alléger les hypothèses de déploiement. Ceci se réalise à travers des moniteurs de niveau SOAP (côté client et côté fournisseur), tout en appliquant un routage niveau HTTP au sein des moniteurs SOAP pour la partie de reconfiguration.

3.3 Le protocole d'échange entre les composants de *MARQ*

Durant le processus d'auto-réparation, plusieurs messages sont échangés entre les différents composants de la gestion de la QdS, comme le montre la figure 3.3. Les messages sont décrits dans le tableau 3.16. Ces messages incluent des données du niveau fonctionnel (le nom de l'opération, les valeurs des paramètres, le résultat de la réponse), des données du niveau QdS (*temps de réponse*, *temps de communication*), et des données de diagnostic et de réparation (des alarmes, et des actions de réparation).

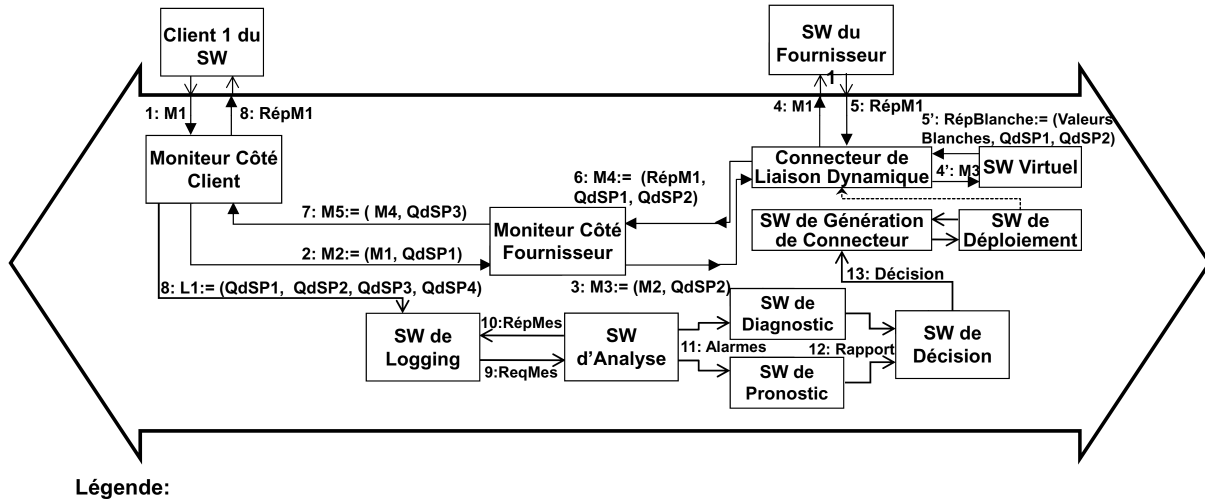

FIG. 3.3 – Le flux de messages échangés entre les services d'auto-réparation

Message	Description
M1	Le message requête
QdSP1	Le paramètre de QdS associé à la requête *M1* du côté du client
QdSP2	Le paramètre de QdS associé à la requête *M1* du côté du fournisseur
RépM1	Le message réponse pour *M1*
QdSP3	Le paramètre de QdS associé à la réponse *RespM1* du côté du fournisseur
QdSP4	Le paramètre de QdS associé à la réponse *RespM1* du côté du client
L1	Le stockage des valeurs de QdS dans le log
ReqMes/RépMes	L'extraction des mesures statistiques liées à la QdS
Alarmes	La détection d'une violation
Rapport	La source de la dégradation
Décision	Le plan de reconfiguration

TAB. 3.16 – La description des messages échangés entre les services d'auto-réparation

Le message *M1* envoyé par le client est intercepté par le *MCC*. Ce dernier l'étend par le premier paramètre de QdS (*QdSP1*) inclus au message sortant *M2*. Le message *M2* est intercepté une deuxième fois par le *MCS*. *M2* est étendu par le deuxième paramètre de QdS (*QdSP2*) inclus au message sortant *M3*. Le message *M3* est intercepté par le *Connecteur de Liaison Dynamique*. Les données fonctionnelles sont extraites du message *M3*. Ceci correspond au contenu initial du message *M1*. Ces données sont utilisées pour créer dynamiquement et localement une requête avec le même contenu, vers le *SW du fournisseur* actuellement lié aux requêtes. En même temps, le message *M3* est transmis au *SW Virtuel*. Les réponses de ces deux services sont collectées par le *Connecteur de Liaison Dynamique* qui substitue la réponse *blanche* par la réponse du *SW du fournisseur*. Il remplace *ValeursBlanches* par *RépM1* dans le message *RépBlanche*. En conséquence, on obtient le message *M4* comme réponse à la requête du client. Le message *M4* est intercepté par le *MCF* pour une troisième extension par le paramètre de QdS *QdSP3* inclus au message sortant *M5*. Puis, il est étendu par le quatrième paramètre de QdS *QdSP4*. Les données de la QdS sont extraites au niveau du *MCC*, et sont envoyées au *SW de Logging* pour être stockées.

Le *SW d'Analyse* interroge périodiquement le *SW de Logging* (message *ReqMes/RépMes*), analyse les valeurs de la QdS, et déclenche des alarmes en cas de violation (message *Alarmes*). Le *SW de Diagnostic/Pronostic* identifie/prédit la source de la dégradation (message *Rapport*). Le *SW de Décision* établit un plan de réparation/reconfiguration (message *Décision*). Ce dernier message sollicite l'*Exécuteur de Reconfiguration* pour débrancher la connexion courante avec le *SW du fournisseur* (*SW du fournisseur 1* dans la figure 3.3), et générer un nouveau connecteur pour router les nouvelles requêtes avec un service équivalent (comme *SW du fournisseur 2* dans la figure 2.1).

Pour illustrer le processus de monitoring, nous avons élaboré des expérimentations avec l'application du *FoodShop* (voir section 4.3). La figure 3.4 montre le contenu des messages SOAP échangés entre le client et le service web *Shop*. Le message *M1* représente le message SOAP initial envoyé par le client. Il contient les paramètres fonctionnels de la requête (*1* et *White meat*, *Orange juice*). Le message *M3* est étendu par deux paramètres de QdS dont leurs valeurs indiquent le temps de communication entre les deux pairs. Le message *M5* représente un message de réponse SOAP enrichi avec des paramètres de QdS. Il montre le résultat de l'invocation du service web *Shop* dans le message de réponse SOAP provenant du *SW Virtuel*. Pour cette invocation, le *temps d'exécution* vaut 147ms.

Fig. 3.4 – Des exemples de messages SOAP

3.4 Le déploiement des composants de *MARQ*

Le déploiement des services et des composants de l'auto-réparation peut être distribué sur trois sites, comme c'est décrit par le tableau 3.17. La partie tierce est un site de confiance entre le client et le fournisseur. Le *Déploiement libre* représente la meilleure configuration de déploiement. En général, plus le *MCC* est proche du côté client, et le *MCF* est proche du côté fournisseur, plus la mesure est plus précise. Les

autres composants peuvent être déployés au site de la partie tierce afin de décharger les sites du client et du fournisseur.

Composantes et services d'autoréparation	Site de Déploiement			
	Des contraintes au site du fournisseur	Des contraintes au site du client	Des contraintes aux sites du client et du fournisseur	Déploiement libre
MCF	Partie tierce	Fournisseur	Partie tierce	Fournisseur
MCC	Client	Partie tierce	Partie tierce	Client
SW du Logging	Partie tierce	Partie tierce	Partie tierce	Partie tierce
Log	Partie tierce	Partie tierce	Partie tierce	Partie tierce
SW d'Analyse	Partie tierce	Partie tierce	Partie tierce	Partie tierce
SW de Diagnostic/Pronostic	Partie tierce	Partie tierce	Partie tierce	Partie tierce
SW de Décision	Partie tierce	Partie tierce	Partie tierce	Partie tierce
Les composantes de l'Exécuteur de Reconfiguration	Partie tierce	Fournisseur	Partie tierce	Fournisseur

Légende: Site de la partie tierce — Site du client — Site du fournisseur

TAB. 3.17 – Les contraintes de déploiement des composants de gestion de la QdS

Les contraintes de déploiement apparaissent à cause de plusieurs raisons telles que les problèmes de sécurité et les droits d'accès. Par exemple, les applications orchestrées avec BPEL limitent l'accès au site client (qui est le processus BPEL lui même) tout en faisant obstacle face au déploiement du *MCC*. Généralement, les fournisseurs (tels que le fournisseur du service *google*) limitent l'accès à leur site web pour des raisons de sécurité, et ils ne permettent pas l'installation de moniteurs ou d'autres composants de gestion de la QdS. Dans ce cas, le *MCF* et l'*Exécuteur de Reconfiguration* peuvent être déployés dans le site de la partie tierce.

3.5 La recherche et la sélection de service de substitution

Pour chaque plan de substitution, un nouveau service web est sélectionné auprès de la liste des services équivalents : *WSDLs de Services Substituables*.

Le processus de substitution d'un service défaillant, appelle le *Service Externe de Découverte des SW* qui assure la recherche ainsi que la sélection d'une liste de services offrant les mêmes fonctionnalités que le service défaillant. La description du WSDL du service sélectionné est stockée par le *WSDLs de Services Substituables*. La recherche doit prendre en considération l'historique de la QdS des services utilisés

pour guider les nouveaux choix. Ceci permettra, par exemple, d'approuver l'utilisation d'un service qui a déjà offert une QdS satisfaisant les besoins des clients et d'éviter l'utilisation d'un service déjà identifié dégradé, sauf si son fournisseur a mis à jour une nouvelle version disponible. Le développement de service fait partie de nos perspectives.

3.6 La conception de *MARQ*

Dans cette section, nous présentons les différents diagrammes élaborés durant le cycle de développement de *MARQ*. Le langage de conception choisi est UML, vu qu'il est un standard et une référence pour la conception de logiciels. La spécification d'UML définit une notation graphique pour visualiser, spécifier et construire des applications logicielles [26].

Le diagramme de composants montré par la figure 3.5, représente l'architecture d'auto-réparation guidée par la QdS. Les quatres composants principaux, *Monitoring*, *Analyse*, *Diagnostic/Pronostic et Décision* et l'*Exécuteur de reconfiguration* sont représentés par des composants interconnectés.

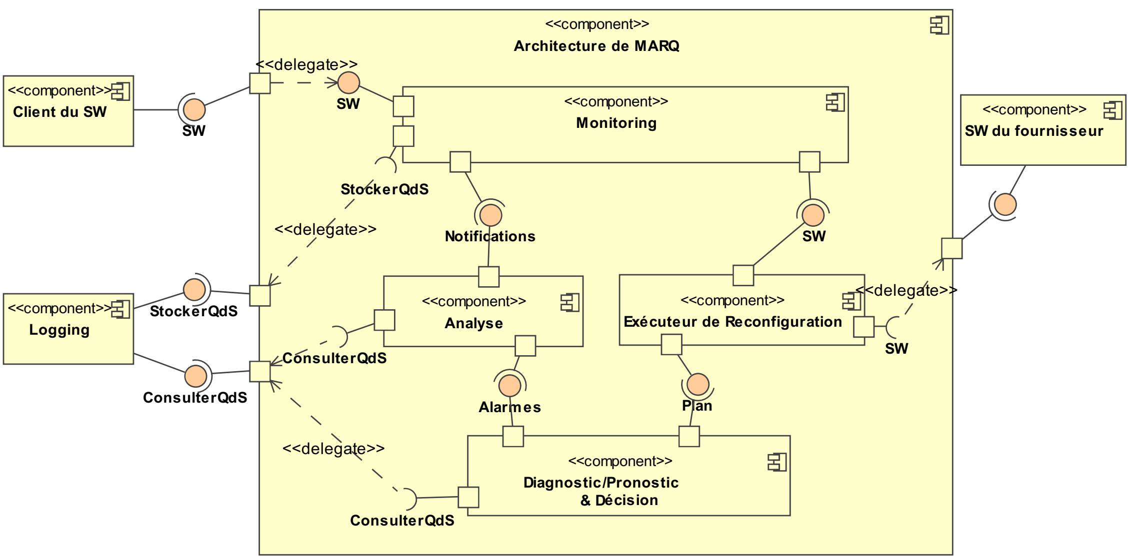

FIG. 3.5 – Diagramme de composants : L'architecture d'auto-réparation

La figure 3.6 détaille le composant de *Monitoring* qui compte plusieurs moniteurs : Le *Moniteur Côté Client*, déployé avec chaque client, et le *Moniteur Côté Fournisseur*, déployé avec le service web du fournisseur. Ainsi, on obtient un moniteur côté

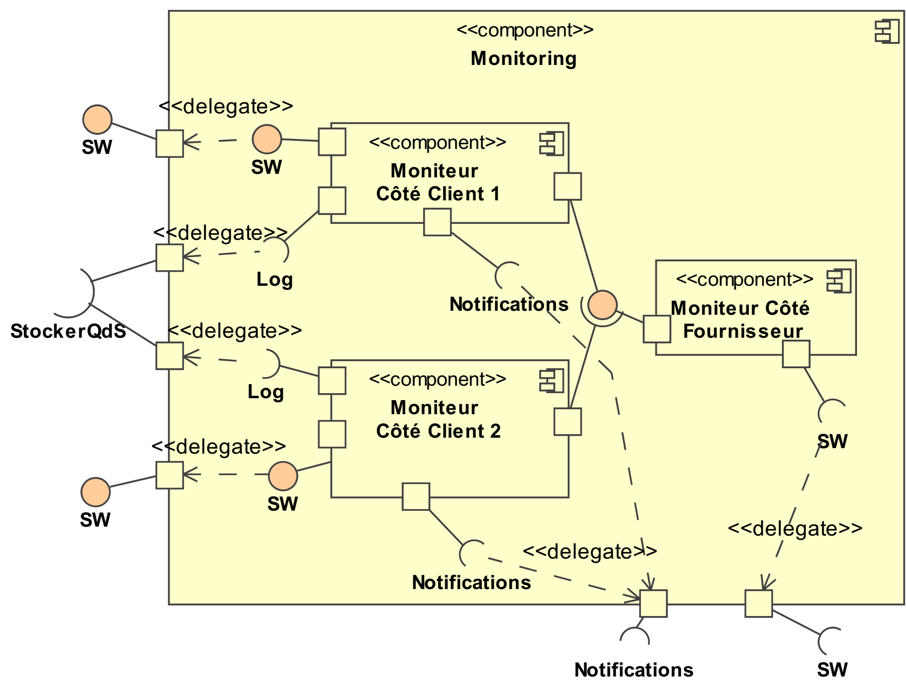

FIG. 3.6 – Diagramme de composants : Le monitoring impliquant deux clients

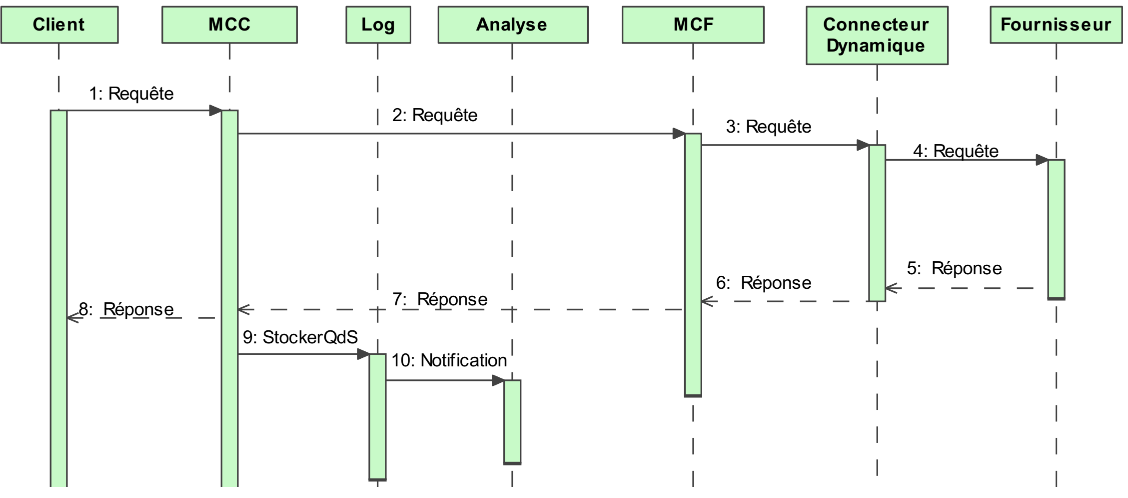

FIG. 3.7 – Diagramme de Séquences : Les actions de monitoring

fournisseur qui intercepte toutes les requêtes vers le service web cible, et un moniteur côté client pour chacun de ses clients.

La figure 3.7 montre les interactions entre le système et les composantes de monitoring dans le cas d'une communication synchrone. Chaque moniteur (*MCC* et *MCF*) intercepte et étend les messages par des valeurs de QdS. Le dernier moniteur extrait

les données concernant la QdS et les envoie au log pour être stockées.

