Numéro d'ordre : XXXX – Année 2006

Thèse

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

En vue de l'obtention du
Doctorat de l'Institut National Polytechnique de Toulouse
Spécialité : Systèmes Informatiques

Par
Nicolas Salatgé

Conception et mise en oeuvre d'une plate-forme pour la sûreté de fonctionnement des Services Web

Soutenue le 8 décembre 2006 devant le jury composé de :

MM.	Jean-Charles Fabre	Directeur de thèse
	Charles Consel	Rapporteur
	Lionel Seinturier	Rapporteur
	Daniel Hagimont	Membre
	Roberto Baldoni	Membre
	Eric Jenn	Membre

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

Rapport LAAS Numéro XXXXX

Table des matières

Table des figures

Remerciements

Les travaux présentés dans ce mémoire ont été effectués au Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique (LAAS-CNRS). Je remercie Malik Ghallab, qui a assuré la direction du LAAS-CNRS depuis mon entrée, pour m'avoir accueilli au sein de ce laboratoire.

Je remercie également Jean Arlat, Directeur de Recherche CNRS, responsable du groupe de recherche Tolérance aux Fautes et Sûreté de Fonctionnemnent informatique (TSF), pour m'avoir permis de réaliser ces travaux dans ce groupe.

Je tiens aussi à remercier tous les membres du jury :

- Lionel Seinturier, professeur des Universités de Lille, qui m'a fait l'honneur de présider ce jury.
- Charles Consel, professeur des Universités à l'ENSEIRB de Bordeaux, responsable du projet « *Compose* » et Lionel Seinturier, professeur des Universités de Lille, d'avoir accepté la lourde tâche de rapporteur. Je les remercie vivement pour leur lecture attentive du document ainsi que de leurs commentaires avisés.
- Roberto Baldoni, professeur des Universités de Rome, Daniel Hagimont, professeur de l'INP de Toulouse et Eric Jenn, ingénieur à Thalès, d'avoir lu en détail ce document.
- Jean-Charles Fabre, professeur de l'INP de Toulouse, de m'avoir orienté tout au long de cette thèse. Ses conseils m'ont permis de mener à bien ce travail de longue haleine.

Je tiens tout particulièrement à remercier Jean-Charles pour ses remarques constructives qui m'ont inspiré diverses réflexions. Je le remercie pour les nombreuses discussions concernant la substance du travail présenté dans ce document et son soutien tout au long de ces années passées.

La rédaction de ce document a nécessité beaucoup d'efforts tant de ma part que des personnes qui ont pris le temps de lire à plusieurs reprises le manuscrit. Je voudrais remercier tout particulièrement Thomas Pareaud pour ses remarques pertinentes. Merci également à ma fiancée Marie-Line et à ma tante Françoise pour les fautes d'orthographe qu'elles ont sue détecter et corriger.

Je lance un clin d'œil amical à tous ceux qui ont su supporter mes humeurs parfois agacées. Je pense tout particulièrement à mes collègues de bureau et amis, Eric, Ana, Ludovic, Christophe, les deux Thomas, Etienne, Benjamin, Sylvain, Vincent, Emilie et tous les autres.

La thèse représente trois ans d'une vie, durant lesquels la famille compte énormément. De ce côté, j'ai été gâté et je remercie très affectueusement Marie-Line à qui je dédie cette thèse. Merci à Marie-Line pour son soutien, sa patience, sa compréhension et son amour. Merci également à mes chers parents Colette et Jean-François, à mon frère Sébastien, à ma belle-sœur Cécile, à mes futurs beaux-parents Bernadette et Jean-Marc, ainsi qu'à tous les autres.

Introduction Générale

Apparus dès la fin des années 1990, à l'aube du 21ème siècle, les Services Web ont provoqué une forte évolution dans le monde de l'informatique distribuée et un bouleversement majeur dans la façon de concevoir des architectures. Un des intérêts de l'informatique distribuée est de faciliter l'interconnexion entre applications distantes, indépendamment des plates-formes et des langages utilisés. Les trois derniers standards CORBA/IIOP (*Common Object Request Broker Architecture / Internet Inter-ORB Protocol*), DCOM (*Distributed Component Object Model*) et RMI (*Remote Method Invocation*) ont été créés dans ce but. Cependant, ces modèles, de part leur complexité, leur aspect fortement couplé sont en fait, incapables de passer à l'échelle. Ils restent donc, le plus souvent, confinés à l'intérieur des entreprises.

Créées, à la base, pour permettre les échanges commerciaux sur Internet, les technologies des Services Web sont, de par leur nature, très ouvertes. Grâce à des standards d'interopérabilité, ces technologies uniformisent la présentation des services offerts par une entreprise et rendent l'accès transparent à tout type de plate-forme. Le développement d'Internet, la démocratisation du haut débit, la structuration des données via XML et la recherche d'interopérabilité sont autant de facteurs qui ont favorisés l'essor des Services Web. Ces composants sont faiblement couplés permettant ainsi la réalisation d'application dynamique, flexible et évolutive à grande échelle. Les entreprises publiaient déjà de l'information via des sites web, utilisaient la messagerie et faisaient du commerce électronique, elles l'utilisent maintenant pour leurs applications métiers (e-commerce, gestion de multinationale, etc.). La première société à avoir introduit un concept de services distribués, proche de ce qui existe aujourd'hui, est Hewlett-Packard avec son produit e-Speak, et ce dès 1999. La suite fut une grande course entre les principaux acteurs du marché qui ont progressivement, par souci d'interopérabilité, adopté les principaux standards des Services Web que sont SOAP [1], WSDL [2] et UDDI [3].

Cette nouvelle technologie permet donc la création de nouvelles applications aux frontières non délimitées et pousse aujourd'hui les développeurs à privilégier la réutilisation de services « sur étagère », plutôt que de procéder à des développements spécifiques pour chaque projet. L'utilisation de services sur étagère permet aux industriels de se concentrer sur leur domaine de compétence, tout en s'épargnant de l'effort de réaliser des fonctionnalités déjà développées dans d'autres secteurs. Cette tendance à la réutilisation, et surtout l'interconnexion croissante des systèmes, ont ainsi favorisé l'émergence de nouveaux concepts d'architectures tel que les AOS (Architectures Orientées Services ou *Service Oriented Architecture*), qui permettent l'interopérabilité de systèmes même lorsqu'ils sont développés par des organisations indépendantes.

Pour l'heure, ces architectures là ne sont pas utilisées pour des applications ayant de fortes contraintes de sûreté, mais cela ne saurait tarder et l'étude de la sûreté de fonctionnement des architectures orientées services est alors un sujet majeur.

Basés sur les protocoles XML, les Services Web (SW) constituent la technologie de base pour le développement d'Architectures Orientées Services (AOS) [4]. Ces architectures permettent de mettre en place des applications faiblement couplées avec un fort degré de configuration dynamique. Elles se basent sur la notion de relations de "service" formalisée par un contrat

qui unit le client et le prestataire de services. Ce contrat est le point charnière de ce type d'applications.