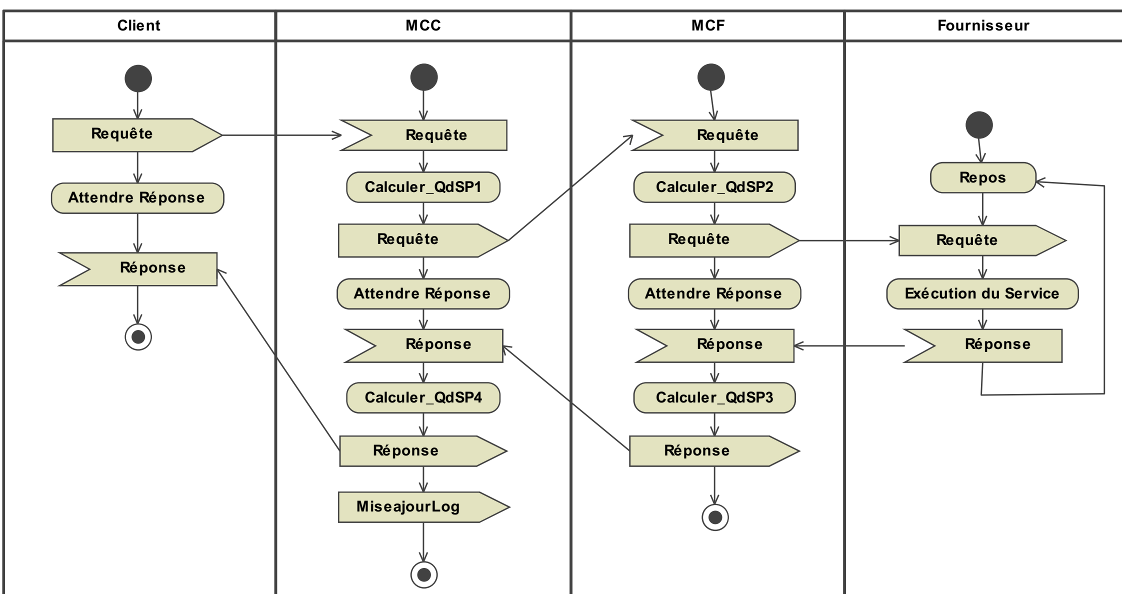

FIG. 3.8 – Diagramme d'activités : Les activités de monitoring

Tandis que la figure 3.7 s'intéresse aux interactions, la figure 3.8 focalise plutôt sur les activités de chaque composant. Cette dernière figure illustre le diagramme d'activité des composants de monitoring. Le client est représenté par le premier *swimlane*, le *Moniteur Côté Client* par le deuxième, le *Moniteur Côté Fournisseur* par le troisième, et le fournisseur par le quatrième. Chaque moniteur calcule la valeur du paramètre de QdS et l'ajoute au message échangé. Suite à une invocation, le fournisseur entre en état de repos, en attente de nouvelles requêtes.

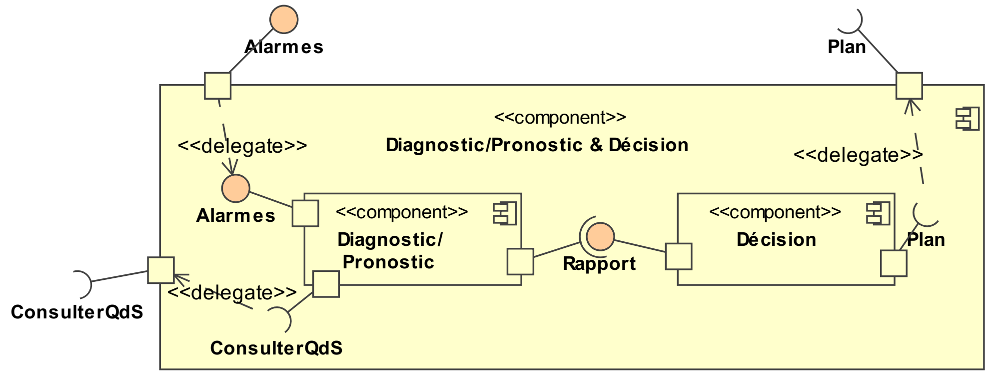

FIG. 3.9 – Diagramme de composants : Le Diagnostic/Pronostic et Décision

La figure 3.9 montre deux composants. Le premier est le composant de *diagnostic/-pronostic* qui identifie/prédit la source de la dégradation en se basant sur des alarmes provenant du composant d'*analyse*. Le deuxième est le composant de *décision* qui élabore le plan de substitution en se basant sur le rapport de diagnostic/pronostic.

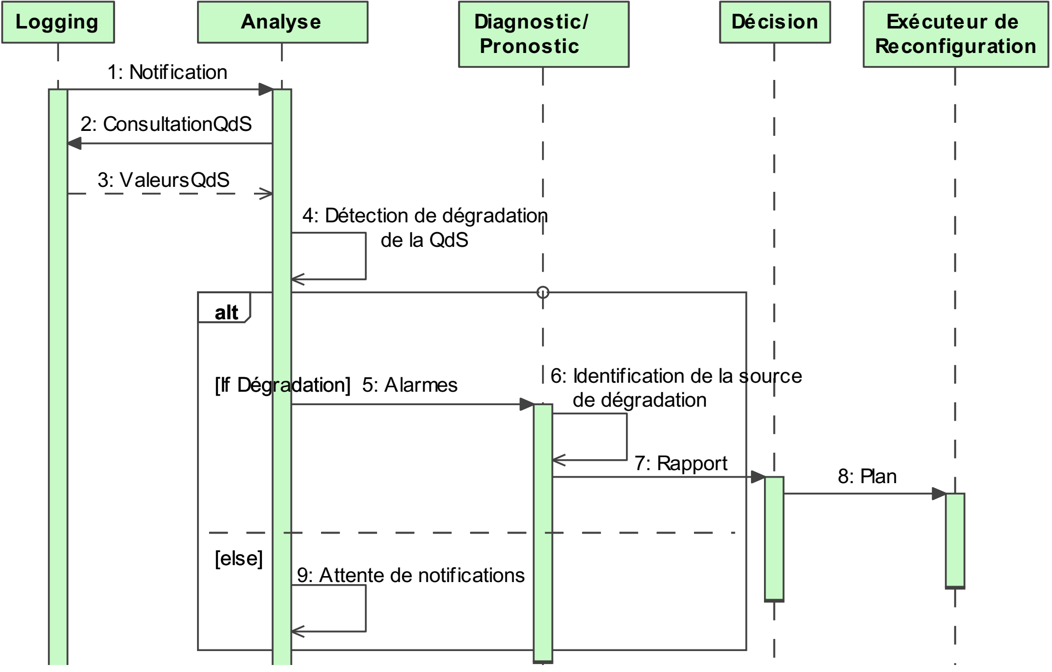

FIG. 3.10 – Diagramme de Séquences : Les interactions entre les modules de *MARQ*

Le composant d'*Analyse* a pour but la détection de violations de la QdS en se basant sur les données de la QdS stockées au log. La succession de violations déclenche des alarmes, comme l'illustre la figure 3.10. Le composant de *Diagnostic* raisonne sur ces alarmes et en cas de dégradation, il identifie la source et envoie un rapport sur l'état du système au composant de *décision* pour planifier les mesures à prendre pour réparer ces dégradations. Le composant de *Décision* demande la substitution du service défaillant auprès du composant *Exécuteur de reconfiguration*.

Le *Connecteur de Liaison Dynamique* est la base de la substitution architecturale. Il est responsable du routage des requêtes des clients vers le service web cible. Le composant *Exécuteur de Reconfiguration*, montré par la figure 3.11, gère ce connecteur. Suite à chaque plan de substitution, un nouveau connecteur est généré par le *SW de Génération de Connecteur*, et déployé par le *SW de Déploiement* afin de connecter les requêtes au nouveau service.

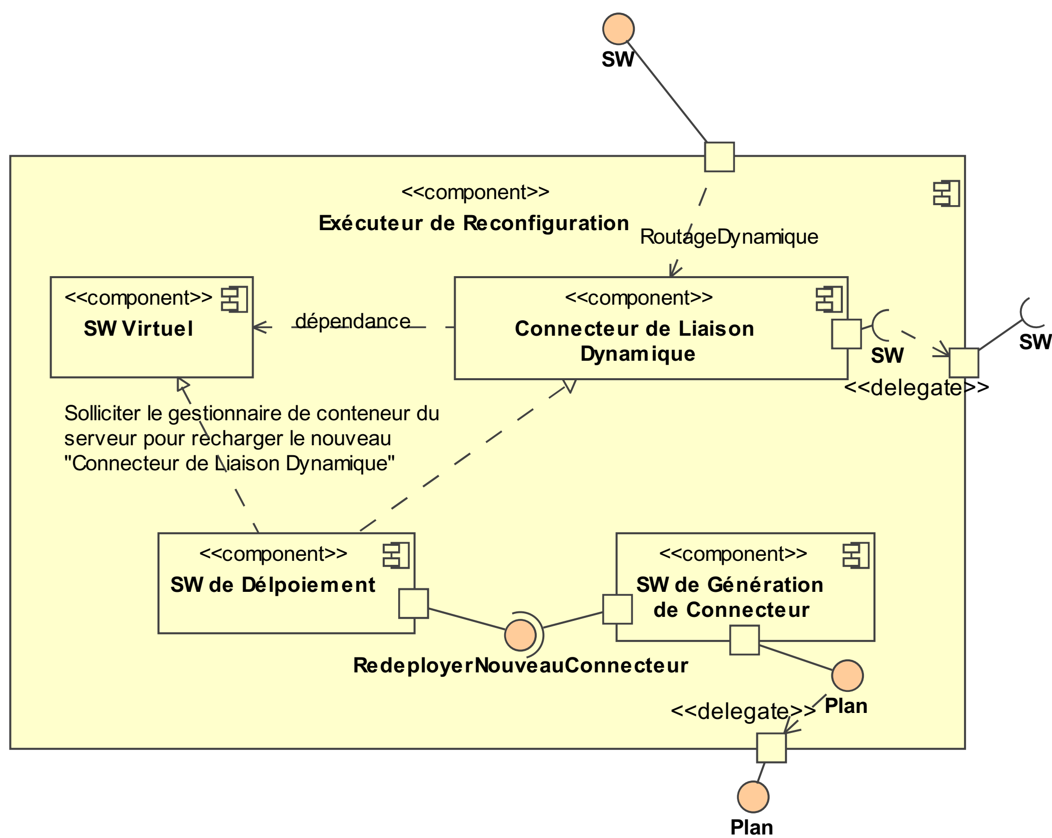

Fig. 3.11 – Diagramme de composants : L'Exécuteur de Reconfiguration

La figure 3.12 montre l'architecture interne du *SW de Génération de Connecteur*. Ce composant reçoit le plan de substitution, génère un *stub* local grâce au compilateur WSDL, et génère et compile le connecteur. Puis, il demande son déploiement.

La figure 3.13 offre une vue d'ensemble du cycle de réparation au sein du middleware *MARQ*. Suite à une dégradation de la QdS du service web du *Fournisseur 1*, un processus de recouvrement est déclenché, partant des alarmes jusqu'à l'action de substitution. La mise en œuvre de l'action de substitution par l'*Exécuteur de Reconfiguration*, déconnecte les requêtes du *Fournisseur 1*, et les connecte au fournisseur sélectionné : le *Fournisseur 2*.

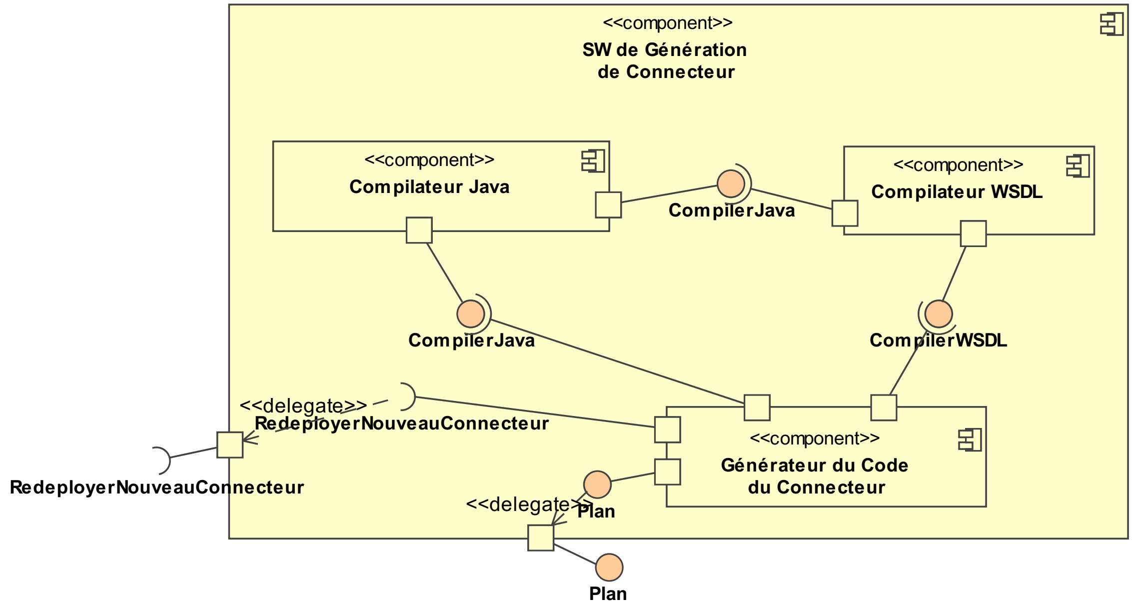

Fig. 3.12 – Diagramme de composants : La génération de connecteur

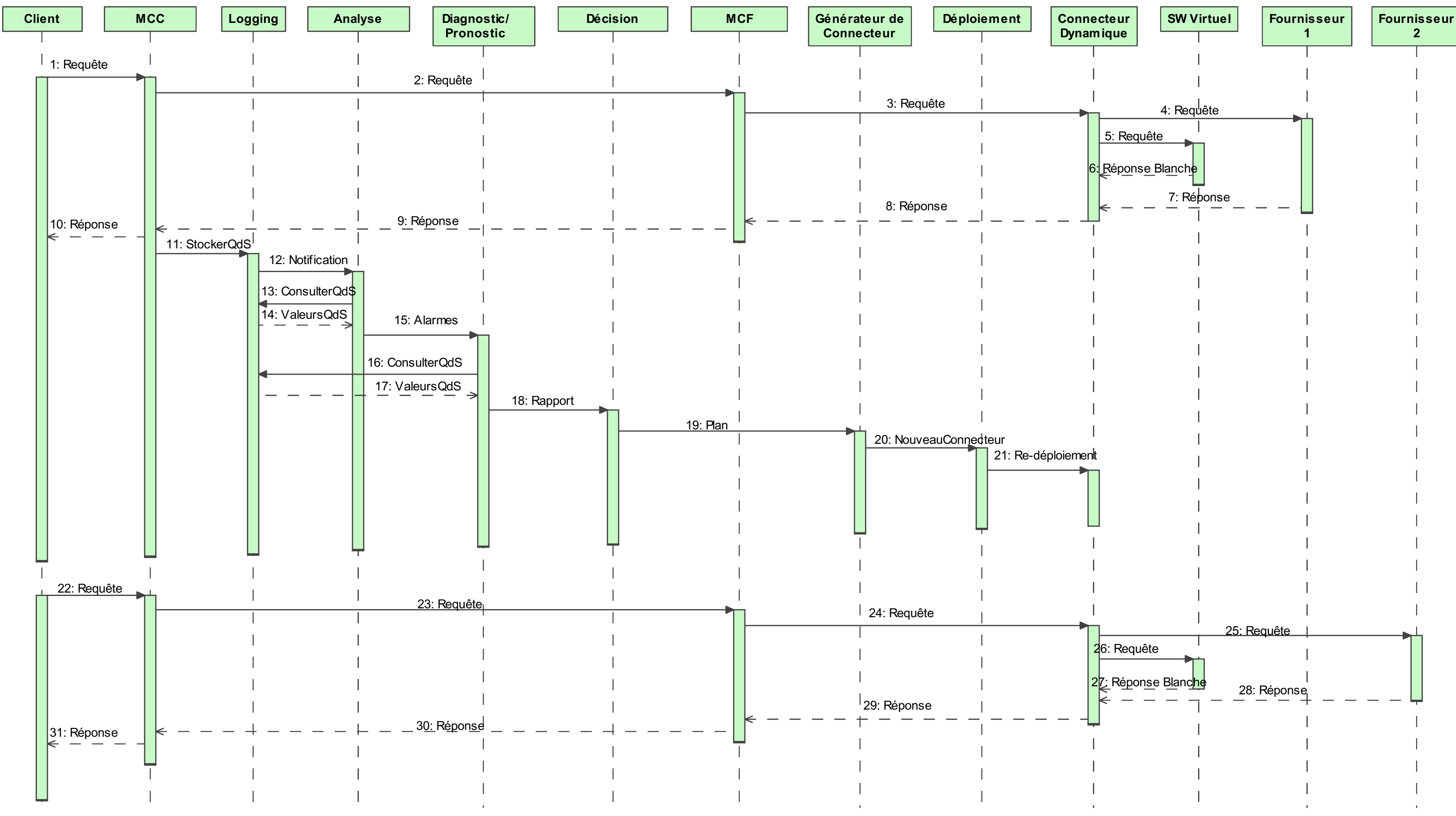

FIG. 3.13 – Diagramme de Séquences : Le processus d'auto-réparation avec *MARQ*

3.7 Règles de transformation en architecture auto-réparable en bus

Dans cette section, nous présentons les trois politiques de transformation d'une architecture initiale de service web en une architecture en bus pour permettre la gestion de la QdS. Nous détaillons les règles et donnons les avantages et les inconvénients de chaque politique.

Premier cas (voir figure 3.14) : Pour chaque couple de service web/client, on instancie un bus.

Les interactions entre chaque couple passe à travers un bus dédié. La gestion de la QdS est propre à chaque client. Aucune interaction n'est prévue entre les bus des différents clients d'un même ou des différents services web. La reconfiguration est exécutée individuellement pour chaque client.