Derrière cette notion de contrat, se cache en fait un conflit d'intérêt fondamental entre les clients et les prestataires. L'objectif avéré d'un prestataire est de développer un service pour attirer le plus de clients possible. D'un point de vue purement marketing, les Services Web peuvent être développés pour satisfaire les besoins des clients, être facile à maintenir et fournir de hauts niveaux de qualité de service. Les prestataires de Services Web doivent s'assurer de la fiabilité et de la disponibilité de leur infrastructure à base de Services Web. Cependant, les prestataires ne peuvent pas tenir compte de tous les besoins possibles des clients et des contraintes liées au développement de l'application donnée. Cela signifie que des mécanismes additionnels doivent être développés et ciblés pour un contexte d'utilisation donné. Les clients, quant à eux, développant une application basée sur les Services Web avec des contraintes de sûreté de fonctionnement ont une perception différente. Leur but est certainement de trouver le service sur le Net fournissant les fonctionnalités prévues, mais ce n'est pas suffisant. Des contraintes de sûreté additionnelles doivent être remplies. Celles-ci sont très dépendantes et spécifiques de l'application orientée services qu'ils souhaitent mettre en oeuvre.

C'est exactement le type de problème que nous examinons dans ces travaux. Les développeurs d'application regardent les Services Web comme des COTS (*Component Off-The Shell*) et donc ignorent leur implémentation et leur comportement en présence de fautes. De ce point de vue, les clients ont besoin de développer des mécanismes de tolérance aux fautes spécifiques bien adaptés à leurs applications.

Le problème est similaire à l'utilisation des composants COTS dans les systèmes critiques de sûreté, et des travaux précédents ont déjà prouvé que des mécanismes tels que les empaqueteurs (ou wrappers) étaient une solution possible [5]. La différence dans le contexte des AOS est que des wrappers prédéfinis ne peuvent pas être spécifiés pour satisfaire tous les besoins possibles.

L'approche doit être plus flexible pour permettre à des mécanismes de sûreté :

1) d'être définis au cas par cas pour une utilisation donnée du Service Web et
2) d'avoir une forte dynamique afin d'être modifiés selon les besoins.

Dans ce but, nous proposons d'insérer dans la communication client-prestataire des connecteurs spécifiques de tolérance aux fautes (SFTC – *Specific Fault Tolerance Connectors*) qui implémentent des filtres et autres techniques de détection d'erreurs (par exemple, des assertions exécutables) ainsi que les mécanismes de recouvrement qui sont déclenchés quand les Services Web ne satisfont plus les caractéristiques de sûreté demandées (voir la Figure 1). Le même Service Web peut être employé dans plusieurs applications orientées services avec différentes contraintes et donc tirer profit de plusieurs connecteurs (SFTCs).

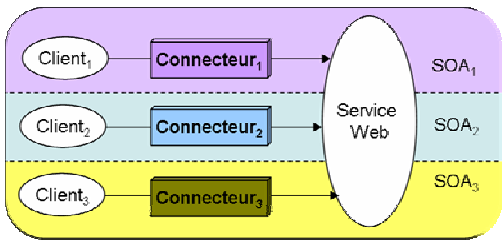

Figure 1: Le connecteur spécifique de tolérance aux fautes

En effet, malgré les mécanismes de sûreté de fonctionnement que peut installer par défaut un prestataire, on peut imaginer qu'il soit nécessaire d'adapter ou de renforcer ces mécanismes pour certains types de clients, sans, bien sûr, modifier ou arrêter les applications existantes. De même, un client doit pouvoir être capable d'appliquer certains mécanismes de sûreté afin de spécifier un Service Web générique avec ses propres contraintes.

D'un point de vue industriel, les entreprises recherchent des approches pour construire des applications réparties à couplage lâche, mais ont encore du mal à franchir le pas. En effet, quelle confiance peut-on accorder à un Service Web, surtout lorsqu'il y a un découplage si grand entre l'interface et l'implémentation d'un COTS? Comment construire des applications orientées services à partir de « boîtes noires » quand on a des contraintes de sûreté de fonctionnement ? En plus des nombreux services existant dans le e-business, on constate aujourd'hui l'essor de services de type e-governement, de services pour les grands systèmes de commandement militaire, ainsi que des services de sauvetage d'urgence (e-rescue) où les aspects de réactivité et de tolérance aux fautes deviennent des problèmes majeurs. Nous pensons que l'ajout de « ***connecteurs*** », capables d'insérer dans des systèmes de ce type des mécanismes de tolérance aux fautes propres à l'application, est fondamental. Ils permettent de spécialiser le Service Web en terme de sûreté de fonctionnement en le rendant, dans une certaine mesure, auto-testable, ce qui augmente la confiance client-prestataire et est donc bénéfique pour les deux acteurs.

Cette problématique peut donc s'énoncer en trois points distincts:

- Comment créer un connecteur fiable garantissant des propriétés de sûreté de fonctionnement spécifique à un contexte donné à partir d'un Service Web générique?
- Quels mécanismes de sûreté (détection, signalement ou recouvrement d'erreur) peut-on introduire dans un tel connecteur?
- Comment permettre à un utilisateur de spécifier et déployer facilement un tel dispositif pour augmenter la sûreté de fonctionnement de son application ?

Il n'existe actuellement que peu de travaux permettant de répondre à ces interrogations. Notre contribution dans ce domaine est de proposer une plate-forme proposant plusieurs outils capables de résoudre les questions précitées, et de répondre ainsi aux problèmes des intégrateurs de systèmes semi-critiques et des fournisseurs de SW.

L'infrastructure de support, nommée IWSD (*Infrastructure for Web Services Dependability*), proposée dans ce mémoire, permet de définir des mécanismes additionnels de tolérance aux fautes développés et insérés par un utilisateur (client ou fournisseur). Notre stratégie consiste à fournir à ces derniers des outils pour rendre l'utilisation des Services Web COTS plus robuste au moyen de connecteurs de tolérance aux fautes. Les outils fournis par la plateforme, se basent sur:

1. l'analyse des requêtes d'entrée et des réponses au moyen d'un analyseur (ou « parseur ») SOAP;
2. la définition d'assertions exécutables liées à l'exigence du client et/ou des fournisseurs;
3. la mise en place de stratégies spécifiques de recouvrement d'erreurs basées sur les redirections et la réplique active;
4. la surveillance et le diagnostic des Services Web et de l'infrastructure elle-même;

5. un langage spécifique ainsi qu'un compilateur pour développer les connecteurs spécifiques de tolérance aux fautes de l'utilisateur.

En effet, afin de réduire au maximum les fautes de développement, nous avons conçu un langage dédié (un *DSL – Domain Specific Language*) appelé DeWeL pour décrire les connecteurs spécifiques de tolérance aux fautes. A l'exécution, la plate-forme IWSD déclenche les diverses actions indiquées par le connecteur en particulier les mécanismes de recouvrement basés sur la redondance de services. IWSD est un conteneur de connecteur muni d'outils de gestion de contrôle et de surveillance. Il joue le rôle de médiateur pour surveiller l'accord passé entre le client et le prestataire.

La Figure 2 présente le type d'applications planétaires construites auquel nous nous intéressons à partir de l'interconnexion de points d'accès qui peuvent être découverts à l'exécution. Ce type d'architecture n'est donc pas figé, elles se construisent dynamiquement en fonction des ressources trouvées correspondant aux besoins fonctionnels du client.

A l'heure actuelle, le nombre d'architectures orientées services sur le Net reste encore restreint. Une des principales raisons vient essentiellement de leur manque de maturité, notamment en matière de sûreté et de sécurité. La plateforme IWSD a essentiellement pour but de rendre un client plus confiant dans le service qu'il utilise en lui permettant d'insérer facilement des mécanismes spécifiques de tolérance aux fautes. Les besoins non fonctionnels des clients mais également des prestataires ont pu être analysés à partir d'un état de l'art rigoureux. Ce manuscrit sera consacré à la présentation de cette plateforme, aux différents modules qui la composent, à ses modes de configuration et à son modèle de fonctionnement à l'exécution. Il est donc constitué de six chapitres, structurés de la façon suivante :

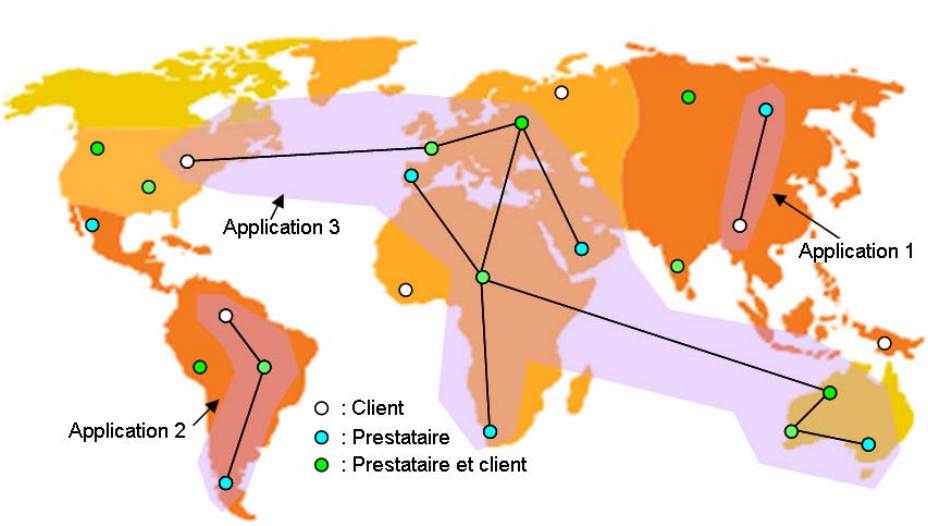

Figure 2: Application à grande échelle à base de Services Web

Le **chapitre 1** présente la problématique à laquelle nous nous intéressons. Il décrit le contexte de notre travail, et les raisons pour lesquelles la mise en place d'une telle plate-forme est nécessaire aujourd'hui. Dans un premier temps, nous décrirons les architectures orientées services ainsi que les différents protocoles basés autour du XML pour les mettre en œuvre. Au regard des principes de la sûreté de fonctionnement des systèmes informatiques, nous examinerons l'état de l'art relatif à la sûreté de fonctionnement des Services Web. Finalement, nous justifierons au travers de cette étude, les différents problèmes que nous abordons et nous présenterons les solutions proposées.

Le **chapitre 2** présente la plate-forme IWSD permettant la mise en place de mécanismes de tolérance aux fautes à base de connecteurs. Nous définirons ses objectifs. Nous détaillerons les spécifications des différents modules et outils qui y sont insérées.

Le **chapitre 3** décrit le langage DeWeL. Nous définirons dans un premier temps l'objectif des DSL. Nous présenterons les différentes caractéristiques de ce nouveau langage. Nous détaillerons par la suite, à travers différents exemples, les possibilités qu'il offre. Nous analyserons enfin le processus de génération de code et de compilation avant d'effectuer un comparatif entre DeWeL et les GPLs (*General Purpose Languages*).

Le **chapitre 4** présente le connecteur en action. Nous verrons les différents mécanismes de détection, de signalement et de recouvrement d'erreurs qu'il peut effectuer. Nous analyserons, par la suite, son interface d'accès qui n'est rien d'autre qu'un nouveau contrat étendu permettant de fournir des caractéristiques non-fonctionnelles du service ciblé.

Le **chapitre 5** donne les résultats expérimentaux que nous avons obtenus pour valider cette plate-forme. Nous présentons les différents Services Web ciblés pour réaliser ces tests. Ces résultats seront par la suite analysés pour mettre en évidence les performances et la robustesse des connecteurs.

Le **chapitre 6** présente nos conclusions sur ce travail et donne les perspectives sur l'extension de nos travaux.

1 Contexte et Problématique

Ce chapitre a pour objectif de présenter le cadre de départ de nos travaux. Dans un premier temps, nous introduisons les concepts de base en sûreté de fonctionnement et en systèmes répartis. Nous définissons ce que nous entendons par architecture orientée service, et décrivons les caractéristiques qui les rendent attrayantes pour les concepteurs d'architectures distribuées. Nous exposons les craintes des intégrateurs quant à la fiabilité de ces composants logiciels de communication qui prennent place au coeur de leurs systèmes.

Enfin, en prenant pour base l'état de l'art des travaux réalisés sur la caractérisation et la tolérance aux fautes des Services Web, nous définirons notre domaine de travail et présenterons la plate-forme qui sera détaillée dans les chapitres suivants de ce mémoire.

1.1 Notions de Sûreté de Fonctionnement

Avant d'aborder l'analyse des architectures orientées services, nous allons rappeler rapidement la terminologie que nous utilisons dans la suite du document pour décrire les notions fondamentales de sûreté de fonctionnement. Cette terminologie et ces concepts sont empruntés à [6, 7].

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

Une ***défaillance*** du système survient lorsque le service délivré dévie de l'accomplissement de la fonction du système, c'est-à-dire de ce à quoi le système est destiné. Une ***erreur*** est la partie de l'état du système qui est susceptible d'entraîner une défaillance, c'est-à-dire qu'une défaillance se produit lorsque l'erreur atteint l'interface du service fourni, et le modifie. Une ***faute*** est la cause adjugée ou supposée d'une erreur. On voit donc qu'il existe une chaîne causale entre faute, erreur et défaillance, représentée dans la Figure 1-1.

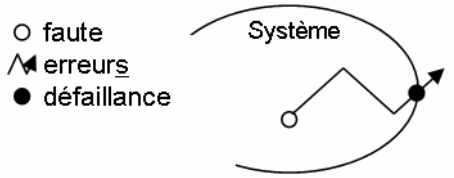

Figure 1-1: Chaîne causale entre faute, erreur et défaillance

Notons que dans les systèmes auxquels nous nous intéressons, qui sont formés par l'interaction de plusieurs sous-systèmes, les notions de faute, d'erreur et de défaillance sont récursives : la défaillance d'un sous-système devient une faute pour le système global (c.f la Figure 1-2).