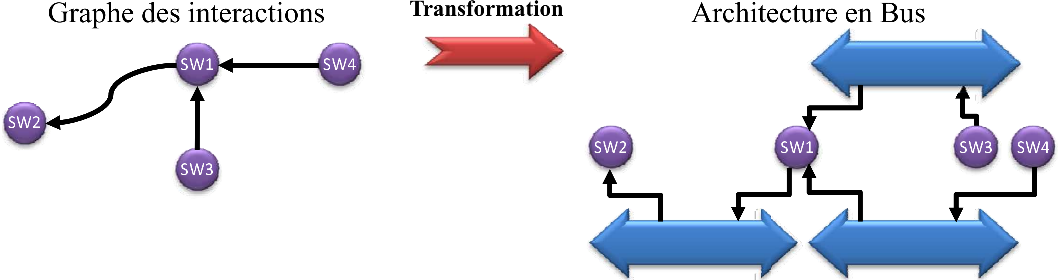

FIG. 3.14 – Un bus par couple client/service

Avantages : Offrir différents contrats de QdS d'un même service web aux différents clients connectés et raffiner les actions de reconfiguration.

Inconvénients : Surcharger l'application par un nombre important de bus à gérer simultanément. La non-interaction des bus, mène à des actions de reconfiguration inutiles (voir section 2.5.1.1).

Réalisation :

-Niveau SOAP : Chaque client aura son propre *Connecteur de Liaison Dynamique* (et par la suite son propre *SW Virtuel*) qui prend en charge la reconfiguration pour celui-ci.

Il est difficile de déployer deux connecteurs côté fournisseur (appartenant à deux bus différents) pour le même service web, qui peuvent distinguer deux requêtes provenant de deux clients indépendants. Les informations incluses dans l'enveloppe SOAP sont insuffisantes pour la distinction.

-Niveau HTTP : Il est difficile de router deux réponses vers deux clients différents. La version actuelle achemine les réponses de tous les clients vers un seul proxy du côté fournisseur.

Deuxième cas (voir figure 3.15) : Pour chaque service web et tous ses clients, on instancie un bus :

Tous les clients d'un service web l'invoque en passant par le même bus. Dans ce cas, une substitution du service web reroute toutes les requêtes de tous les clients vers le nouveau service sélectionné.

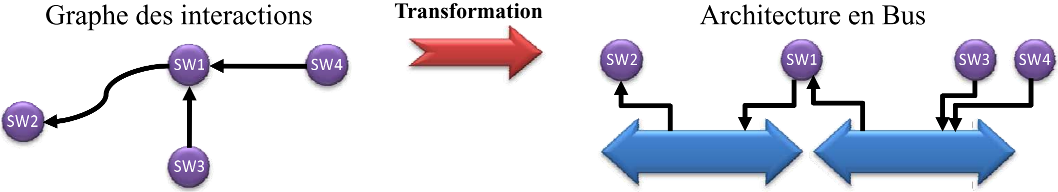

Fig. 3.15 – Un bus par un service et ses clients

Avantages : Avoir moins de charge pour le système (moins de bus). La gestion centralisée de la QdS d'un service web nécessite moins d'interactions distantes.

Inconvénients : Offrir un contrat unique de QdS pour tous les clients.

Réalisation : C'est la version implantée (voir chapitre 4 pour plus de détails).

Troisième cas (voir figure 3.16) : Pour toute l'application (tous les services web), on instancie un bus :

Toutes les interactions au sein de la même application passent à travers un bus unique.

Fig. 3.16 – Un bus pour tous les services de l'application

Avantages : Centraliser le monitoring et la reconfiguration, facilitant la gestion globale de la QdS.

Inconvénients : Surcharger le bus par une masse importante de communication.

Réalisation :

Niveau SOAP : Le *SW Virtuel* englobe toutes les opérations de tous les services. Le routage est fait au niveau opération. Une gestion des opérations redondantes doit être mise en place.

Niveau HTTP : Il faut mettre à jour la base de données de routage du proxy, et configurer le serveur du fournisseur pour qu'il envoie ses réponses au proxy.

3.8 Intégration de l'auto-réparation de niveau classe et de niveau instance

Cette partie présente l'intégration architecturale étudiée du prototype d'auto-réparation guidée par les propriétés fonctionnelles (on s'intéresse au prototype développé dans le cadre du projet WS-DIAMOND par nos partenaires italiens sous forme d'une extension du moteur BPEL : le *SH-BPEL*[69]) et du prototype d'auto-réparation guidée par la QdS (*MARQ*). L'intégration est illustrée avec l'application du *FoodShop*. Le but principal de cette intégration est de gérer simultanément les fautes fonctionnelles et les dégradations de la QdS. Elle permet, aussi, de gérer la phase transitoire durant laquelle le service est indisponible lors d'une opération de reconfiguration (la période incluant la génération et le déploiement d'un nouveau *Connecteur de Liaison Dynamique*). Dans cette étude, on s'intéresse au service web sans état de l'application du *FoodShop* à savoir le *LocalShop*, le *LocalWH*, et le *LocalSupplier*. Le maintien de la cohérence durant un processus d'exécution est considéré par ces propositions d'intégration.

3.8.1 Intégration passive

Une première solution d'intégration étudiée, suppose que les deux middlewares (c'est-à-dire *SH-BPEL* et *MARQ*) sont connectés, mais n'interagissent pas directement à travers l'échange de requêtes. Pour une telle solution, aucune modification n'est nécessaire pour les deux middlewares.

3.8.1.1 La phase transitoire gérée par *SH-BPEL* (I1)

En cas de fautes fonctionnelles, les modules d'auto-réparation fonctionnelle accomplissent la réparation au niveau instance tout en contournant *MARQ*. Cette action

de substitution n'aura pas de mauvaise conséquence sur la gestion de la QdS parce qu'elle concerne seulement l'instance courante du processus en cours d'exécution. Les autres requêtes ne seront pas affectées par cette substitution.

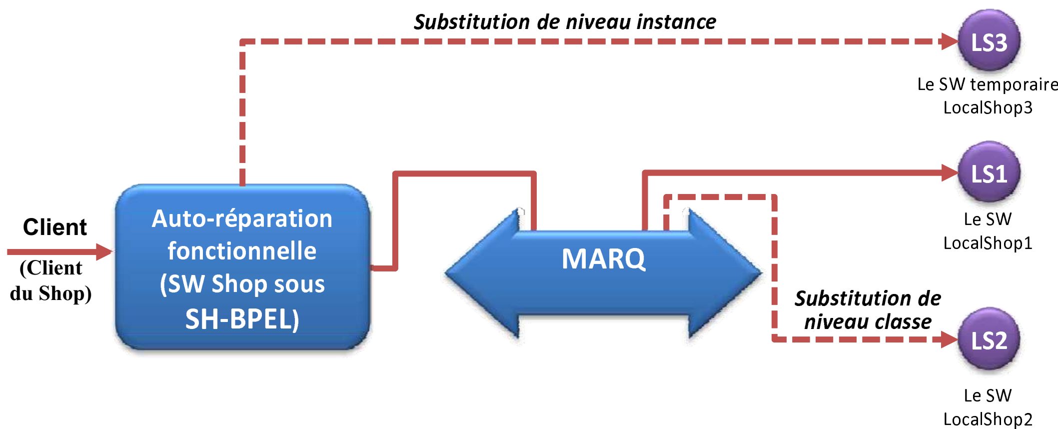

Fig. 3.17 – La phase transitoire gérée par le *SH-BPEL*

En cas de dégradation de la QdS, *MARQ* entame une reconfiguration basée sur une substitution de niveau classe. Durant la phase transitoire (la phase de redéploiement), le service web de l'ancien fournisseur est inaccessible temporairement. Les requêtes envoyées durant cette période produisent des fautes fonctionnelles. Pour cette première architecture d'intégration possible, ces fautes seront gérées par l'auto-réparation fonctionnelle et manipulées comme une perte momentanée de connexion avec le service web, comme le montre la figure 3.17.

3.8.1.2 La phase transitoire gérée par *MARQ* (I2)

Comme pour la section précédente, les deux modules sont connectés sans nécessité des actions interactives. La seule différence réside dans le fait que c'est *MARQ* qui gère lui-même la phase transitoire. Ceci est accompli grâce à un nouveau *Connecteur de Liaison Dynamique* déployé côté client, comme l'illustre la figure 3.18.

Si une faute fonctionnelle est détectée, les modules d'auto-réparation fonctionnelle accomplissent une réparation au niveau instance tout en contournant *MARQ* comme on l'a décrit dans la section précédente.

Au cours d'une opération de reconfiguration effectuée par *MARQ*, le service web de l'ancien fournisseur est inaccessible temporairement durant la phase transitoire. Contrairement à l'architecture précédente, ce problème est géré grâce à un *Connec-*

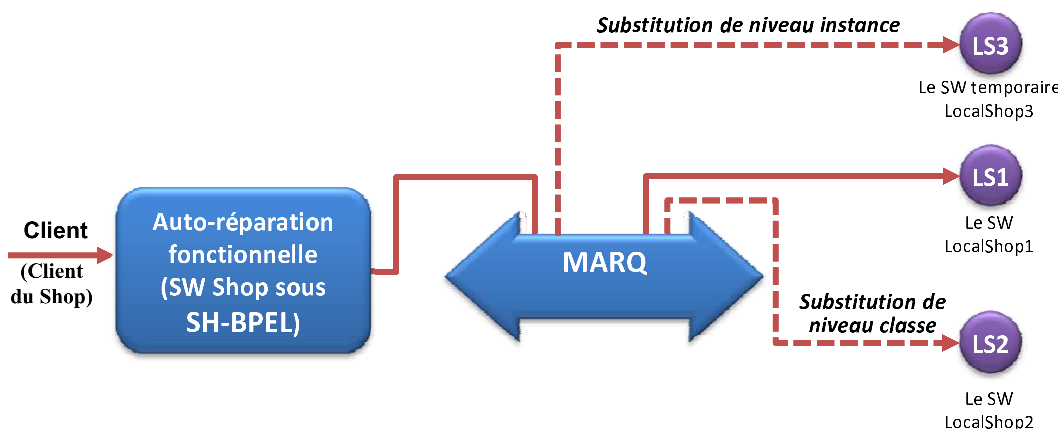

FIG. 3.18 – La phase transitoire gérée par *MARQ*

teur de Liaison Dynamique déployé côté client. Ce connecteur prend en charge la redirection des requêtes des clients à un nouveau fournisseur temporaire (Le SW transitoire *LocalShop3* à la figure 3.18), jusqu'à la fin de la phase transitoire.

3.8.2 Intégration active (I3)

Pour l'intégration active, illustrée par la figure 3.19, les deux prototypes sont connectés par des interfaces étendues afin de permettre l'interaction directe et active. Ils coopèrent pour échanger des informations sur le plan de réparation actuel ou pour demander une reconfiguration.

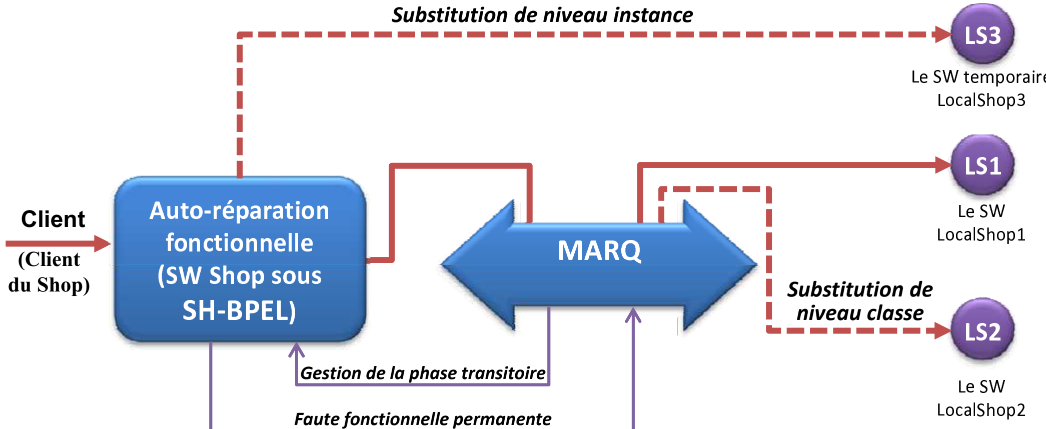

FIG. 3.19 – L'intégration active entre *MARQ* et *SH-BPEL*

D'une part, *MARQ* peut demander, par exemple, à *SH-BPEL* une reconfiguration

durant la phase transitoire. Pour s'occuper de la déconnexion temporaire du fournisseur du SW, *MARQ* envoie des alarmes aux modules de *SH-BPEL*. Il l'informe de cette phase transitoire durant la reconfiguration de niveau classe et pour garantir la disponibilité continuelle du service pour les processus en cours d'exécution. D'autre part, *SH-BPEL* peut demander une réparation de niveau classe des fautes fonctionnelles persistantes. Il envoie des demandes de reconfiguration à *MARQ* afin de substituer le service pour les futures invocations.

3.8.3 Illustration

Dans cette partie, nous considérons les différentes possibilités d'intégration des modules de *MARQ* et du *SH-BPEL* avec l'application du *FoodShop*. L'instance de *MARQ* est déployé et connecté avec l'instance de *SH-BPEL* comme montré dans la figure 3.20. Le processus d'auto-réparation considère les interactions impliquant des services web sans état (par exemple, les services web *LocalShop* et *LocalWH*). Autrement, on ne considère pas les cas ou le service web du fournisseur est assimilé à un processus BPEL.

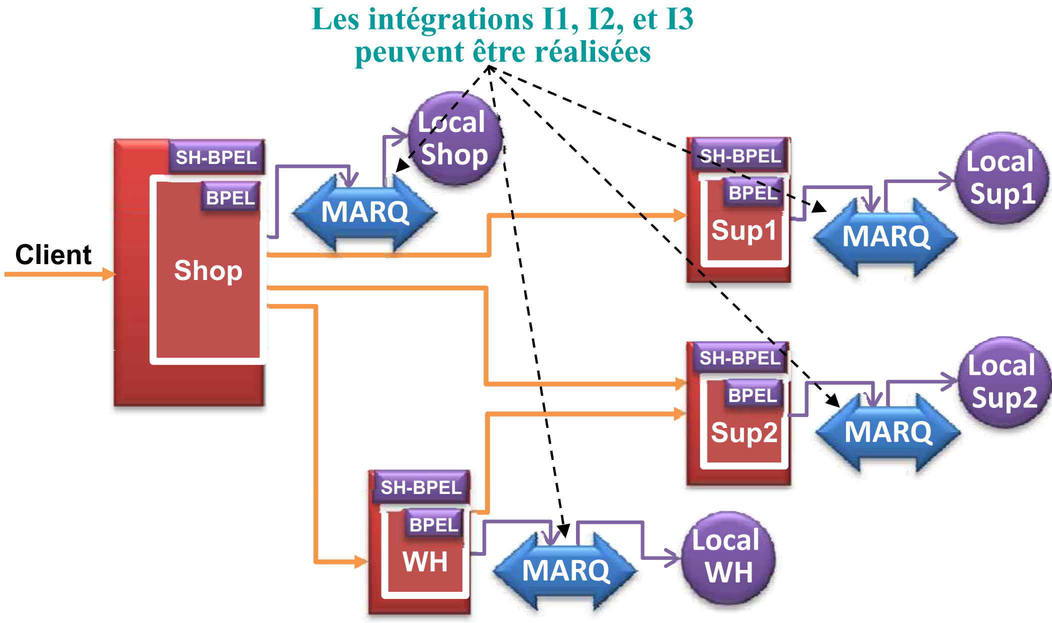

FIG. 3.20 – L'intégration de *MARQ* et *SH-BPEL* illustrée par le *FoodShop*

Cette version est implantée et intégrée avec le *FoodShop* en utilisant le schéma de l'intégration passive (I1) dans la section 3.8.1.1.

3.9 Conclusion

Dans ce chapitre, nous avons mis l'accent sur la reconfiguration par substitution au sein de *MARQ*. En effet, la substitution d'un service web est équivalente à une action de reconfiguration architecturale, qui est mise en œuvre grâce à un *Connecteur de Liaison Dynamique*. Ce dernier est généré et déployé automatiquement au cours de l'exécution. La conception, ainsi que l'intégration de notre middleware avec l'auto-réparation du niveau instance sont étudiées et présentées dans ce chapitre. La validation et l'expérimentation feront l'objet du chapitre suivant.

4

Chapitre 4 : Expérimentations et Applications

4.1 Introduction

L'objectif de ce chapitre est de valider l'approche proposée. Pour ce faire, nous présentons deux scénarios de deux différentes applications. Le premier scénario considère un *Système de Revue Coopérative* (*SRC*), qui est une coopération de services web interactifs gérant les conférences scientifiques. Le deuxième scénario représente un cas d'étude pour le commerce électronique : le *FoodShop*. Cette application est implantée sous forme de chaîne de fournisseurs, où une orchestration de services web agissant simultanément comme des fournisseurs et des clients avec des conversations hautement imbriquées. Les fonctionnalités de *MARQ* sont illustrées à travers ces applications, partant des expérimentations à grande échelle et du diagnostic/pronostic, jusqu'à la reconfiguration au moment de l'exécution.