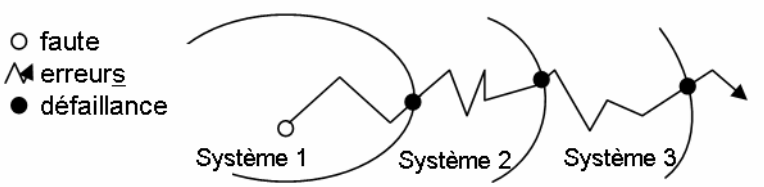

Figure 1-2: Récursivité de la chaîne causale faute => erreur => défaillance

1.1.1 La tolérance aux fautes

Pour que le système soit digne de confiance, et que cette faute ne se propage pas jusqu'au service délivré aux utilisateurs, on peut combiner un ensemble de méthodes qui peuvent être classées comme suit:

- la **prévention des fautes**, qui permet de limiter l'introduction de fautes pendant la phase de développement;
- la **tolérance aux fautes**, qui a pour objectif d'éviter la défaillance du système en utilisant des techniques de détection ou de recouvrement d'erreur;
- l'**élimination des fautes**, qui réduit la présence (nombre, sévérité) des fautes. Pendant la phase de développement, elle consiste à vérifier, diagnostiquer et corriger les fautes. Lors de l'utilisation, il s'agit de maintenance corrective ou préventive.
- la **prévision des fautes**, qui estime la présence et les conséquences de fautes par des tests prenant en compte leurs activités et leurs occurrences;

Parmi ces méthodes, la tolérance aux fautes [8] peut être mise en œuvre par le traitement des erreurs et des fautes [9]. Le traitement d'erreur est destiné à éliminer les erreurs, si possible avant qu'une défaillance ne survienne. Le traitement de faute est destiné à éviter qu'une, ou des fautes ne soient activées à nouveau.

Le traitement d'erreur fait appel à trois primitives:

- **détection d'erreur**, qui permet d'identifier un état erroné comme tel;
- **diagnostic d'erreur**, qui permet d'estimer les dommages créés par l'erreur qui a été détectée et par les erreurs éventuellement propagées avant la détection;
- **recouvrement d'erreur**, qui permet de substituer un état exempt d'erreur à l'état erroné.

L'un des apports de nos travaux est de proposer des mécanismes permettant de détecter, de signaler ou de recouvrer une erreur d'un composant particulier, le Service Web, afin d'éviter qu'elle ne se propage au reste du système.

1.1.2 La caractérisation

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

- ***les défaillances en valeur***, quand la valeur du service délivrée ne permet plus l'accomplissement de la fonction du système;

- ***les défaillances temporelles***, quand les conditions temporelles de délivrance du service ne répondent pas aux besoins des utilisateurs.

Les modes de défaillance par arrêt sont relatifs à la fois aux valeurs et aux conditions temporelles: l'activité du système n'est plus perceptible aux utilisateurs. On distingue une absence d'activité par figement (quand l'état des sorties reste constant), et par silence (dans un système réparti, le noeud concerné n'envoie plus de message).

1.2 Les Architectures Orientées Services (AOS)

L'architecture orientée services est le terme utilisé pour désigner un modèle d'architecture pour l'exécution d'applications logicielles réparties. Les deux derniers modèles (CORBA et DCOM) relèvent de l'architecture par composants logiciels répartis plutôt que de l'architecture orientée services, et le terme « service » est généralement absent de leur terminologie (sauf, par exemple, dans CORBA où l'on parle de services CORBA à propos de fonctions offertes par la plate-forme middleware aux composants applicatifs). De plus, un des avantages des AOS réside dans le fait que les applications réparties n'ont plus besoin d'un système de middleware réparti **commun** pour communiquer, mais seulement des protocoles et des technologies de communications intéropérables sur Internet. Ces protocoles et technologies seront d'ailleurs présentés par la suite.

Construire une architecture orientée services signifie donc d'abord concevoir une architecture en réseau de relations de services, entre applications réparties. La description d'une relation de services est formalisée par un contrat de services. Ce contrat décrit les engagements réciproques du prestataire et du client du service.

L'émergence des technologies de Services Web est censée apporter un niveau d'interopérabilité très élevé avec un degré de couplage très faible et donc d'établir les fondations des architectures orientées services à haut niveau de configuration dynamique. Une architecture d'applications réparties faiblement couplées (ou avec un degré de couplage très faible) est constituée d'un ensemble décentralisé d'applications réparties autonomes (Services Web), lesquelles interagissent sur la base de protocoles de communications, et sont mises en oeuvre à l'aide de technologies ouvertes et non intrusives.

Parmi les exemples visibles d'AOS, on peut citer la SNCF[1] qui a mis en place ce type d'architecture pour son système de réservation (recherche d'horaire, demande de tarif, réservation, ...) qui répond ainsi aussi bien aux terminaux des guichets des agences et gares qu'aux sollicitations du site web de commande en ligne.

Cette partie a pour but d'éclaircir et de définir la notion de « service », de présenter les concepts de base des AOS, et enfin d'analyser les perspectives et problématiques d'un tel type d'architecture.

1.2.1 Qu'est-ce qu'un service ?

La notion de service fait remonter d'un cran le niveau d'abstraction auquel les développeurs d'architectures à composants ont été habitués. Traditionnellement, le composant est représenté par une boîte constituée de plusieurs facettes et/ou réceptacles (des entrées et

[1] Voir : http://www.prnewswire.co.uk/cgi/news/release?id=110004

des sorties). Lorsqu'on parle de service, celui-ci est caractérisé par un point. Une interface, le contrat de service (le document WSDL) précise les messages d'entrée que peut recevoir ce point d'entrée pour réaliser la prestation. Une architecture orientée services peut donc être représentée par une interconnexion de multiples points d'accès (voir Figure 2).

En résumé, la notion de service est le produit d'une démarche d'abstraction par rapport au logiciel qui l'implémente, au processus qui l'exécute et au port qui le localise. Cependant, le service reste un objet très concret et technique. La réalisation d'un service passe par des messages échangés, des transitions d'état et des actions que l'application cliente **suppose** assurés de la part de l'application prestataire du service. Les caractéristiques fonctionnelles, opérationnelles et d'interface d'un service sont consignées dans un contrat. Un Service Web n'est donc rien d'autre que la réalisation supposée de la prestation définie dans le contrat de service.

1.2.2 Le contrat de service

Le contrat de service du modèle des AOS s'inspire directement du modèle des contrats professionnels de service. Il s'agit d'un document qui développe l'ensemble des points permettant de décrire et donc de définir la relation de service.

Dans le monde des relations professionnelles, un contrat de service est un document comportant plusieurs parties et dont les éléments terminaux sont généralement appelés *articles* et *clauses*. Le contrat de service contient des articles et des clauses consacrés à des sujets tels que: l'identification des parties contractantes, la description de la prestation objet du contrat, les modalités d'exécution, les modalités d'interaction entre les parties (il est aussi courant que le contrat de service décrive les actions à entreprendre en cas de défaillance d'un des contractants ou d'impossibilité de réalisation de la prestation).

Dans le monde des AOS, les éléments du contrat de service sont organisés en six thèmes majeurs (voir Figure 1-3): l'identification des parties, la description des fonctions du service, la description de l'interface du service, la description de la qualité de service, la description du cycle de vie du service et du contrat, la description des termes de l'échange.