4.2 La revue coopérative : Un processus de coordination distribuée

La plupart des communautés scientifiques ont, récemment, établi un environnement logiciel afin de soutenir la gestion de conférences. Leur but est de réduire le coût de communication, pour faciliter la coopération, et pour appliquer un processus d'évaluation équitable. Le *SRC* (Système de Revue Coopérative) est une implantation du processus de gestion de conférences scientifiques proposé dans le cadre du projet Européen WS-DIAMOND. La figure 4.1 montre une vue d'ensemble de son architecture. Cette architecture a pour objectif d'assurer une flexibilité d'échange de services entre les différents composants du système de gestion de conférences à travers des instances de *MARQ*. L'architecture développée est une architecture orientée service divisée en trois parties (voir figure 4.1) :

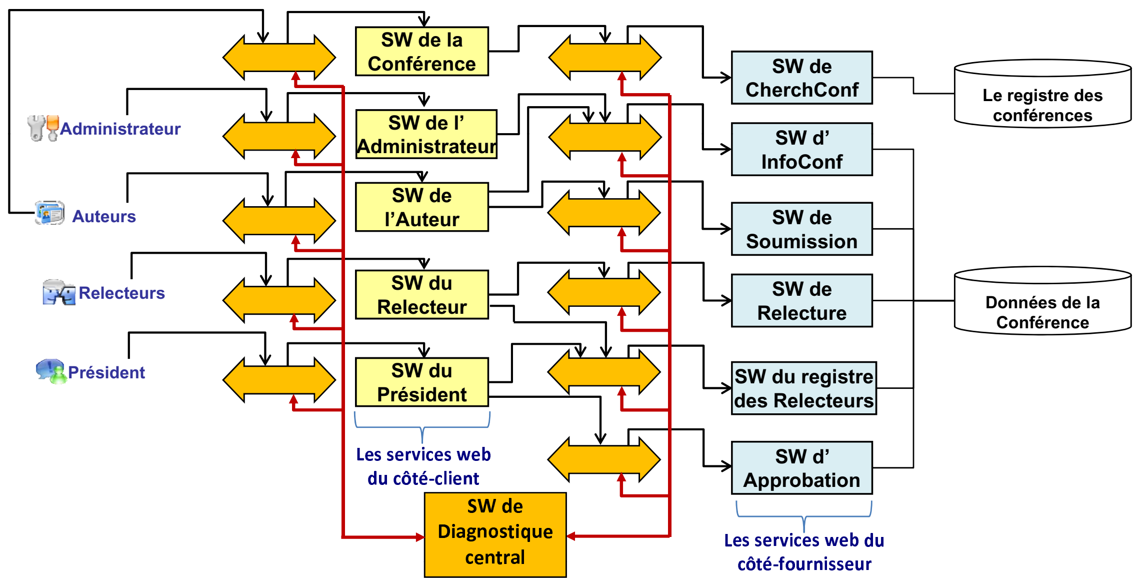

FIG. 4.1 – Le Système de Revue Coopérative avec les composants d'auto-réparation

Les Clients : ils sont composés des acteurs du système, à savoir : Administrateurs, Auteurs, Relecteurs et Président.

Les composants d'auto-réparation : ils gèrent les dégradations de la QdS entre chaque paire de client/fournisseur. En raison de la complexité des interactions, nous avons opté pour une entité centrale de diagnostic afin de synchroniser et coordonner les actions de réparation. Le service de diagnostic central reçoit les rapports sur l'état d'un service web de l'application des services de diagnostic locaux. Il utilise sa connais-

sance de la structure architecturale de l'application pour raisonner efficacement et identifier la source de la dégradation. Par la suite, un rapport de diagnostic final est envoyé aux différents services de décision locaux qui déclenchent, à leur tour, la substitution des services dégradés.

Les services web : ils incluent les services web du côté-client et ceux du côté-fournisseur. Les services web du côté-client sont utilisés par les clients pour explorer dynamiquement et invoquer les services adéquats. Les services web du côté-fournisseur offrent les fonctionnalités relatives à la revue coopérative.

Comme c'est décrit dans la figure 4.1, plusieurs acteurs collaborent afin d'accomplir les tâches de gestion du processus de revue coopérative, à savoir : le président du comité, les relecteurs, et les auteurs. Le processus de revue est précédé d'une étape de recherche de conférences appropriées pour les domaines d'activité de l'auteur. Pour ce faire, nous avons développé deux types de services web principaux : les services web de *RevueCooperatives* et le service web de *CherchConf*. Le service web *CherchConf* permet aux utilisateurs de rechercher des conférences existantes selon des critères bien spécifiques. Les services web de *RevueCooperatives* fournissent plusieurs fonctionnalités comme la gestion de la soumission, et le processus de relecture. Dans notre travail, nous avons mesuré la QdS du *SRC*. Nous nous sommes intéressés principalement à la mesure et à l'évaluation de la QdS du scénario de recherche de conférences.

4.2.1 Description du scénario « Recherche de conférences »

Le diagramme montré par la figure 4.2 décrit le processus où les utilisateurs cherchent des appels à communication susceptibles d'être pertinents pour leurs domaines de recherche.

L'acteur envoie une requête de recherche de conférences (*CherchConf*) avec les paramètres contenant toutes les exigences sur la conférence (*DonnéesConf*, telles que thème, date limite de soumission, éditeur, etc.). Le service *Recherche* reçoit cette requête et génère une nouvelle requête *ConfRequete(DonnéesConf)* auprès du service *Choréographeur*. Ce dernier lance une première recherche auprès du registre en invoquant l'opération *ConfRequeteReg(DonnéesBasiqueConf)*. *DonnéesBasiqueConf* représente les données de base d'une conférence. A partir du résultat de la dernière opération, le *Choréographeur* invoque la méthode *ConfRequeteFourn(DonnéesConf)* du service *FournisseurInfoConf* qui retourne la liste de toutes les conférences répon-

dant aux critères de recherche déjà définis. L'extraction de cette liste se fait suite à l'accès à la base de données qui contient la liste des conférences.

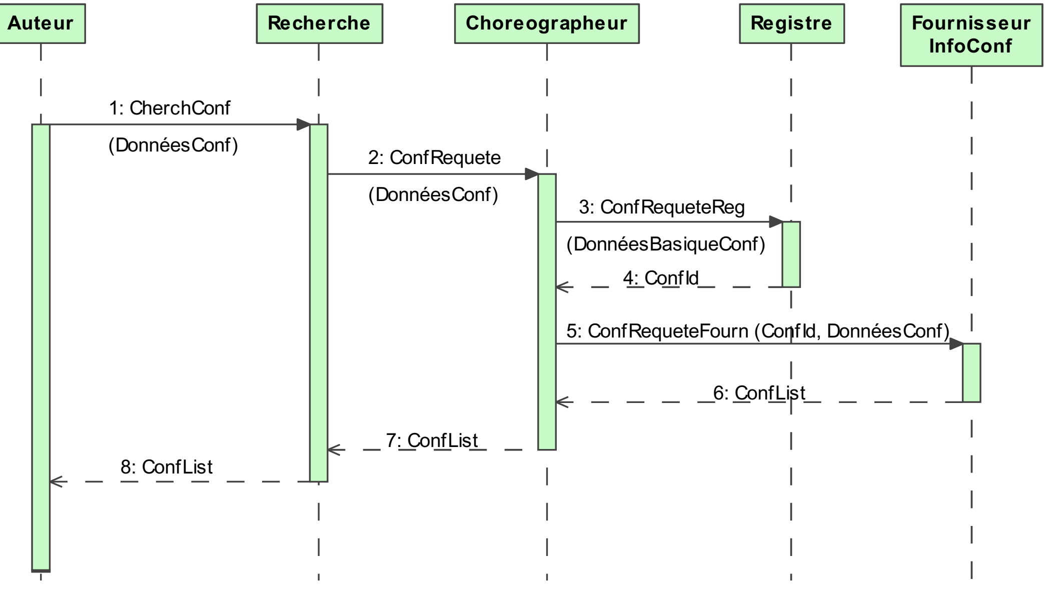

FIG. 4.2 – Diagramme de séquences : Recherche de Conférences

Pour l'implantation, les fonctionnalités des entités *Recherche* et *Choréographeur* sont incluses dans le service web du côté client, et les fonctionnalités des entités *Registre* et *FournisseurInfoConf* sont offertes par le service web du côté fournisseur.

4.2.2 Hypothèses d'implantation

Le comportement du moniteur n'est pas décrit dans le fichier WSDL. Le code du moniteur côté fournisseur, qui est inclus dans la classe *ServerHandler*, est enregistré du côté fournisseur. Pour le mettre en œuvre, le fichier *server-config.wsdd* (Web Service Deployment Descriptor sous le répertoire *WEB-INF* de l'implantation du service web) doit être modifié, afin de contenir la description suivante :

```
<!-- ****************************************** -->
    <handlerInfoChain>
        <handlerInfoclassname="ServerHandler"/>
    </handlerInfoChain>
<!-- ****************************************** -->
```

Le moniteur du client, dont la logique métier se trouve dans la classe *ClientHandler*,

est enregistré du côté client. Aussi, il faut ajouter les lignes suivantes dans le code du client :

```
/****************************************/
    java.util.List list = helloService.getHandlerRegistry().getHandlerChain(
        new Name(nameSpaceUri, portName));
    list.add(new javax.xml.rpc.handler.HandlerInfo(ClientHandler.class,null,null));
/****************************************/
```

Nous travaillons actuellement sur l'intégration des moniteurs côté client dans l'API d'Axis. Ainsi, le client n'aura pas besoin de modifier son code. Il n'a qu'à remplacer seulement l'API d'Axis initiale par la nôtre. En d'autres termes, nous offrons une nouvelle version du fichier *axis.jar* pour remplacer l'ancien fichier.

4.2.3 Environnement de développement

Afin de lancer un nombre important de clients sur un ou plusieurs services, et être le plus proche du contexte de grande échelle, nous avons choisi d'effectuer les mesures de QdS sur une grille informatique.

4.2.3.1 Les grilles de calcul

Les grilles informatiques sont des plate-formes de calcul à grande échelle, hétérogènes et distribuées. Le concept de grille informatique correspond à la réalisation de vastes réseaux mettant en commun des ressources informatiques géographiquement distantes. Les grilles de calcul ou de données permettront d'effectuer des calculs et des traitements de données à une échelle sans précédent. Le concept de grille peut englober des architectures matérielles et logicielles très différentes, en fonction des objectifs recherchés. Nous identifions deux classes de plate-formes différentes :

- Grilles de production : elles sont des plate-formes applicatives, qui doivent fournir les mêmes services (heures CPU, temps de réponse) de manière constante, ininterrompue et fiable. C'est typiquement le fonctionnement des centres de calcul qui sert de modèle pour les grilles de production.
- Grilles expérimentales de recherche : elles sont par définition motivées par des travaux de recherche expérimentaux. Il s'agit d'une part de recherches en informatique (systèmes distribués, calcul parallèle, réseaux, protocoles, bases de données, fouille de données) et d'autre part de recherches dans d'autres domaines

(physique, biologie, calcul à haute performance, etc...).

Les grilles expérimentales de recherche n'offrent pas de garantie de service, dans la mesure où des modifications peuvent être apportées sur l'infrastructure ou sur le système d'exploitation, afin d'étudier de nouveaux concepts ou algorithmes, de tester un nouveau middleware, ou encore de mettre en œuvre un nouveau protocole réseau. En résumé, les caractéristiques des grilles de recherche sont la souplesse, le temps de réponse, et l'ouverture.

4.2.3.2 La plate-forme *Grid'5000*

La plate-forme *Grid'5000* est une grille matérielle et logicielle, interconnectant à très haut débit une dizaine de clusters de PC de grandes tailles. Pour fixer un ordre de grandeur, chaque cluster pourrait comprendre 500 unités de calcul, d'où le total de 5000 qui donne le nom de code du projet *Grid'5000* [18]. La liste des sites inclut Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Nice et Toulouse, soit neuf villes couvrant le territoire français de façon complète[1]. La plate-forme *Grid'5000* pourrait être considérée comme un grand émulateur de l'Internet. Dans ce cas, les nœuds de calcul seront utilisés soit comme de vrais clients ou serveurs, soit comme des systèmes autonomes de l'Internet (routeurs logiciels émulant latence et pertes de paquets).

4.2.3.3 Configuration et préparatifs de l'expérimentation

Afin de réaliser des expérimentations et des mesures de QdS sur le service web de *Recherche de conférences*, nous avons utilisé la grille *Grid'5000* . L'architecture de déploiement est montrée par la figure 4.3. Les services web sont déployés sur le site de Toulouse, et les clients sont déployés sur les autres sites (Bordeaux, Lyon, etc.). Plusieurs requêtes sont lancées en même temps, à travers le middleware *MARQ*. La grille *Grid'5000* est composée de plusieurs nœuds opérant avec Fedora Linux (voir le tableau 4.1).

La configuration de l'environnement expérimental est illustrée dans le tableau 4.1. Nous avons réservé des nœuds localisés sur différents sites. Nous choisissons deux nœuds comme serveurs d'application hébergeant les services web. Le premier pour le service web du côté client (le *chorégrapheur*, voir figure 4.2) et le deuxième

[1]Le nombre de villes était de neuf au moment de la réalisation des expérimentations, mais actuellement le nombre est de quatorze villes universitaires.

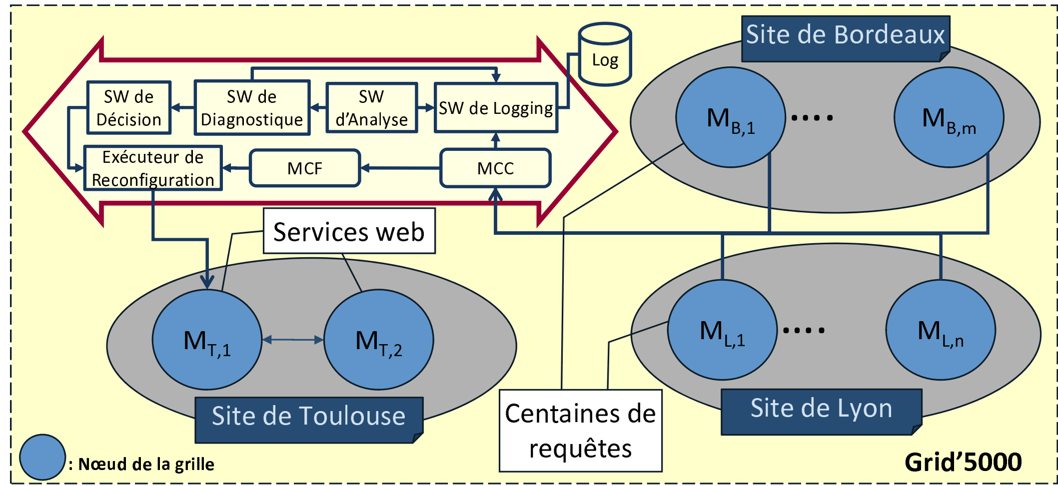

FIG. 4.3 – L'infrastructure de l'expérimentation

Nom du Site	*Toulouse*	*Sophia*		*Bordeaux*	*Lyon*
		Azur (cluster1)	Helios (cluster2)		
Modèle	Sun Fire V20z	IBM eServer 325	Sun Fire X4100	IBM eServer 325	IBM eServer 325
CPU	AMD Opteron 248 2.2 GHz (dual core)	AMD Opteron 246 2.0 GHz (dual core)	AMD Opteron 275 2.2 GHz (dual core)	AMD Opteron 248 2,2 GHz (dual core)	AMD Opteron 246 2.0 GHz (dual core)
Mémoire	2 GB	2 GB	4 GB	2 GB	2 GB
Débit du réseau	Gigabit Ethernet	2 x Gigabit Ethernet	4 x Gigabit Ethernet	Gigabit Ethernet	Gigabit Ethernet

TAB. 4.1 – La configuration des nœuds de la plate-forme *Grid'5000*

pour le service web de *Recherche de conférences*. Nous avons réservé et exécuté respectivement 1, 3, 5, 10, 25, 50, 75, 100, 200, 350 et 500 clients. Pour chaque requête, le *MCC* envoie les valeurs des paramètres de QdS au log.

Pour la réservation sur la grille, il faut se connecter au frontal d'un site (par exemple le site toulousain) en s'authentifiant avec un login et un mot de passe. La réservation est réalisée avec la commande *oargridsub*. Elle permet la réservation de plusieurs nœuds sur différents clusters.

Un exemple de réservation est montré dans ce qui suit :

```
/**********************************/
oargridsub helios :nodes=47 -w 02 :00 :00 idpot :nodes=20, lille :nodes=40,
capricorne :nodes=30, sagittaire :nodes=40, azur :nodes=50,helios :nodes=50,
toulouse :nodes=20, paraci :nodes=60, paravent :nodes=90
/**********************************/
```

Cette commande permet de réserver 47 nœuds du cluster *helios*, 20 nœuds du cluster *idpot*, 40 nœuds du cluster *Lille*, ..., pour une durée de deux heures spécifiée à travers l'option *-w*. L'exécution réussie de la commande *oargridsub* retourne un numéro de réservation appelé *IdJob*. Sur chaque site de la grille *Grid'5000*, nous disposons d'un répertoire personnel indépendant des autres sites. Avant de réaliser une expérience sur la grille, nous devons nous assurer que les codes et leurs configurations sont bien disponibles sur les sites qui seront utilisés. En fait, chaque site dispose d'un frontal. Nous nous servons de la commande *rsync*, pour appliquer la synchronisation des données du répertoire personnel. Voici un exemple pour la mise à jour des données entre frontaux :

```
/**********************************/
rsync –delete -avz  /mydocs sync.lille.grid5000.fr :
/**********************************/
```

Cette commande permet de synchroniser le frontal du site de Lille avec le contenu du répertoire *mydocs* situé dans le frontal de Toulouse.

Nous utilisons la commande suivante pour récupérer la liste des noms des nœuds réservés dans le fichier *~/machines* :

```
/**********************************/
oargridstat -w -l IdJob > ~/machines
/**********************************/
```

Avec *IdJob* qui dénote l'identificateur de la réservation.

Pour la configuration et le lancement des serveurs et des bases de données, nous utilisons des scripts shell Unix. A travers ces scripts, nous pouvons construire et exécuter plusieurs clients concurrents en même temps.

Le script suivant lance en parallèle les clients qui existent sur les nœuds réservés et qui sont inscrits dans le fichier *~/machines*.

```
/***********************************/
#!/bin/csh
foreach machine ('cat ~/machines')
ssh $machine ~/exp/RechConfClient/run_client.sh&
end

/***********************************/
```

Nous avons utilisé *Apache Tomcat5.5* comme serveur d'application, *Axis1.4* comme conteneur de service web, *Java1.5* comme langage de programmation et *MySQL5* comme système de gestion de base de données.