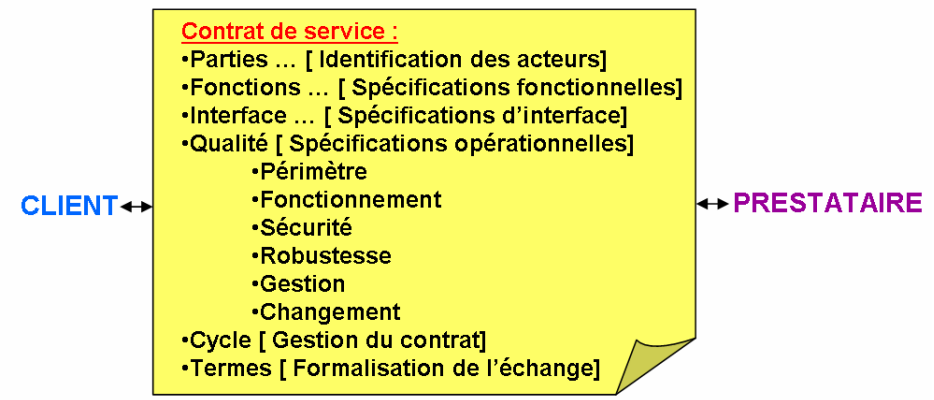

Figure 1-3: Le contrat de service

Parmi ces termes, la qualité de service est un élément majeur puisqu'elle est:

- Un terrain de compétition entre prestataires qui offrent des services exhibant le même modèle fonctionnel et la même interface.

- L'objet d'observation et de suivi de la prestation de la part des clients et des tiers, ainsi que de notation des prestataires par des tiers indépendants jouant le rôle d'organismes d'évaluation.

Hormis ce qui a trait à la sécurité et à la gestion des transactions (et qui est très important par ailleurs), les technologies de Services Web ne proposent pas encore de formalisme pour publier dans le contrat de service les engagements de qualité de service exploitables par les utilisateurs (développeurs, exploitants, …etc.) et par les applications (paramétrage dynamique des délais d'attente, par exemple). IBM a évoqué, dans des travaux datés de 2001 (*Web Services Flow Language* ou WSFL, version 1.0) [10], le développement à venir de WSEL[2] (*Web Services EndPoint Language*), un langage qui devrait permettre de préciser certaines caractéristiques du prestataire (EndPoint) et notamment certains engagements de qualité de service rendu. A la date de rédaction de cet ouvrage, il n'y a pas de résultats publiés de ces travaux.

1.2.3 L'agrégation et la dissémination de service

Une architecture orientée services peut être conçue par une approche incrémentale, résultat de la combinaison de deux démarches de base, présentées ci-dessous, qui sont :

- **L'agrégation de services (cf. Figure 1-4):** La notion d'agrégation fait monter le niveau de découplage entre service et implémentation. La Figure 1-4 représente le service agrégeant de l'agence de voyage. Ce service est issu de la composition (ou de l'orchestration) de services atomiques (ici, les services de réservation d'avions, d'hôtels et de voitures). L'implémentation d'un service par agrégation d'autres services est inconnue du client qui utilise le service agrégeant. En effet, celui-ci ne connaît pas et n'a pas besoin de connaître l'éventuelle complexité de l'implémentation du service en ce qui concerne l'agrégation d'autres services. Du point de vue du client, le service agrégeant est un service atomique décrit de façon usuelle par un contrat. Un service atomique peut être implémenté par agrégation récursive de services atomiques. En guise de remarque, l'agrégation de services se fait par la mise en oeuvre de processus métier à l'aide d'outils tel que BPEL4WS [11]. Il s'agit ici d'établir la coopération entre les services agrégés, à savoir : la division du travail et l'échange d'informations nécessaires pour que les activités de ces services agrégés contribuent efficacement à la réalisation du service agrégeant. Il est également nécessaire de coordonner les services agrégés en synchronisant les étapes de leurs activités. L'implémentation du service agrégeant organise la coopération et coordonne la réalisation des services agrégés, en faisant éventuellement appel à des services spécialisés de coordination comme dans le cas de la gestion des transactions avec des protocoles tels que WS-Coordination [12] et WS-Transaction [13].

[2] Voir : http://www.service-architecture.com/web-services/articles/web_services_endpoint_language_wsel.html

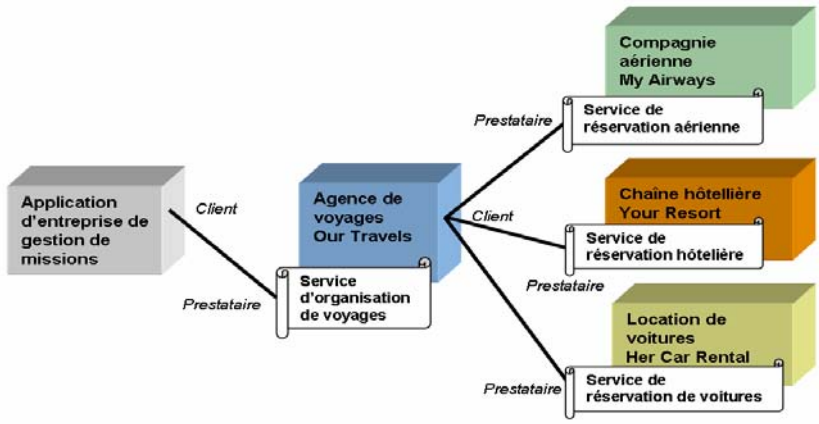

Figure 1-4: Agrégation de services

- **La dissémination de services (cf. Figure 1-5):** La dissémination de services permet d'effectuer la décentralisation des données. Comme on peut le voir sur la Figure 1-5, plusieurs applications sont prestataires du même service sur des données différentes. Une entreprise peut donc implémenter un ensemble de services modulaires à partir d'applications différentes. Le point essentiel est que le client d'un des services issus d'une démarche de dissémination bénéficie de la modularité du service sans que la question de la modularité de l'implémentation ne soit même posée. En informatique, on parle traditionnellement, à ce propos, de « boîte noire » et d'information « hiding ». Dans l'architecture orientée services, le découplage entre interface et implémentation est poussé aux extrêmes limites : un service est tout simplement un contrat, et une occurrence d'un service en exécution est tout simplement un port (dont l'adresse peut être connue dynamiquement).