En utilisant l'ensemble de ces commandes, nous avons pu construire et lancer plusieurs clients en même temps. Chaque client envoie des requêtes *SOAP* vers le service web *ConfSearch* qui seront interceptées par les moniteurs de *MARQ*. Les clients invoquent continuellement le service web durant 10 minutes. Les expériences sont réalisées et répétées plusieurs fois, et nous gardons à la fin les valeurs moyennes. Chaque requête sera interceptée quatre fois afin qu'elle soit enrichie avec les valeurs des paramètres de QdS. Avant que la réponse n'atteigne le client, le *MCC* lance un processus léger (*Thread*) pour enregistrer les valeurs des paramètres de QdS recueillies. Le but du processus léger est de ne pas bloquer la réponse, au niveau du moniteur, au moment du logging. Ces valeurs sont stockées dans une base de données MySQL à travers le *SW de Logging* (voir la figure 2.1). En se basant sur les formules montrées dans la section 1.2, nous interrogeons le log et nous déduisons les temps d'exécution et de réponse, aussi bien que la disponibilité et le débit. Par la suite, nous dressons et interprétons des courbes pour avoir une idée sur le comportement du service web durant différentes conditions de charge. Cela nous aide à bien diagnostiquer l'état de service et à bien gérer la QdS.

4.2.3.4 Résultats et analyses

Le tableau 4.2 montre les résultats des expérimentations réalisées avec le service web *ConfSearch*. La première ligne montre qu'un seul client invoque le service web *ConfSearch* 6193 fois durant 10 minutes. Avec 500 clients concurrents, 50% des requêtes échouent. On remarque que l'augmentation du nombre de clients mène à

une surcharge du serveur et décroît la performance. Cependant, la valeur du temps d'exécution augmente d'une façon monotonique, alors que le temps de communication varie en raison du traffic injecté par les autres utilisateurs de la grille *Grid'5000*.

Nombre de clients	Nombre de Requêtes	Requêtes réussites	Requêtes échouées	Duré expérimentation	Temps d'Exécution (ms)			Temps de Communication (ms)		
					Min	Max	Avg	Min	Max	Avg
1	6464	6193	271	10Min	10	222	18,163	40	1253	59,767
3	16285	15368	917	10Min	10	654	22,795	35	2447	71,895
5	18218	16903	1315	10Min	9	638	28,496	35	155917	99,683
10	29783	25528	4255	10Min	9	964	55,076	27	5454	128,297
25	35304	26337	8967	10Min	9	2989	79,059	33	5700	310,039
50	39087	25563	13524	10Min	9	4903	87,033	35	21554	737,783
75	42227	24554	17673	10Min	9	6052	95,330	31	185258	1352,367
100	43118	24380	18738	10Min	9	6021	97,900	39	211162	1780,646
200	44072	24084	19988	10Min	9	5626	116,434	34	245921	1653,905
350	44243	24869	19374	10Min	9	6271	116,985	24	489053	1735,029
500	47981	25736	22245	10Min	9	5919	117,739	24	217436	1660,850

TAB. 4.2 – Tableau récapitulatif des valeurs issues de l'expérimentation.

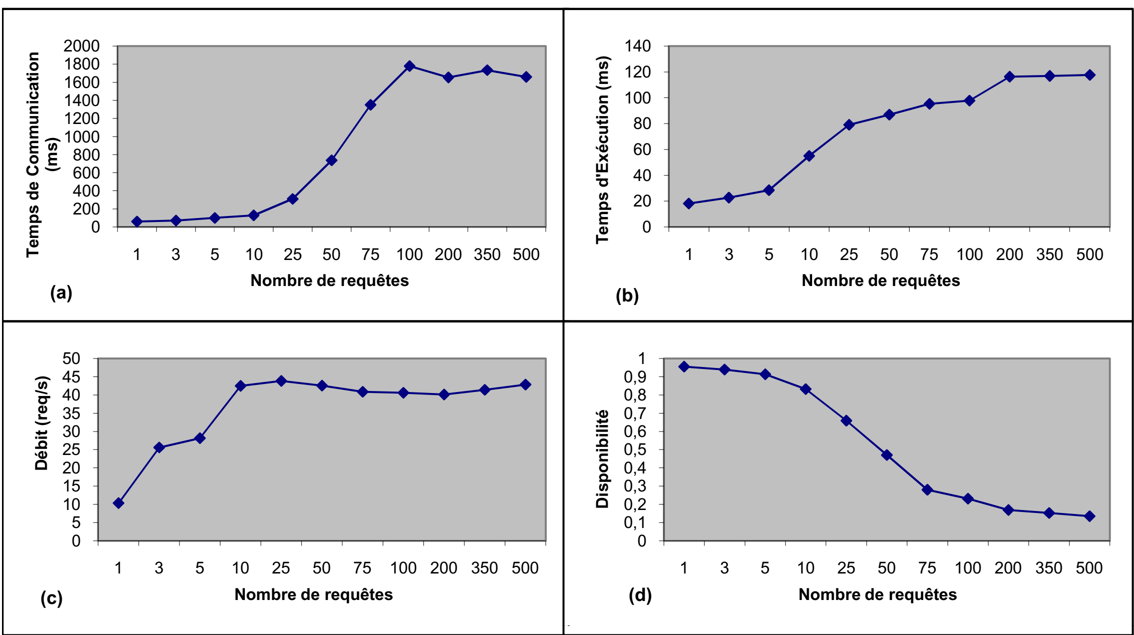

FIG. 4.4 – La variation des paramètres de QdS

La figure 4.4.a affiche la variation du temps de communication selon la croissance du nombre des clients. Les courbes continuent leur croissance jusqu'à atteindre le niveau de 100 clients. A partir de ce niveau, elle reste aux alentours de 1,7 seconde.

Le temps de communication varie entre environ 100ms pour 10 clients à 2000ms pour 100 clients. Il augmente rapidement tout en suivant le nombre de clients, ce qui montre l'importance de ce paramètre dans le temps de réponse observé par les clients. De telles informations sont analysées et modélisées pour soutenir un monitoring et un diagnostic rigoureux pour cette application.

La figure 4.4.b montre l'évolution du temps d'exécution tout en augmentant le nombre des clients. Il augmente continuellement d'environ 20ms avec 1 client jusqu'à 120ms avec 500 clients simultanés. La croissance du nombre des clients surcharge le service et fait régresser sa performance.

La figure 4.4.c présente la variation du débit avec 1 à 500 clients concurrents. Ça nous permet de conclure que le service web peut répondre au maximum à environ 40 requêtes par seconde. Ce seuil est atteint avec 25 clients simultanés et reste fixe même si on augmente le nombre des clients.

Nous avons enregistré le nombre d'exceptions levées ainsi que le nombre de réponses erronées du service web. Par la suite, nous avons dressé la courbe de la disponibilité du service (voir la figure 4.4.d). Nous notons que l'invocation continue d'un client unique pendant 10 minutes déclenche 271 exceptions. Le service répond à moins de 80% des requêtes si le nombre de clients simultanés excède 100. Nous remarquons que la majorité des réponses erronées ont été rattachées à l'exception *connection refused*, qui signifie que la capacité du serveur d'application est dépassée.

Temps de réponse / Nombre de clients	<1sec	<2sec	<3sec	<4sec	<5sec	<6sec	<7sec	<8sec	<9sec	<10sec	≥10sec
1	99,97%	0,03%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
5	99,96%	0,02%	0,02%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
10	99,24%	0,63%	0,13%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
25	93,94%	3,84%	2,07%	0,15%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
50	54,27%	35,90%	6,82%	2,30%	0,55%	0,15%	0,01%	0,00%	0,00%	0,00%	0,00%
75	26,98%	57,63%	9,38%	4,44%	0,81%	0,29%	0,36%	0,09%	0,02%	0,00%	0,00%
100	18,37%	50,80%	15,99%	9,25%	4,01%	1,24%	0,24%	0,06%	0,03%	0,01%	0,00%
200	18,22%	51,86%	11,52%	8,55%	6,32%	0,86%	0,56%	0,68%	0,44%	0,30%	0,69%
350	11,98%	50,60%	22,48%	7,97%	2,72%	1,02%	0,54%	0,60%	0,41%	0,53%	1,15%
500	11,06%	45,43%	24,93%	6,01%	2,76%	1,46%	0,34%	0,25%	0,08%	0,09%	7,59%

TAB. 4.3 – La scalabilité du service web *ConfSearch*

Nous avons classifié le temps de réponse dans le tableau 4.3 dans différents intervalles (en seconde) : [0, 1] contient des requêtes qui ont pris moins d'une seconde, [1, 2] inclut des requêtes qui ont pris entre 1 et 2 secondes, etc. Nous avons divisé le

nombre de requêtes de chaque intervalle par le nombre total des requêtes réussies afin d'obtenir un résultat en pourcentage. Ces pourcentages représentent la scalabilité du service web *ConfSearch*.

Les trois premières lignes du tableau 4.3 montrent respectivement le résultat d'exécution de 1, 5 et 10 clients simultanés. Environ 100% des requêtes sont servies dans un temps de réponse moins d'1 seconde. Quand on dépasse 50 clients, le service web souffre du grand nombre de requêtes simultanées et ralentit son temps de réponse. Par exemple, environ 80% des requêtes sont accomplies dans un temps de réponse supérieur à 1 seconde et ce pour 100 et 200 clients simultanés. Nous remarquons que la performance se dégrade autant que le nombre de clients accroît et la disponibilité décroît jusqu'à atteindre 20% (voir la figure 4.4.d).

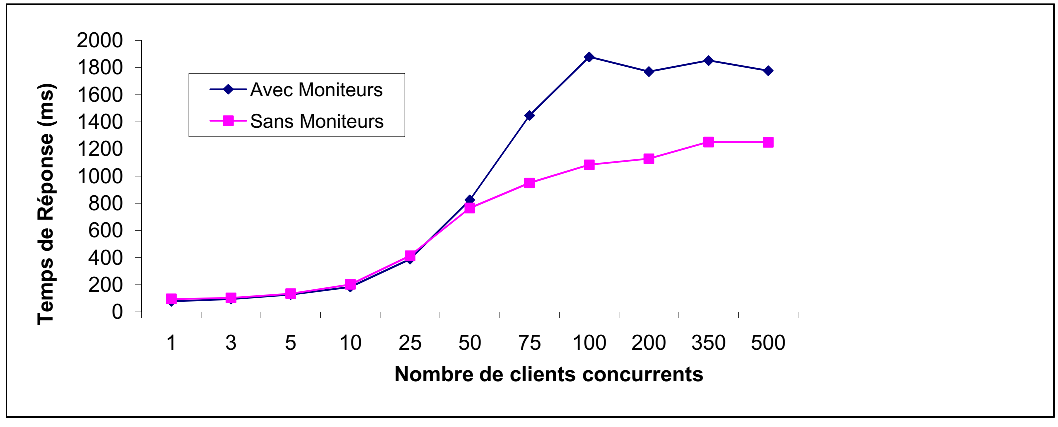

Fig. 4.5 – La surcharge des moniteurs

Dans la figure 4.5, nous avons dressé deux courbes du temps de réponse. Dans la première, la mesure est accomplie avec nos moniteurs. Dans la deuxième, la mesure est réalisée dans le code du client et sans utiliser des moniteurs. Avec moins de 50 clients simultanés, les deux courbes sont similaires et la surcharge des moniteurs est presque nulle. Le retard peut atteindre 0,5 seconde lorsque nous dépassons 50 clients simultanés. Ceci implique que *MARQ* est appropriée pour un service invoqué par moins de 50 clients simultanés. Si nous excédons 50 clients simultanés, nous devons tenir compte de la charge ajoutée par les moniteurs.

4.3 Le *FoodShop* : Un exemple de chaîne de commandes automatisées

L'application du *FoodShop* est étudiée dans le cadre du projet WS-DIAMOND (voir la figure 4.6). C'est une application basée sur les services web qui représente une compagnie de vente et de livraison de nourriture. Cette compagnie possède plusieurs services web en ligne, à savoir : le *Shop*, représentant la boutique, et plusieurs *WareHouses* (*WH1*, ..., *WHn*), représentant les entrepôts, qui sont situés dans différentes régions et qui sont responsables du stockage de la marchandise impérissable et de la livraison physiquement des produits vendus aux clients. En cas de produit périssable, qui ne peut pas être stocké (comme le lait), ou en cas de produit épuisé, le *WareHouse* interagit avec les services web *Suppliers* (*SUP1*, ..., *SUPm*) pour s'approvisionner.

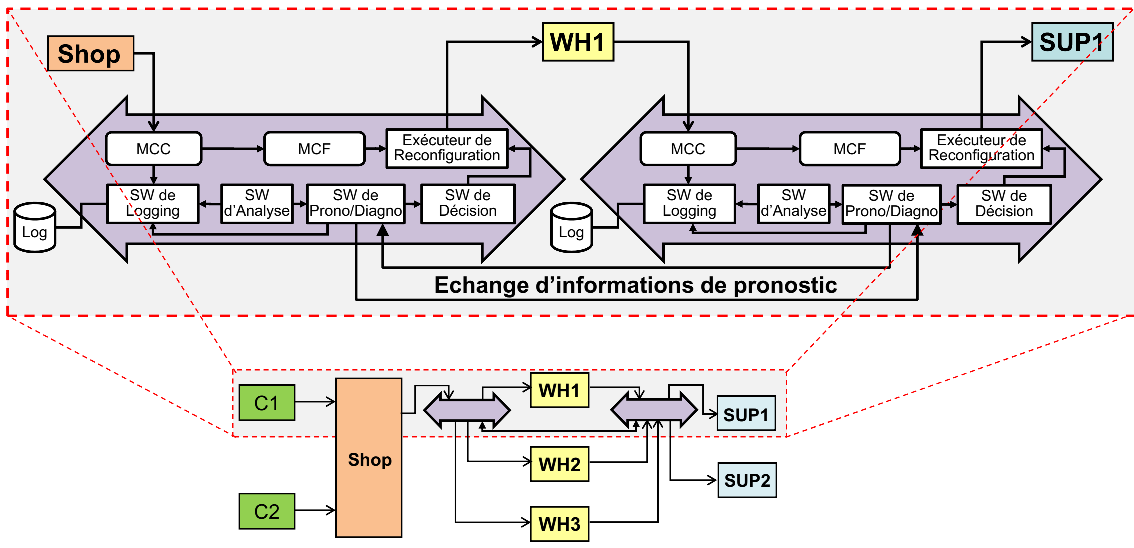

FIG. 4.6 – L'instanciation de *MARQ* avec l'application du *FoodShop*

Le *FoodShop* est une composition de services web interactifs. Il est implanté en utilisant BPEL (plus précisément *ActiveBPEL*[2] version 2.1). Le déploiement d'une instance des composants de *MARQ* avec cette application illustre des situations où des contraintes de déploiement doivent être prises en considération. Le client s'exécute au sein d'un processus BPEL, ce qui restreint le déploiement du *MCC* (pour les moniteurs niveau SOAP). Plus de détails concernant les contraintes de déploiement sont disponibles à la section 3.4.

[2]http ://www.activevos.com/

Une instance de *MARQ* est déployée entre chaque paire de client/fournisseur de SW de l'application *FoodShop*, qui est composée du *Shop*, des entrepôts *WH1*, *WH2* et *WH3* et des fournisseurs *SUP1* et *SUP2*, comme le montre la figure 4.6.

Les services de Diagnostic/Pronostic échangent des informations, afin de coordonner leurs actions d'auto-réparation. Par exemple, pour les deux services imbriqués *WH1* et *SUP1*, la dégradation de la QdS de *SUP1* peut se propager à *WH1* du point de vue du *Shop*. Cela déclenche un processus d'auto-réparation dans les deux instances déployées de *MARQ*. La première entre le *Shop* en tant que client, et le *WH1* en tant que SW, et la deuxième entre le *WH1* en tant que client et le *SUP1* en tant que SW. Si aucune coordination n'est implantée, chaque instance de *MARQ* substitue son fournisseur. La première substitue WH1 -qui est affecté par la propagation- et la deuxième substitue SUP -qui est dégradé-. Cependant, un raisonnement global sur la dégradation déduit que la dégradation de *WH1* est due au phénomène de propagation et que seulement *SUP1* doit être substitué (voir raisonnement à la section 2.5.1.1).

4.3.1 Hypothèses d'implantation

L'hypothèse d'implantation du côté client se résume à la modification de l'URL pour toutes les requêtes partantes du client. Cette modification remplace l'adresse IP et le port du serveur hébergeant le service web par l'IP et le port d'écoute du proxy. Par la suite, toutes les requêtes seront envoyées au proxy qui prendra en charge la gestion de la QdS, par des extensions du message reçu, et l'acheminement de la requête vers le *SW du fournisseur*.

Côté serveur, on configure le tomcat du *fournisseur du SW* pour qu'il envoie les réponses au proxy. Pour ce faire, on ajoute le code suivante dans le fichier de configuration du serveur *catalina* qui se trouve sous le *$TOMCAT_HOME$* (*catalina.sh* pour Linux/Unix, et *catalina.bat* pour Windows) :

```
/*************************************/
JAVA_OPTS= -Dhttp.proxyHost=192.168.2.210 -Dhttp.proxyPort=8080
-DproxySet=true
/*************************************/
```

Avec *192.168.2.210* qui désigne l'adresse IP de la machine hébergeant le proxy et *8080* le port de réception du proxy.

4.3.2 Implantation

L'instanciation de *MARQ* avec l'application du *FoodShop* donne lieu à un système de monitoring au niveau HTTP capable d'enregistrer des données telles que l'adresse IP de la machine de déploiement, les noms des services web communicants, les noms des opérations invoquées, la durée d'exécution ainsi que le type de la communication à savoir synchrone ou asynchrone. Ces informations permettent la découverte automatique et dynamique de toutes les parties impliquées dans l'application ainsi que les échanges qui avaient lieu entre elles. Il s'agit d'une reconstruction du profil de l'application.