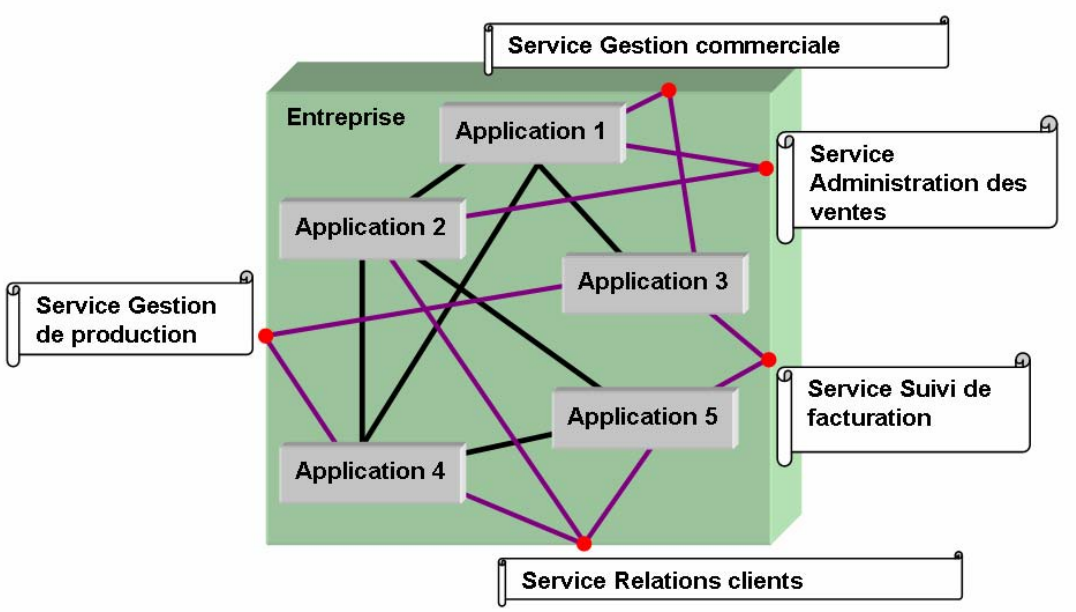

Figure 1-5: Dissémination de services

L'agrégation et la dissémination des services constituent les approches de base de conception et de mise en oeuvre d'architectures orientées services. Les architectures orientées services complexes qui vont se déployer dans les années à venir seront sans doute le résultat de la combinaison de ces deux approches. Il est important d'insister sur le caractère incrémental et

étalé dans le temps de telles démarches. Le modèle de l'architecture orientée services se prête parfaitement à l'urbanisation des systèmes d'information au sens large (intra et inter-entreprises). Il permet ainsi de planifier dans le temps et dans l'espace la réutilisation et la fin de vie des applications existantes, la refonte des applications, la mise en oeuvre de nouvelles applications à valeur ajoutée, tout en garantissant leur interopérabilité. En effet, le concept de l'urbanisation de l'habitat (organisation des villes, du territoire) a été réutilisé en informatique pour formaliser ou modéliser l'agencement du système d'information (SI) de l'entreprise. On utilise le terme d'urbanisation pour mettre l'accent sur le travail progressif nécessaire pour faire évoluer le système d'information vers une cible correctement urbanisée. Les évolutions des stratégies des entreprises (regroupement et fusion, acquisition, diversification des offres commerciales, e-commerce, gestion de la relation client, nouveau mode ou canal de distribution, partenariat, réorganisation, externalisation, redéploiement des fonctions de back et front office, etc.) impliquent des changements structurels importants et accroissent l'interdépendance (dépendance mutualisée) et l'imbrication des applications informatiques avec le risque de renforcer l'effet « plat de spaghettis » du système d'information. Cette complexité croissante a des conséquences sur les coûts, les durées et les risques des projets d'évolution des SI. Les architectures orientées services ont été créées dans le but de faciliter le travail d'urbanisation des grands systèmes informatiques.

1.2.4 Des architectures dynamiques

Cet aspect dynamique de la configuration de l'architecture est au coeur même du concept d'architecture orientée services (ce qui n'empêche pas par ailleurs de mettre en oeuvre des architectures orientées services totalement statiques).

Dans une architecture dynamique, les services qui la composent, les applications prestataires qui interviennent, ainsi qu'un certain nombre de propriétés opérationnelles des prestations de services ne sont pas définis avant sa mise en place, mais sont composés, configurés, établis, voire négociés, au moment de l'exécution. Ce processus peut être itératif : il est possible de reconfigurer une architecture dynamique à la volée lors de son fonctionnement normal, ou bien à l'occasion d'un dysfonctionnement.

Le degré de couplage des architectures réparties évolue sur un continuum qui va du très fortement couplé au très faiblement couplé. Les architectures orientées services permettent de modéliser des architectures à n'importe quel degré de couplage (cf. Figure 1-6).

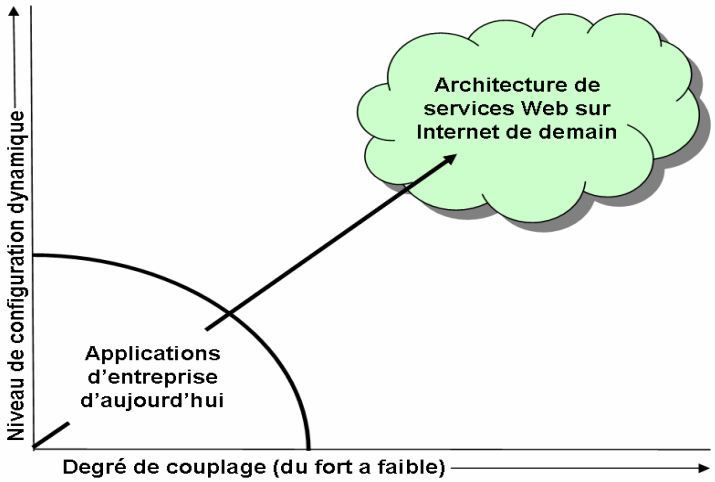

Figure 1-6: Degré de couplage et niveau de configuration dynamique

Les architectures dynamiques permettent aux applications qui les constituent de choisir dynamiquement (à l'exécution)

- les points d'accès des prestataires ;
- les techniques de liaison (implémentations des interfaces) avec les prestataires
- les prestataires des services ;
- les services (contrats) qu'elles utilisent en tant que clientes.

Avec les technologies des services Web disponibles actuellement, on peut notamment établir des architectures dans lesquelles les applications participantes peuvent choisir dynamiquement les services qu'elles consomment, les prestataires de ces services, les ports d'accès de ces prestataires.

Les avantages en termes de qualité de service, de configuration dynamique, de facilité de maintenance et d'évolution des architectures faiblement couplées sont évidents. En revanche, leurs conceptions et leurs mises en oeuvre sont beaucoup plus complexes que celles des architectures fortement couplées. En effet, les AOS possèdent également beaucoup d'inconvénients du fait de leur complexité de mise en oeuvre. L'aspect « boite noire » d'un service pose beaucoup de problèmes pour caractériser sa sûreté de fonctionnement. L'intégrateur de service ne connaît pas du tout l'implémentation du service auquel il accède, il ne fait que se connecter à un port derrière lequel se cache une prestation supposée définie par le contrat associé. Cet aspect « boîte noire » n'est pas uniquement vrai pour les AOS, c'est aussi le cas d'architectures orientées objets telles que CORBA. Cependant, le lien qui unit le client et le prestataire dans ce type d'architecture est beaucoup plus étroit du fait que ce sont des applications essentiellement déployées sur un Intranet.

1.2.5 Des architectures « boîtes noires »

Dans une AOS, les relations de service qui lient les applications participantes sont régies par des contrats de service. Les prestations, issues des contrats, doivent impérativement être décrites au niveau fonctionnel, et il est incorrect d'inclure les modèles d'implémentation des applications prestataires dans les contrats de service. L'implémentation de l'application prestataire est donc une boîte noire par rapport au contrat de service, sauf pour ce qui touche l'implémentation de la communication.