En plus des différentes fonctionnalités d'auto-réparation, l'application de *FoodShop* a été enrichie par une interface graphique de monitoring. En effet, en se basant sur les données recueillies du monitoring, nous dressons dynamiquement un graphe de visualisation (i) des machines hôtes tout en les associant aux services web hébergés, (ii) des opérations cibles d'invocation par les services web, (iii) d'une estimation statistique de l'état de l'opération, ainsi que (iv) de la séquence d'invocation des opérations. Par conséquence, nous obtenons deux parties déployées avec l'application :

- *MARQ* : c'est une instance de notre middleware d'auto-réparation en se basant sur les proxies HTTP. Elle permet le monitoring, le logging, ainsi que la reconfiguration en terme de reroutage de requêtes. Cette partie utilise deux tables de base de données : une pour les données du monitoring, et l'autre pour le routage.
- *L'interface graphique de monitoring et d'analyse de la QdS* : elle comporte deux volets (voir figure 4.7). Celui du côté gauche extrait les données du log, et construit graphiquement le profil de l'application. Le volet de droite montre une analyse statistique de l'état des services web impliqués, en se basant sur les *chaînes de Markov cachées* (voir section 2.5.1.2). Cette partie utilise une table de base de données pour l'estimation des états des services web.

L'expérimentation du *FoodShop* a été réalisée sur plusieurs machines distribuées. Nous avons installé et configuré cinq machines indépendantes, à savoir : *M1* pour le *Shop*, *M2* pour le *Warehouse*, *M3* pour le *Supplier*, *M4* pour le deuxième *Warehouse* (utilisé pour la substitution) et *M5* pour le Proxy HTTP (pour l'interception, le logging et le reroutage). Sur une autre machine, nous avons lancé un générateur de requêtes périodiques qui joue le rôle du client. Les requêtes sont construites au hazard, en se basant sur la liste des produits disponibles auprès du *Shop*. Le visualisateur graphique (voir figure 4.7) est déployé aussi sur cette dernière machine

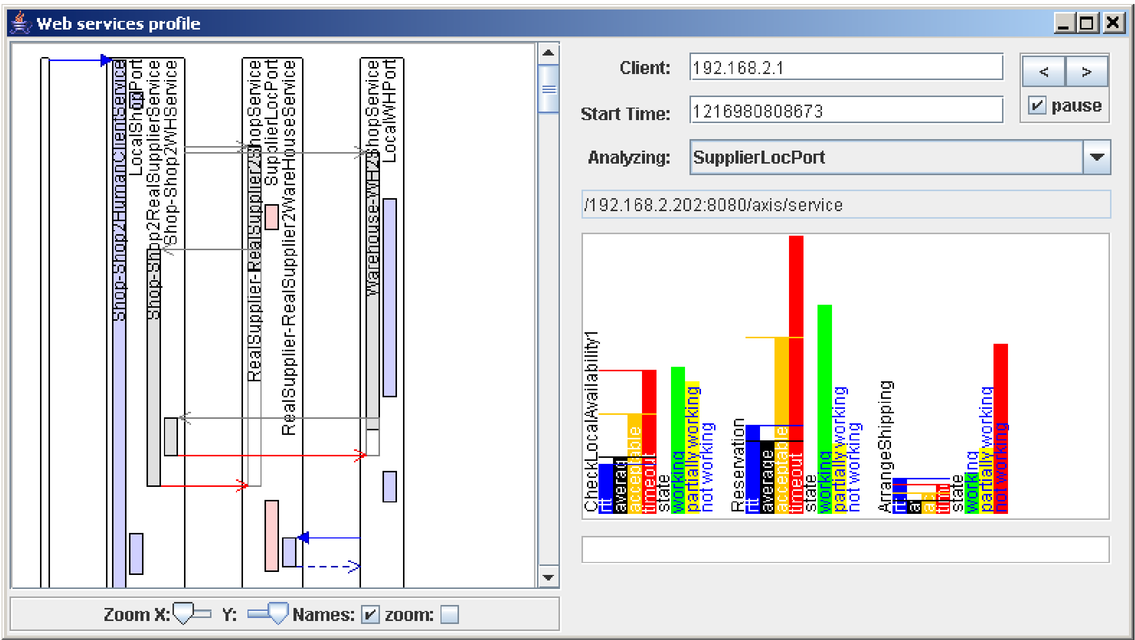

FIG. 4.7 – Le visualisateur graphique appliqué au *FoodShop*

ainsi qu'un injecteur de retard utilisé pour l'injection de dégradation et la vérification de la réaction du système de gestion de la QdS.

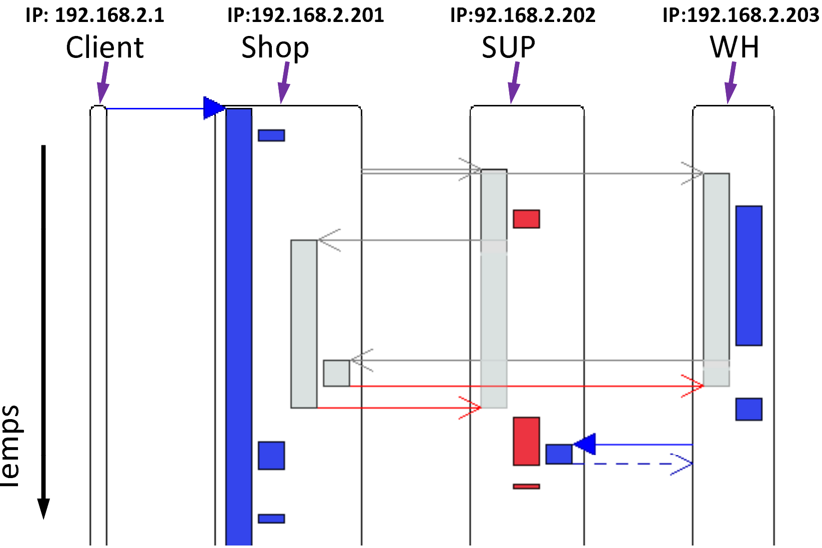

FIG. 4.8 – L'historique des conversations

La figure 4.8 focalise sur les détails du volet gauche du visualisateur graphique. Elle représente une reprise graphique des différentes interactions entre les services web

de l'application du *FoodShop*. Dans cette figure, chaque boxe (grand rectangle vide) représente un nœud de déploiement, ainsi que les services web hébergés dedans. Pour le cas du *FoodShop*, les boxes représentent les machines du client, du *Shop*, du *Supplier* et du *Warehouse*. Chaque ligne fléchée désigne un échange de messages entre les opérations. Une colonne pleine symbolise l'exécution d'une opération. Une colonne pleine non liée à des flèches signifie que l'opération est invoquée localement. La couleur de cette colonne reflète le type de communication utilisée pour cette invocation. Le rectangle bleu désigne les opérations synchrones. Les communications asynchrones sont signalées par les rectangles gris. Un rectangle rouge signifie que l'opération du service web est actuellement sélectionnée pour l'analyse statistique dans le volet droit (voir figure 4.9).

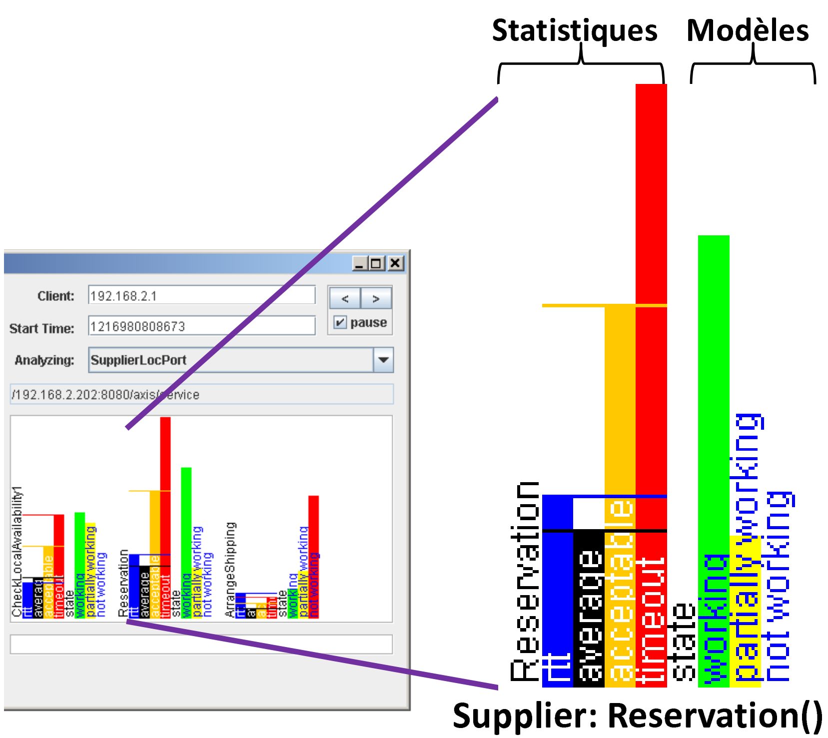

Fig. 4.9 – Les opérations statistiques

La figure 4.9 montre le volet droit du visualisateur graphique. Dans la partie supérieure, on spécifie l'adresse *IP* du client cible par l'analyse. Le champ dessous, affiche les différents départs des requêtes vers l'opération sélectionnée pour l'analyse. Cette sélection est garantie par une liste de choix disponible pour l'utilisateur du visualisateur. La liste de choix est située sous le champ du temps de départ de la requête.

La partie inférieure contient les statistiques et le modèle de pronostic de l'état des opérations du service web sélectionné. Le nom de l'opération est affiché à droite, suivi des calculs statistiques et des modèles de pronostic (voir section 2.5.1.2). Concernant les statistiques, la première colonne -en bleu- désigne la dernière valeur du paramètre de QdS mesurée par le monitoring. La deuxième colonne -en noir- indique la valeur moyenne de ce paramètre (voir équation 2.5). La troisième colonne -en orangé- présente le seuil de l'acceptance (voir équation 2.3). La dernière colonne -en rouge- calcule le seuil maximal possible (voir équation 2.2).

Concernant le modèle, il estime l'état de toutes les opérations du service choisi à travers les *chaînes de Markov cachées*, tout en se basant sur les statistiques déjà calculées (voir figure 2.8). La première colonne -en vert- représente l'état *Working*. La deuxième colonne -en jaune- désigne l'état *Partially Working*. La troisième colonne -en rouge- indique l'état *Not Working*. La colonne la plus haute entre les trois, désigne l'état de l'opération. Par exemple dans la figure 4.9, l'état de l'opération *Reservation* est *Working*.

4.3.3 La réparation

La réparation est implantée à deux niveaux : HTTP et SOAP.

4.3.3.1 Le connecteur de reconfiguration niveau HTTP

Une estimation de dégradation de la QdS, déclenche une opération de reconfiguration assurée par le proxy HTTP. La décision de réparation est suivie soit par la substitution du service dégradé, soit par la substitution de l'opération dégradée. En effet, le monitoring enregistre le nom de l'opération invoquée pour chaque requête, et ceci permet de raisonner sur la QdS offerte par une opération bien spécifique. Le plan de reconfiguration est équivalent à une requête *SQL* structurée comme suit :

```
/***************************************/
INSERT INTO PLAN SET SERVICE="old_wsdl_address", ACTION="old_operation",
NEWSERVICE="new_wsdl_address", NEWACTION="new_Operation";
/***************************************/
```

Avec *old_wsdl_address* dénotant le *WSDL* du service défaillant (où le service contenant l'opération défaillante, *old_operation* dénotant le nom de l'opération dégradée (champ à laisser vide en cas de substitution d'un service), *new_wsdl_address* déno-

tant le *WSDL* du service cible, et *new_ Operation* le nom de l'opération cible (champ à laisser vide en cas de substitution d'un service).

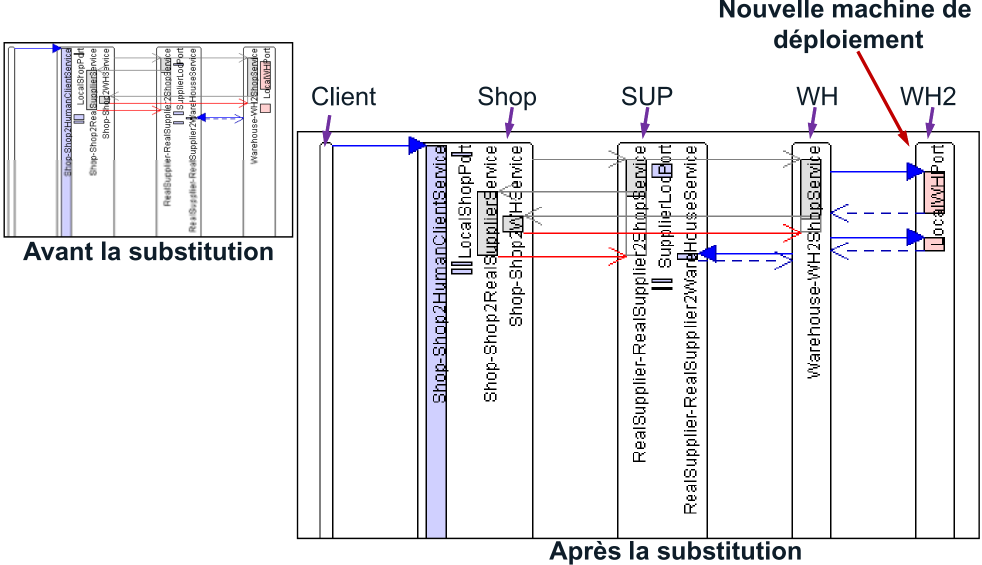

FIG. 4.10 – Le reroutage des requêtes

La figure 4.10 montre comment la liaison avec le service web du *WareHouse* (*LocalWHPort*) est reroutée vers un nouveau *Warehouse* (WH2) déployé sur une nouvelle machine. Dans la partie *Avant la substitution*, le *WH2* ne figure pas. Suite à la reconfiguration, les données de monitoring interprétée par le visualisateur graphique, affiche un nouveau *swimlane*. Il représente le nouveau *WareHouse* (*WH2*) impliqué dans l'application.

4.3.3.2 Le connecteur de reconfiguration niveau SOAP

Une version de *MARQ* niveau SOAP a été développée et instanciée avec l'application du *FoodShop*. La figure 4.11 détaille le développement de ce prototype[3].

On considère, à gauche, le client qui envoie des requêtes au *Shop1* qui représente le service web situé à droit. Entre le client et le fournisseur, nous déployons une instance de *MARQ*. Nous utilisons un *Injecteur de Retard* pour simuler une dégradation de la QdS. De suite, les services d'*Analyse* et de *Diagnostic* découvrent et identifient la dégradation. Le service de *Reconfiguration* substitue *Shop1* par *Shop2*, suite à une décision de recouvrement, d'une façon transparente au client. Nous utilisons

[3]Démonstration disponible sur http ://www.laas.fr/~khalil/TOOLS/QoS-4-SHWS/index.html

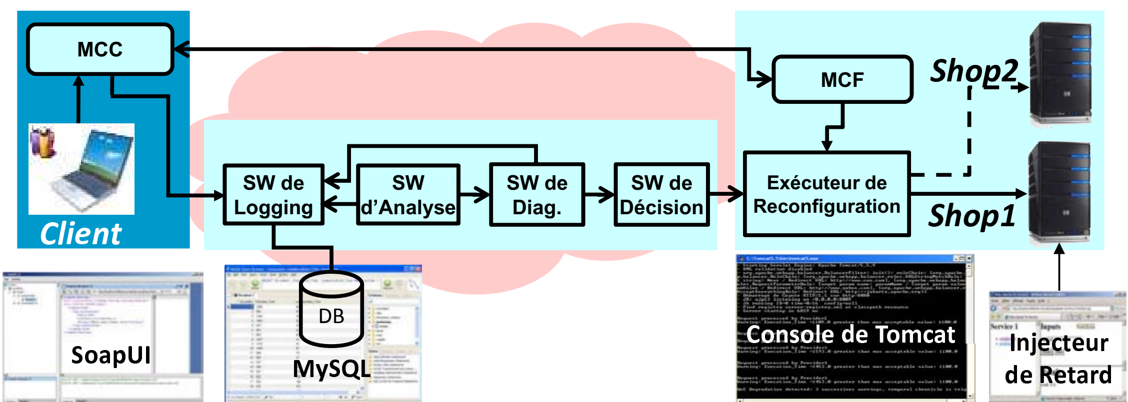

FIG. 4.11 – Le prototype de *FoodShop*

Apache Tomcat5.9 comme serveur de web, Axis1.4 comme conteneur de service web, ActiveBPEL2.1 comme moteur de BPEL, SoapUI1.5 comme client, MySQL5 comme système de gestion de base de données pour le logging et Java comme langage de programmation.

Nous avons calculé la valeur de N (voir la section 2.4) afin d'estimer au bout de combien de violations successives nous déclenchons une opération de substitution. Les valeurs et les probabilités utilisées et les chances sont déduites à partir des données de QdS issues du monitoring.

$P[any\ state \rightarrow TrespV] = 0.78 \times 0.13 + 0.23 \times 0.09 = 0.1221$
Avec : $P[TrespOK] = 0.78$, $P[TrespV] = 0.23$, $P[TrespOK \rightarrow TrespV] = 0.13$,
et $P[TrespV \rightarrow TrespV] = 0.09$.