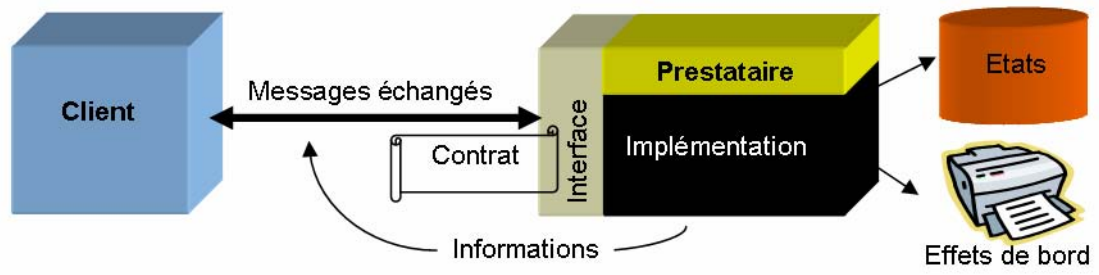

Figure 1-7: Architecture boîte noire avec une interface transparente

Par ailleurs, les fonctions publiées dans le contrat de service ne sont, peut être, qu'une partie des fonctions mises en oeuvre par l'application prestataire. D'autres fonctions peuvent être publiées dans d'autres contrats pour lesquels l'application joue également le rôle de prestataire. Dans tous les cas, la description fonctionnelle complète de l'application, en termes d'objectifs, d'actions, d'informations et de règles, à savoir son coeur métier, constitue

également une boîte noire pour les clients et les autres prestataires de services. Plus précisément, certaines parties du modèle fonctionnel sont publiées dans les contrats de service (avec différents niveaux de visibilité), alors que d'autres parties restent cachées aux clients et aux autres prestataires.

Cet aspect des AOS pose un réel problème à la caractérisation de la sûreté de fonctionnement ainsi qu'à la création et l'efficacité de mécanismes d'empaquetage.

1.3 Les Services Web

Les Services Web sont, à l'heure actuelle, le seul moyen de mettre en place des architectures orientées services. Dans cette partie, nous allons nous échapper un peu de l'aspect conceptuel des architectures orientées services pour nous attarder sur l'aspect fonctionnel, technique, des Services Web. Nous étudierons tout d'abord les différents protocoles de base permettant de mettre en place une architecture à base de Service Web. Nous verrons enfin les outils existants permettant d'implémenter facilement un Service Web.

1.3.1 Les protocoles de base des Services Web

La pile des technologies de Services Web commence à proprement parler avec les protocoles de base : WSDL et SOAP. Ces protocoles imposent un format de message XML. WSDL (*Web Services Description Language*) est le langage de description des Services Web, même s'il n'est pas formellement imposé par l'architecture de référence du W3C. On peut cependant considérer aujourd'hui qu'une description WSDL est nécessaire pour qu'une application puisse revendiquer la qualification de service Web. SOAP (*Simple Object Access Protocol*) est, quant à lui, le protocole standard d'interaction, d'échange d'informations entre un client et un prestataire.

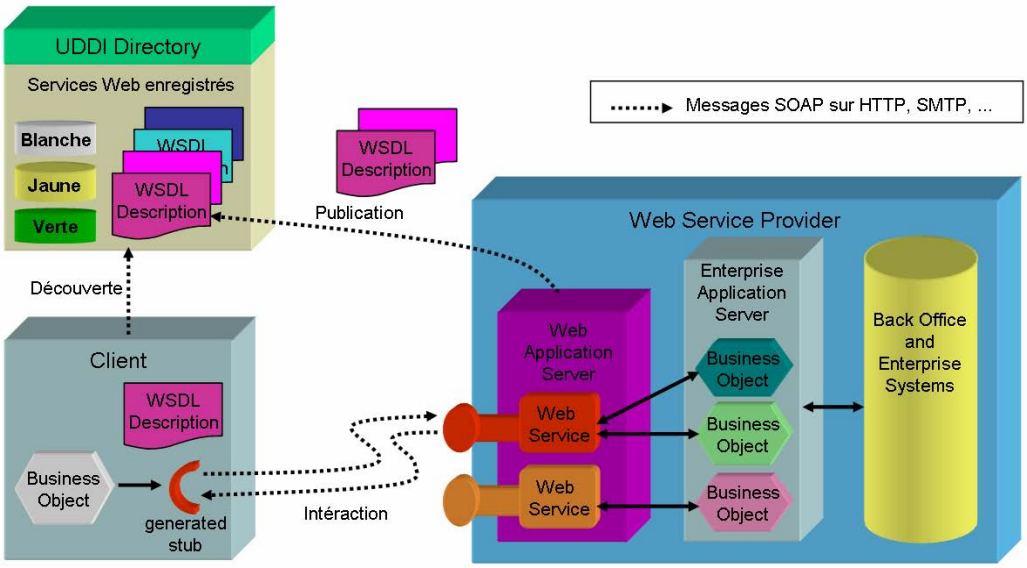

Figure 1-8: Les protocoles de base des Services Web

Les prestataires des Services Web, leurs interfaces et leurs points d'accès, peuvent être enregistrés, découverts et localisés via des technologies d'annuaire comme UDDI (*Universal Description, Discovery and Integration of Web Services*). Autant un standard ouvert (non-propriétaire) sur les annuaires de services semble indispensable, surtout pour la mise en oeuvre d'architectures dynamiques, autant la technologie UDDI, qui est clairement une

technologie de Services Web, n'est pas encore formellement considérée aujourd'hui comme le standard des annuaires.

WSDL, SOAP et UDDI (cf. Figure 1-8) constituent l'ensemble des technologies clés de Services Web, sur lesquelles d'autres technologies plus proches de la problématique applicative peuvent être spécifiées et mises en oeuvre. La Figure 1-9 présente de façon succincte les différents protocoles à base de Services Web qui peuvent se greffer autour des protocoles de base.

La partie qui suit va s'attarder à détailler précisément certaines caractéristiques des protocoles de base. Cette partie est en fait essentielle pour bien comprendre par la suite les différents outils qui ont été insérés dans la plate-forme IWSD et leur complexité.