$\Leftrightarrow 0.1221 \times 0.09^{N-1} \leq 0.04$, *Avec la disponibilité*$= 0.96$.
$\Leftrightarrow N \geq 2.115$

Ce qui signifie que N doit être au moins égale à 3. Autant nous accroissons la valeur de N, autant la précision de détection de dégradation augmente. Toutefois, le choix d'une valeur de N très supérieure à la valeur minimale, peut empêcher la détection de quelques dégradations.

La figure 4.12 illustre l'évolution du temps de réponse du service suite à une dégradation. Le service de monitoring de *MARQ* surveille l'évolution du temps de réponse au cours de l'exécution de *Shop1*. Au bout de douze invocations, une dégradation est détectée selon la chronique montrée dans la figure 2.4 avec N=3. La partie verte

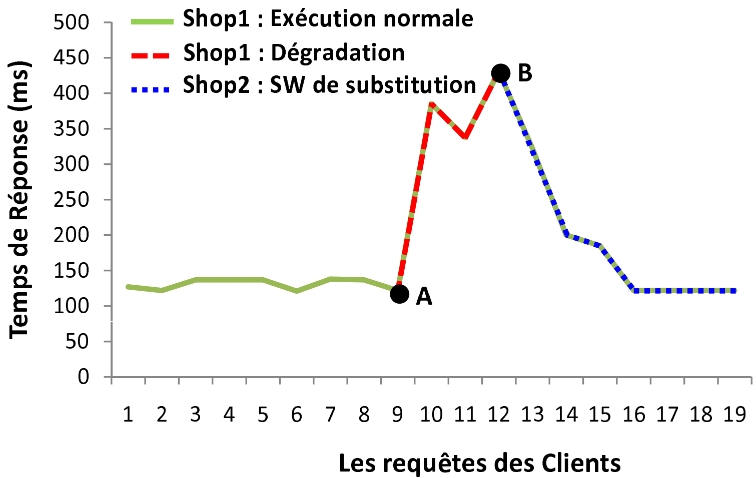

Fig. 4.12 – Le recouvrement de dégradation

de la courbe représente le fonctionnement normal du *Shop1*. Au niveau du point **A**, nous injectons un retard (un temps additionnel au moment de l'exécution du service et pour toutes les requêtes) et nous suivons le comportement de *MARQ*. Après trois violations successives, ce dernier détecte la dégradation (la partie rouge de la courbe dans la figure 4.12), et réagit par la substitution du service web *Shop1* au niveau du point **B**. Suite au reroutage des clients vers le nouveau *Shop* (*Shop2*), le temps de réponse regagne son comportement normal (la partie bleue de la courbe dans la figure 4.12).

4.4 Conclusion

Dans ce chapitre, nous avons mis en pratique notre approche présentée au deuxième et au troisième chapitres. Ceci a montré la faisabilité, et a prouvé l'efficacité de notre middleware avec différents types d'application. Le service de monitoring est validé avec une expérimentation à grande échelle et la charge des moniteurs reste négligeable tant que le nombre des clients simultanés ne dépasse pas 50. Le service d'analyse a réussi à détecter les violations de la QdS. Une fenêtre de visualisation du pronostic à travers les *chaînes de Markov cachées* a été implantée. Le *Connecteur de Liaison Dynamique* de reconfiguration est implanté de deux manières différentes : au niveau SOAP et au niveau HTTP. La reconfiguration niveau HTTP offre la possibilité d'appliquer une reconfiguration au niveau d'une opération spécifique d'un service web.

Conclusion Générale

La principale motivation qui a régit ce travail est le développement d'une architecture reconfigurable guidée par la QdS pour les applications coopératives à base de service web. Ce travail propose une architecture en bus logiciel au niveau application qui permet la construction d'applications réparties auto-réparables. Cette architecture repose sur des dispositifs, modèles et algorithmes couvrant le cycle d'auto-réparation. Elle a été instanciée et illustrée avec différents types de compositions de services web, à savoir l'orchestration et la chorégraphie.

Les principales contributions de cette thèse peuvent se résumer comme suit :

- Le monitoring non intrusif des messages SOAP dans les conteneurs de déploiement côté client et côté fournisseur du service web. Ceci est réalisé grâce à des moniteurs, générés et déployés automatiquement, et qui marquent et collectent les valeurs des paramètres de QdS. Ces moniteurs prennent en charge les spécificités des communications synchrones et asynchrones. Les expérimentations menées à grande échelle sur la plate-forme *Grid'5000*, montrent que la charge de nos moniteurs est presque nulle tant que le nombre de clients ne dépasse pas les 50 clients concurrents.
- La détection et l'identification de la source de dégradation de la QdS. Ceci correspond à la phase d'exploitation des valeurs obtenues par le monitoring. Cette fonction permet de s'assurer du bon fonctionnement de l'application, et le cas échéant déclencher des alarmes. Nous focalisons sur la surveillance de l'évolution d'une caractéristique donnée de QdS plus que sur ses valeurs absolues pour distinguer entre une dégradation transitoire et une dégradation permanente. Ce processus se poursuit par l'identification de la source de dégradation. Des fonctionnalités de détection de la propagation de dégradation sont élaborées pour éliminer les actions de reconfiguration inutiles.
- L'élaboration d'une interface graphique pour le monitoring et l'analyse de la QdS. Elle permet la reconstruction des interactions entre les services web impliqués

dans l'application sous forme de diagramme de séquences. Aussi, cette interface visualise les résultats d'une analyse probabiliste basée sur les modèles *Markoviens*, et reflétant une estimation des états des services web.
- La reconfiguration dynamique pour la prévention et la rectification de la dégradation de la QdS. Ceci est mis en œuvre grâce à un *Connecteur de Liaison Dynamique*, généré et déployé automatiquement. Celui-ci permet le reroutage des requêtes des clients vers un service offrant de meilleures QdS. La granularité d'une action de reconfiguration varie de la substitution d'une simple opération, à la substitution entière d'un service.

Toutefois, notre approche présente quelques limites. D'une part, elle ne traite que les services web sans état, et elle a besoin d'une extension pour traiter ceux avec état. D'autre part, les modèles élaborés ne sont pas adaptés pour traiter les interruptions brusques des services web. Notre middleware ne réagit qu'après des signes de dégradation qui précèdent l'état de dysfonctionnement.

Les perspectives des travaux présentés dans ce mémoire suivent différents axes de réflexion et se situent dans différents contextes de recherche.

À court terme, nous prévoyons l'extension des moniteurs, afin de les rendre paramétrables à travers une interface graphique d'administration. Ces extensions permettent la prise en considération de nouvelles QdS (telles que la réputation, le prix, la robustesse) en plus de celles déjà traitées dans ce travail.

Nous planifions aussi, de conforter l'utilisateur à travers une automatisation de la génération du code des moniteurs à partir d'une description détaillée par annotation sémantique. Les attributs de QdS, ainsi que les fonctions de calcul des valeurs des paramètres de QdS seront exprimés via des annotations sémantiques et enrichiront la description du service web. La spécification SAWSDL (*Semantic Annotations for WSDL*) permet l'extension du WSDL par des annotations sémantiques. Un *parseur* sera mis au point afin de parcourir les descriptions WSDL, pour générer et pour déployer automatiquement le code des moniteurs correspondants.

À court terme, nous planifions d'exploiter les données de monitoring de la QdS (niveau HTTP) pour la reconstruction de la structure d'interaction entre les différents services web composant l'application. En effet, ces moniteurs enregistrent des informations telles que l'adresse IP de la machine de déploiement, les noms des services web communicants, les noms des opérations invoquées, la durée d'exécution ainsi que le type de la communication à savoir synchrone ou asynchrone. En se basant sur ces données, nous avons réussi à élaborer un outil qui reconstruit tous les profils

d'interaction d'une composition de services web en cours d'exécution. Un de nos objectifs est de se servir de ce profil afin d'évaluer l'architecture en cours d'exécution (réelle) avec l'architecture conçue avec les graphes d'architecture. Une telle approche permettrait de vérifier la conformité des instances d'architecture à leurs modèles par différenciation. Ceci permettrait éventuellement de produire les actions de reconfiguration à exécuter pour recouvrir la déformation architecturale. En plus, nous visons à coordonner les actions au niveau architectural et au niveau exécution afin d'éviter des sur-réactions de l'application ou l'exécution d'actions qui s'opposent ou s'annihilent.

À moyen terme, nous visons l'intégration d'un service de découverte de services web. Il prendra en charge la recherche d'un service offrant des fonctionnalités équivalentes à celui détecté défaillant. La découverte doit inclure des techniques ciblant les aspects fonctionnels -décrivant l'aspect logique- et les aspects non fonctionnels -décrivant l'aspect qualitatif-. La recherche doit, aussi, prendre en considération l'historique de la QdS des services utilisés pour guider le processus de sélection. Ceci permettra, par exemple, d'approuver l'utilisation d'un service qui a déjà offert une QdS satisfaisant les besoins des clients et d'éviter l'utilisation d'un service déjà identifié comme dégradé, sauf si son fournisseur a déployé et rendu disponible une nouvelle version.

Nous envisageons aussi, l'implantation des procédures d'intégration entre l'auto-réparation de niveaux classes et instance (voir section 3.8). Le but est (i) de gérer le temps d'indisponibilité temporaire durant la phase de déploiement d'un nouveau *Connecteur de Liaison Dynamique* et (ii) de coordonner les actions de reconfiguration entre les mécanismes des deux niveaux.

La version actuelle de notre middleware gère l'auto-réparation des services web sans état. Il est possible de prendre en considération la gestion de la QdS des services web avec état. Des études sur la mobilité du code sont programmées à moyen terme. En effet, la mobilité considère la migration du code du service tout en mémorisant son état ainsi que le dernier point de contrôle (*checkpoint*). Suite à son déploiement sur le nouveau site, le service reprend son exécution à partir de ce dernier point. La mobilité peut aussi considérer seulement l'état du service. En d'autres termes, elle copie l'état du service défaillant et le transmet vers le nouveau service cible.

Dans un autre cadre, nous envisageons, à moyen terme, d'optimiser la QdS d'une composition de services web. Pour ce faire, un système de gestion de priorités peut être mis en place et implanté dans une partie tierce entre le client et le service. Il

assurera le rôle d'un ordonnanceur. Par exemple, d'après les expérimentations faites sur la plate-forme *Grid'5000*, le temps d'exécution varie selon le débit actuel du service web (nombre de requêtes simultanées entrain d'être traitées). Par conséquent, nous pouvons minimiser le temps de réponse tout en diminuant le nombre de services s'exécutant en parallèle (abaisser le débit pour une vitesse d'exécution plus rapide). Dans ce cas, la notion de priorité décide de l'ordonnancement des requêtes des clients.

Publications

Revue Scientifique

[J1] Riadh Ben Halima, Karim Guennoun, Mohamed Jmaiel, and Khalil Drira. Providing Predictive Self-Healing for Web Services : A QoS Monitoring and Analysis-based Approach. *Journal of Information Assurance and Security (JIAS)*. Vol.3, N°3, pp.175-184, Septembre 2008.

Manifestations internationales avec actes

[C1] Riadh Ben Halima, Emna Fki, Mohamed Jmaiel, and Khalil Drira. Experiments Results and Large Scale Measurement Data for Web Services Performance Assessment. *To appear in the 1st IEEE Workshop on Performance evaluation of communications in distributed systems and Web based service architectures*, July 5-8, 2009, Sousse, Tunisia, 6pages. IEEE Computer Society. (taux de sélection : 37%)

[C2] Riadh Ben Halima, Mohamed Jmaiel, and Khalil Drira. A QoS-Oriented Reconfigurable Middleware For Self-Healing Web Services. *In the 6th IEEE International Conference on Web Services (ICWS'2008)*, September 23-26, 2008, Beijing, China, pages 104-111. IEEE Computer Society. (taux de sélection : 16%)

[C3] Olga Nabuco, Riadh Ben Halima, Khalil Drira, Maria Grazia Fugini, Stefano Modafferi, and Enrico Mussi. Model-based QoS-enabled self-healing Web Services. *1st International Workshop on Data Management in Virtual Engineering (DM-VE'08)*, September 1-5, 2008, Turin, Italy, pages 711-715. IEEE Computer Society.

[C4] Riadh Ben Halima, Karim Guennoun, Mohamed Jmaiel, and Khalil Drira. Non-intrusive QoS Monitoring and Analysis for Self-Healing Web Services. *In the The First IEEE International Conference on the Applications of Digital Information and Web Technologies (ICADIWT'2008).* August 4-6, 2008. Ostrava, Czech, pages 549-554. IEEE Computer Society. (taux de sélection : 102/250)

[C5] René Pegoraro, Riadh Ben Halima, Khalil Drira, Karim Guennoun, and Joao Mauricio Rosrio. A framework for monitoring and runtime recovery of web service-based applications. *In 10th International Conference on Enterprise Information Systems (ICEIS'2008)*, 12-16 June 2008, Barcelona, Spain. Pages 201-206.

[C6] Riadh Ben Halima, Mohamed Jmaiel, and Khalil Drira. A QoS-driven reconfiguration management system extending web services with self-healing properties. *In 16th IEEE International Workshops on Enabling Technologies : Infrastructures for Collaborative Enterprises (WETICE'2007) : Workshop on Information Systems δ Web Services (ISWS)*, 18-20 June 2007, Paris (France), pages 339-344. IEEE Computer Society. (taux de sélection : 6/20)

[C7] Riadh Ben Halima, Mohamed Jmaiel, et Khalil Drira. Une approche orientée règle pour la spécification formelle des architectures dynamiquement configurables. *7ème Conférence Internationale sur les NOuvelles TEchnologies de la REpartition (NOTERE'2007)*. 4-8 Juin, 2007, Marrakech, Maroc. Pages 405-412.

[C8] Riadh Ben Halima, Mohamed Jmaiel, and Khalil Drira. Graphical simulation of the dynamic evolution of the software architectures specified in Z. *In 8th International Workshop on Principles of Software Evolution (IWPSE'2005)*, September 5-6, 2005, Lisbon, Portugal, pages 45-48. IEEE Computer Society.

Rapports techniques

[RT1] Riadh Ben Halima, Khalil Drira, Karim Guennoun, and Francisco Moo-Mena. Specification of execution mechanisms and composition strategies for self-healing Web services -Phase 1-. Delivrable 3.1, *Projet IST WS-DIAMOND N°516933*, Février 2007, 102p.

[RT2] Riadh Ben Halima, Khalil Drira, and Karim Guennoun. Specification of execution mechanisms and composition strategies for self-healing Web services -Phase 2-. Delivrable 3.2, *Projet IST WS-DIAMOND N°516933*, Février 2008, 63p.

Bibliographie

[1] Mehmet Aksit and Bedir Tekinerdogan. Aspect-oriented programming using composition-filters. In *ECOOP '98 : Workshop ion on Object-Oriented Technology*, page 435, London, UK, 1998. Springer-Verlag.

[2] Mani Anbazhagan and Nagarajan Arun. Understanding quality of service for web services. Technical report, IBM developerWorks., 2002.

[3] William W. Hines (and Douglas C. Montgomery. *Probability and Statistics in Engineering and Management Science.* John Wiley and Sons, 2nd edition, 1980.

[4] Tony Andrews and al. *Business Process Execution Language for Web Services Version 1.1.* IBM, http ://www.ibm.com/developerworks/library/ws-bpel, May 2003.

[5] S. Araban and L. S. Sterling. Measuring quality of service for contract aware web services. In *First Australian Workshop on Engineering Service-Oriented Systems*, pages 54–56, 2004.

[6] D. Ardagna, C. Cappiello, M.G. Fugini, E. Mussi, B. Pernici, and P. Plebani. Faults and recovery actions for self-healing web services. In *WWW2006, May 2006, Edinburgh, UK.*, 2005.

[7] Danilo Ardagna, Marco Comuzzi, Enrico Mussi, Barbara Pernici, and Pierluigi Plebani. Paws : A framework for executing adaptive web-service processes. *IEEE Software*, 24(6) :39–46, 2007.

[8] Ashish Agarwal Asaf Arkin. *The BPML specification.* BPML Working Draft 0.4, http ://www.bpmi.org, March 2001.

[9] Daniel Austin, Abbie Barbir, Christopher Ferris, and Sharad Garg. *Web Services Architecture Requirements.* W3C, http ://www.w3.org/TR/wsa-reqs, February 2004.

[10] Gilbert Babin and Michel Leblanc. Les web services et leur impact sur le commerce b2b. CIRANO Burgundy Reports 2003rb-07, CIRANO, 2003.

[11] Arindam Banerji, Claudio Bartolini, Dorothea Beringer, Venkatesh Chopella, Kannan Govindarajan, Alan Karp, Harumi Kuno, Mike Lemon, Gregory Pogossiants, Shamik Sharma, and Scott Williams. *Web Services Conversation Language (WSCL) 1.0*. W3C, http ://www.ebxml.org/specs/ebBPSS.pdf, March 2002.

[12] Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS '06 : Proceedings of the IEEE International Conference on Web Services*, pages 63–71, Washington, DC, USA, 2006. IEEE Computer Society.

[13] Luciano Baresi and Sam Guinea. Towards dynamic monitoring of ws-bpel processes. In *3rd International Conference on Service Oriented Computing, 2005, Proceedings*, volume 3826 of *Lecture Notes in Computer Science*, pages 269–282. Springer, 2005.

[14] Gordon S. Blair, Geoff Coulson, Lynne Blair, Hector Duran-Limon, Paul Grace, Rui Moreira, and Nikos Parlavantzas. Reflection, self-awareness and self-healing in openorb. In *WOSS '02 : Proceedings of the first workshop on Self-healing systems*, pages 9–14, New York, NY, USA, 2002. ACM Press.

[15] David Booth, Hugo Haas, Francis Mccabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture. Technical report, W3C, Web Services Architecture Working Group, February 2004.

[16] S. Bouchenak, F. Boyer, D. Hagimont, S. Krakowiak, A. Mos, N. Depalma, V. Quema, and J.-B. Stefani. Architecture-based autonomous repair management : An application to j2ee clusters. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS)*, Orlando, Florida, USA, October 2005.

[17] Eric Bruneton and Michel Riveill. Experiments with javapod, a platform designed for the adaptation of non-functional properties. In *REFLECTION '01 : Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 52–72, London, UK, 2001. Springer-Verlag.

[18] Franck Cappello, Frederic Desprez, Michel Dayde, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Nouredine Melab, Raymond Namyst, Pascale Primet, Olivier Richard, Eddy Caron, Julien Leduc, and Guillaume Mornet. Grid'5000 : A large scale, reconfigurable, controlable and monitorable grid

platform. In *Grid2005 6th IEEE/ACM International Workshop on Grid Computing*, 2005.

[19] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *ESOP07*, pages 2–17. Springer.

[20] Ethan Cerami. *Web Services Essentials.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[21] Shang-Wen Cheng, David Garlan, and Bradley R. Schmerl. Making self-adaptation an engineering reality. In *Self-star Properties in Complex Information Systems, Conceptual and Practical Foundations [the book is a result from a workshop at Bertinoro, Italy, Summer 2004]*, volume 3460 of *Lecture Notes in Computer Science*, pages 158–173. Springer, 2005.

[22] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley R. Schmerl, and Peter Steenkiste. An architecture for coordinating multiple self-management systems. In *4th Working IEEE / IFIP Conference on Software Architecture (WICSA 2004), 12-15 June 2004, Oslo, Norway*, pages 243–254. IEEE Computer Society, 2004.

[23] David M. Chess, Alla Segal, Ian Whalley, and Steve R. White. Unity : Experiences with a prototype autonomic computing system. In *1st International Conference on Autonomic Computing (ICAC 2004), 17-19 May 2004, New York, NY, USA*, pages 140–147. IEEE Computer Society, 2004.

[24] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) Version 1.2.* W3C, http ://www.w3.org/TR/wsdl, March 2001.

[25] F. Cohen. Performance testing soap-based applications : Is your web service production-ready ? Technical report, IBM developerWorks., November 2001.

[26] OMG Consortium. Unified modeling language : Superstructure. Technical Report Version 2.0, Object Management Group, http ://www.omg.org/docs/ptc/04-10-02.pdf, October 2004.

[27] Francisco Curbera, William A. Nagy, and Sanjiva Weerawarana. Web services : Why and how. In *In OOPSLA 2001 Workshop on Object-Oriented Web Services. ACM*, 2001.

[28] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In *WOSS '02 : Proceedings of the first workshop on Self-healing systems*, pages 21–26, New York, NY, USA, 2002. ACM Press.

[29] Bradley Schmerl David Garlan, Shang-Wen Cheng. Increasing system dependability through architecture-based self-repair. *Appears in Architecting Dependable Systems*, 2003.

[30] Noël de Palma, Sara Bouchenak, Daniel Hagimont, Sylvain Sicard, , and Christophe Taton. Jade : Un Environnement d'Administration Autonome. *Techniques et Sciences Informatiques*, 27(9-10) :1225–1252, 2008.

[31] Kiran Devaram and Daniel Andresen. Soap optimization via parameterized client-side caching. In *International Conference on Parallel and Distributed Computing Systems*, pages 785–790, November 2003.

[32] Ada Diaconescu. A framework for using component redundancy for self-adapting and self-optimising component-based enterprise systems. In *OOPSLA '03 : Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 390–391, New York, NY, USA, 2003. ACM Press.

[33] Glen Dobson, Russell Lock, and Ian Sommerville. Qosont : a qos ontology for service-centric systems. In *31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA 2005)*, pages 80–87. IEEE Computer Society, 2005.

[34] Christophe Dousson, Paul Gaborit, and Malik Ghallab. Situation recognition : Representation and algorithms. In *Thirteenth International Joint Conference on Artificial Intelligence (IJCAI 1993)*, 1993.

[35] Thomas Erl. *Service-Oriented Architecture : Concepts, Technology, and Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[36] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, Irvine, 2000. Chair-Richard N. Taylor.

[37] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02 : Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002. ACM Press.

[38] Selvin George, David Evans, and Steven Marchette. A biological programming model for self-healing. In *SSRS '03 : Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems*, pages 72–81, New York, NY, USA, 2003. ACM Press.

[39] Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. Self-healing systems - survey and synthesis. *Decis. Support Syst.*, 42(4) :2164–2185, 2007.

[40] Martin Gudgin and al. Simple object access protocol (soap) 1.2. Technical report, W3C Note 08, World Wide Web Consortium, May 2003.

[41] Martin Gudgin, Marc Hadley, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. *Simple Object Access Protocol (SOAP) Version 1.2*. W3C, http ://www.w3.org/TR/2001/WD-soap12-20010709/, July 2001.

[42] K. GUENNOUN. *Architectures dynamiques dans le contexte des applications à base de composants et orientées service*. PhD thesis, Université Paul Sabatier, Toulouse, 183p., 2006. Doctorat.

[43] Assia Hachichi, Cyril Martin, Gaël Thomas, Simon Patarin, and Bertil Folliot. Reconfigurations dynamiques de services dans un intergiciel à composants corbacm. In *1ère Conférence Francophone sur le Déploiement et la (Re) Configuration de Logiciels*, Grenoble, France, October 2004.

[44] Matti A. Hiltunen, Richard D. Schlichting, Carlos A. Ugarte, and Gary T. Wong. Survivability through customization and adaptability : the cactus approach. In *DARPA Information Survivability Conference and Exposition*, pages 294–306, 1999.

[45] Sun J., Tenhunen J., and Sauvola J. Cme : a middleware architecture for network-aware adaptive applications. In *14th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, volume 1, pages 839–843, Beijing, China, 2003. IEEE Computer Society.

[46] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 25(1) :157–187, 1995.

[47] Skene James, Lamanna D. Davide, and Emmerich Wolfgang. Precise service level agreements. In *ICSE '04 : Proceedings of the 26th International Conference on Software Engineering*, pages 179–188, Washington, DC, USA, 2004. IEEE Computer Society.

[48] O'Sullivan Justin, Edmond David, and Ter Hofstede Arthur. What's in a service ? *Distrib. Parallel Databases*, 12(2-3) :117–133, 2002.

[49] Barry Douglas K. *The Savvy Manager's Guide to Web Services and Service-Oriented Architectures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[50] Kyungran Kang, Jihwan Song, Jinhyung Kim, Heejung Park, and We-Duke Cho. Uss monitor : A monitoring system for collaborative ubiquitous computing environment. *IEEE Transactions on Consumer Electronics*, 53(3) :911–916, 2007.

[51] Nickolaos Kavantzas, David Burdett, Greg Ritzinger, Tony Fletcher, Yves Lafon, and Charlton Barreto. *Web Services Choreography Description Language Version 1.0.* W3C, http ://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/, November 2005.

[52] Alexander Keller and Heiko Ludwig. The wsla framework : Specifying and monitoring service level agreements for web services. *Journal of Network and System Management*, 11(1), 2003.

[53] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, 2003.

[54] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtierand, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241*, pages 220–242. Springer-Verlag, 1997.

[55] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *Middleware 2000, IFIP/ACM International Conference on Distributed Systems Platforms, New York, NY, USA, April 4-7, 2000, Proceedings*, volume 1795 of *Lecture Notes in Computer Science*, pages 121–143. Springer, 2000.

[56] P. Sampatakos L. Dimopoulou, E. Nikolouzou and I.Venieris. Qmtool : An xml-based management platform for qos aware ip networks. *IEEE Network*, 17(3) :8–14, 2003.

[57] D. Davide Lamanna, James Skene, and Wolfgang Emmerich. Slang : A language for defining service level agreements. In *FTDCS '03 : Proceedings of the The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'03)*, page 100, Washington, DC, USA, 2003. IEEE Computer Society.

[58] Kangchan Lee, Jonghong Jeon, Wonseok Lee, Seong-Ho Jeong, and Sang-Won Park. Qos for web services : Requirements and possible approaches. Technical report, W3C, Web Services Architecture Working Group, November 2003.

[59] Duran Limon. *A Resource Management Framework for Reflective Multimedia Middleware.* PhD thesis, Lancaster University, UK, October 2001.

[60] Dong Liu and Ralph Deters. Bust : enabling scalable service orchestration. In *InfoScale '07 : Proceedings of the 2nd international conference on Scalable information systems*, pages 1–10, ICST, Brussels, Belgium, Belgium, 2007. ICST

(Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[61] Giorgia Lodi, Fabio Panzieri, Davide Rossi, and Elisa Turrini. Sla-driven clustering of qos-aware application servers. *IEEE Trans. Softw. Eng.*, 33(3) :186–197, 2007.

[62] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P. King, and Richard Franck. Wsla language specification, version 1.0. Technical report, IBM Corporation, http ://www.research.ibm.com/wsla/, January 2003.

[63] Fredj Manel, Georgantas Nikolaos, Issarny Valerie, and Zarras Apostolos. Dynamic service substitution in service-oriented architectures. In *SERVICES '08 : Proceedings of the 2008 IEEE Congress on Services - Part I*, pages 101–104, Washington, DC, USA, 2008. IEEE Computer Society.

[64] Raphael Marvie and Marie-Claude Pellegrini. Etat de l'art des modèles de composants. Technical Report 2, CESURE RNRT Project, May 2000.

[65] N. Medvidovic and M. Mikic-Rakic. Programming-in-the-many : A software engineering paradigm for the 21st century. In *Workshop on New Visions for Software Design and Productivity : Research and Applications*, Nashville, Tennessee, December 2001.

[66] Daniel A. Menascé. Qos issues in web services. *IEEE Internet Computing*, 6(6) :72–75, 2002.

[67] Stal Michael. Web services : beyond component-based computing. *Commun. ACM*, 45(10) :71–76, 2002.

[68] Marija Mikic-Rakic, Nikunj Mehta, and Nenad Medvidovic. Architectural style requirements for self-healing systems. In *WOSS '02 : Proceedings of the first workshop on Self-healing systems*, pages 49–54, New York, NY, USA, 2002. ACM Press.

[69] Stefano Modafferi, Enrico Mussi, and Barbara Pernici. Sh-bpel : a self-healing plug-in for ws-bpel engines. In *MW4SOC '06 : Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, pages 48–53, New York, NY, USA, 2006. ACM.

[70] Francisco Moo-Mena and Khalil Drira. Reconfiguration of web services architectures : A model-based approach. In *12th IEEE Symposium on Computers and Communications (ISCC 2007), July 1-4,*, pages 357–362, Aveiro, Portugal, 2007. IEEE Computer Society.

[71] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Eternal : a component-based framework for transparent fault-tolerant corba. *Softw. Pract. Exper.*, 32(8) :771–788, 2002.

[72] Priya Narasimhan, Louise E. Moser, and P. m. Melliar-Smith. Using interceptors to enhance corba. *Computer*, 32(7) :62–68, 1999.

[73] Eric Newcomer and Greg Lomow. *Understanding SOA with Web Services (Independent Technology Guides)*. Addison-Wesley Professional, 2004.

[74] F. Ogel, B. Folliot, and I. Piumarta. On reflexive and dynamically adaptable environments for distributed computing. In *ICDCSW '03 : Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 112, Washington, DC, USA, 2003. IEEE Computer Society.

[75] OMG. Portable interceptors : Request for proposals. Technical report, Object Management Group, ftp :// ftp.omg.org/pub/docs/orbos/98-09-11.pdf, 1998.

[76] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3) :54–62, 1999.

[77] Karn P. and Partridge C. Improving round-trip time estimates. in reliable transport protocols. *ACM Transaction on Computer Systems*, 9(4) :363–373, 1991.

[78] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10) :46–52, 2003.

[79] Srivastava P.K. and Sahu S. Secured remote tracking of critical autonomic computing applications. published in IEEE E-Tech 2004, Karachi, Pakistan.

[80] Yang Qun, Yang Xian-Chun, and Xu Man-Wu. A framework for dynamic software architecture-based self-healing. *SIGSOFT Softw. Eng. Notes*, 30(4) :1–4, 2005.

[81] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Readings in speech recognition*, pages 267–296, 1990.

[82] Shuping Ran. A model for web services discovery with qos. *SIGecom Exch.*, 4(1) :1–10, 2003.

[83] Nicolas Repp, Rainer Berbner, Oliver Heckmann, and Ralf Steinmetz. A cross-layer approach to performance monitoring of web services. In *Proceedings of the Workshop on Emerging Web Services Technology (in conjunction with IEEE ECOWS 2006)*. CEUR-WS, Dec 2006.

[84] George R. Ribeiro-Justo and Tereska Karran. Modelling organic adaptable service-oriented enterprise architectures. In *On The Move to Meaningful Internet Systems 2003 : OTM 2003 Workshops, OTM Confederated International Workshops, HCI-SWWA, IPW, JTRES, WORM, WMS, and WRSM 2003, Catania, Sicily, Italy, November 3-7, 2003, Proceedings*, volume 2889 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2003.

[85] Florian Rosenberg, Christian Platzer, and Schahram Dustdar. Bootstrapping performance and dependability attributes ofweb services. In *ICWS '06 : Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, pages 205–212, Washington, DC, USA, 2006. IEEE Computer Society.

[86] Abdulmotaleb El Saddik. Performance measurements of web services-based applications. *IEEE Transactions on Instrumentation and Measurement*, 55(5) :1599–1605, October 2006.

[87] Douglas C. Schmidt and Chris Cleel. Applying patterns to develop extensible orb middleware. *IEEE Communications Magazine*, 37 :54–63, 1999.

[88] H. Schmidt. Trustworthy components-compositionality and prediction. *Journal of Systems and Software*, 65(3) :215–225, 2003.

[89] George Spanoudakis and Khaled Mahbub. Non-intrusive monitoring of service-based systems. *International Journal of Cooperative Information Systems*, 15(3) :325–358, 2006.

[90] Roy Sterritt and David F. Bantz. Pac-men : Personal autonomic computing monitoring environment. In *15th International Workshop on Database and Expert Systems Applications (DEXA 2004), wsith CD-ROM, 30 August - 3 September 2004, Zaragoza, Spain*, pages 737–741. IEEE Computer Society, 2004.

[91] Yehia Taher, Djamal Benslimane, Marie-Christine Fauvet, and Zakaria Maamar. Towards an approach for web services substitution. In IEEE, editor, *10th IEEE International Database Engineering and Applications Symposium (IEEE IDEAS 2006)*, dec 2006.

[92] Business Process Project Team. *ebXML Business Process Specification Schema 1.0*. UN/CEFACT OASIS, http ://www.ebxml.org/specs/ebBPSS.pdf, May 2001.

[93] The Axis Development Team. Axis architecture guide, 1.4 version, on-line at : http ://archive.apache.org/dist/ws/axis/1_4/, April 2006.

[94] Gerald Tesauro, David M. Chess, William E. Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O. Kephart, and Steve R. White. A multi-agent systems

approach to autonomic computing. In *3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004), 19-23 August 2004, New York, NY, USA*, pages 464–471. IEEE Computer Society, 2004.

[95] Rajesh K. Thiagarajan, Amit K. Srivastava, Ashis K. Pujari, and Visweswar K. Bulusu. Bpml : A process modeling language for dynamic business models. In *WECWIS '02 : Proceedings of the Fourth IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS'02)*, page 239, Washington, DC, USA, 2002. IEEE Computer Society.

[96] Niko Thio and Shanika Karunasekera. Automatic measurement of a qos metric for web service recommendation. In *ASWEC '05 : Proceedings of the 2005 Australian conference on Software Engineering*, pages 202–211, Washington, DC, USA, 2005. IEEE Computer Society.

[97] Vladimir Tosic, Bernard Pagurek, Kruti Patel, Babak Esfandiari, and Wei Ma. Management applications of the web service offerings language (wsol). In *Advanced Information Systems Engineering, 15th International Conference, CAiSE 2003, Klagenfurt, Austria, June 16-18, 2003, Proceedings*, volume 2681 of *Lecture Notes in Computer Science*, pages 468–484. Springer, 2003.

[98] UDDI, http ://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf. *UDDI technical white paper*, September 2000.

[99] Arnaud VEZAIN. Les service web - présentation générale. Technical report, Association HERMES, February 2005.

[100] J.F. Vilas, J.P. Arias, and A.F. Vilas. An architecture for building web services with quality-of-service features. In *In 5th International Conference on Web-Age Information Management (WAIM 2004)*, 2004.

[101] Andreas Vogel, Brigitte Kerhervé, Gregor von Bochmann, and Jan Gecsei. Distributed multimedia and qos : A survey. *IEEE MultiMedia*, 2(2) :10–19, 1995.

[102] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[103] Steve R. White, James E. Hanson, Ian Whalley, David M. Chess, and Jeffrey O. Kephart. An architectural approach to autonomic computing. In *1st International Conference on Autonomic Computing (ICAC 2004), 17-19 May 2004, New York, NY, USA*, pages 2–9. IEEE Computer Society, 2004.

[104] David S. Wile and Alexander Egyed. An externalized infrastructure for self-healing systems. In *WICSA '04 : Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, page 285, Washington, DC, USA, 2004. IEEE Computer Society.

[105] Emmerich Wolfgang and Kaveh Nima. Component technologies : Java beans, com, corba, rmi, ejb and the corba component model. *SIGSOFT Softw. Eng. Notes*, 26(5) :311–312, 2001.

[106] Zhao Xiangpeng, Cai Chao, Yang Hongli, and Qiu Zongyan. A qos view ofweb service choreography. In *ICEBE '07 : Proceedings of the IEEE International Conference on e-Business Engineering*, pages 607–611, Washington, DC, USA, 2007. IEEE Computer Society.

[107] Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. Qos-aware middleware for web services composition. *IEEE Trans. Softw. Eng.*, 30(5) :311–327, 2004.