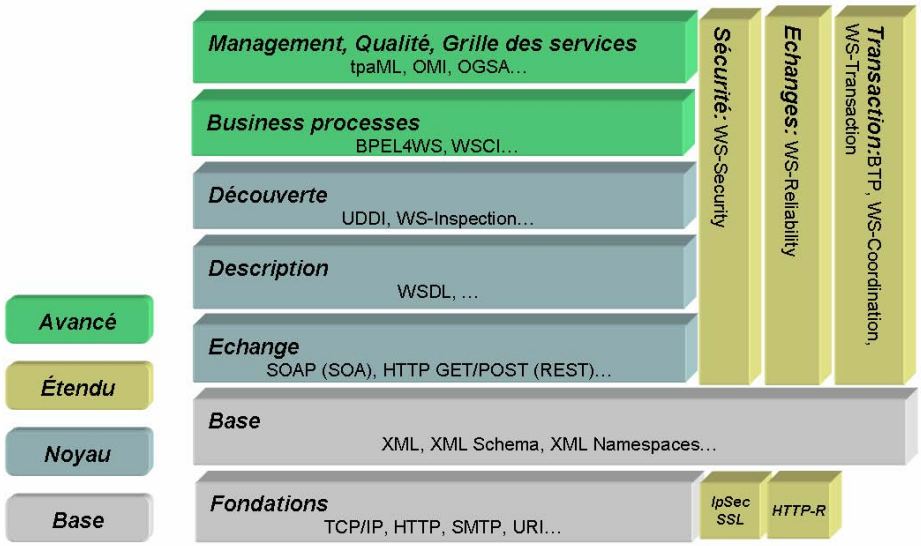

Figure 1-9: La pile des protocoles WS

1.3.1.1 XML

XML (*Extensible Markup Language*) est un langage de balisage extensible [14] qui a été mis au point par le XML Working Group sous l'égide du *World Wide Web Consortium* (W3C). Les spécifications XML 1.0 sont reconnues comme recommandations par le W3C, ce qui en fait un langage reconnu. XML est un standard qui sert de base pour créer des langages balisés spécialisés; c'est un « méta langage ». Il est suffisamment général pour que les langages basés sur XML, appelés aussi dialectes XML, puissent être utilisés pour décrire toutes sortes de données et de textes. Il s'agit donc partiellement d'un format de données. Son objectif est, dans un échange entre systèmes informatiques, de transférer, en même temps, des données et leurs structures. Permettant de coder n'importe quel type de donnée, depuis l'échange EDI (*Electronic Data Interchange ou Echange de Données Informatisées*) [15] jusqu'aux documents les plus complexes, son potentiel est de devenir le standard universel et multilingue d'échange d'informations.

XML Namespaces est une extension de la recommandation XML qui permet de créer des espaces de nommages. Les espaces de noms d'XML permettent de qualifier de manière unique des éléments et des attributs. On sait alors à quel domaine de définition se rapporte un objet et comment il doit être interprété, selon sa spécification. Différencier des espaces de noms permet de faire coopérer, dans un même document, des objets ayant le même nom, mais une signification différente, souvent liée à un modèle de contenu différent. Cette spécification est une avancée importante car, à partir du moment où beaucoup de formats s'expriment selon

XML, les risques de "collision de noms" deviennent plus importants et cette spécification prend alors toute son importance.

1.3.1.2 WSDL

WSDL [16] est l'outil pivot de la technologie des Services Web car il permet véritablement de donner une description d'un Service Web indépendante de sa technologie d'implémentation. Les traits principaux du langage sont présentés via l'exemple d'un des services Web les plus populaires : l'accès programmatique par SOAP au moteur de recherche Google[3].

Dans la mise en place des architectures de Services Web aujourd'hui, la fonction de contrat de service est portée par le document WSDL. Le choix d'un format universel et extensible de structuration de documents basé sur XML s'impose, pour satisfaire les deux contraintes principales qui pèsent sur le contrat de service :

- Le contrat de service doit être, en même temps, lisible par des acteurs humains et exploitable (généré, interprété, agrégé, décomposé, indexé, mémorisé, …etc.) par des agents logiciels.
- Le contrat de service doit être tout aussi facilement extensible, c'est-à-dire adaptable à l'évolution des services, des architectures et des technologies, et capable d'accueillir les nouveaux formalismes propres à de nouveaux types d'engagements.

Actuellement, le document WSDL ne permet de définir que l'implémentation de l'interface, à savoir : les styles d'échange, les formats des messages, les conventions de codage, les protocoles de transport, les ports de réception.

Un document WSDL définit une suite de descriptions de composants (cf. Figure 1-10). Le fichier WSDL est principalement composé de 6 éléments, avec d'autres éléments optionnels:

- **definitions:** élément racine du document, il donne le nom du service, déclare les espaces de noms utilisés, et contient les éléments du service (Obligatoire).
- **types:** décrit tous les types de données utilisés entre le client et le prestataire. WSDL n'est pas exclusivement lié à un système spécifique de typage, mais utilise par défaut la spécification XML Schema. La présentation de schéma XML est effectuée dans le chapitre suivant (Définition abstraite).
- **message:** décrit un message unique, que ce soit un message de requête seul ou un message de réponse seul. L'élément définit le nom du message et peut contenir (ou pas) des éléments « **part** », qui font référence aux paramètres du message ou aux valeurs retournées par le message (Définition abstraite).
- **portType:** combine plusieurs messages pour composer une opération. Chaque opération se réfère à un message en entrée et à des messages en sortie. (Définition abstraite).
- **binding:** décrit les spécifications concrètes concernant la manière dont le service sera implémenté: protocole de communication et format des données pour les opérations et

[3] Voir: http://api.google.com/GoogleSearch.wsdl

messages définis par un type de port particulier. WSDL possède des extensions internes pour définir des services SOAP; de fait, les informations spécifiques à SOAP se retrouvent dans cet élément (Définition concrète).

- **service:** défini les adresses permettant d'invoquer le service donné, ce qui sert à regrouper un ensemble de ports reliés. La plupart du temps, c'est une URL invoquant un service SOAP (Définition concrète).

```
<definitions name="GoogleSearch" targetNamespace="urn:GoogleSearch"  xmlns:typens="urn:GoogleSearch"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"   xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types><xsd:schema>…</xsd:schema></types>
  <message name="doGoogleSearch">
    <part name="key"          type="xsd:string"/>
    <part name="q"            type="xsd:string"/>
    <part name="start"        type="xsd:int"/>
    <part name="maxResults"   type="xsd:int"/>
    <part name="filter"       type="xsd:boolean"/>
    <part name="restrict"     type="xsd:string"/>
    <part name="safeSearch"   type="xsd:boolean"/>
    <part name="lr"           type="xsd:string"/>
    <part name="ie"           type="xsd:string"/>
    <part name="oe"           type="xsd:string"/>
  </message>
  <message name="doGoogleSearchResponse">
    <part name="return"       type="typens:GoogleSearchResult"/>
  </message>
  <portType name="GoogleSearchPort">
    <operation name="doGoogleSearch">
      <input message="typens:doGoogleSearch"/>
      <output message="typens:doGoogleSearchResponse"/>
    </operation>
  </portType>
  <binding name="GoogleSearchBinding" type="typens:GoogleSearchPort">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="doGoogleSearch">
      <soap:operation soapAction="urn:GoogleSearchAction"/>
      <input>
        <soap:body use="encoded" namespace="urn:GoogleSearch"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded" namespace="urn:GoogleSearch"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
  </binding>
  <service name="GoogleSearchService">
    <port name="GoogleSearchPort" binding="typens:GoogleSearchBinding">
      <soap:address location="http://api.google.com/search/beta2"/>
    </port>
  </service>
</definitions>
```

Figure 1-10: Contrat WSDL de Google

De nombreuses implémentations de Services Web se sont juste limitées à l'utilisation de la couche de transport SOAP pour faire intérargir un client et un prestataire sans déclarer le service au travers d'un document WSDL. Cela est suffisant dans une situation où le service en question ne présente qu'un intérêt limité, propre aux deux acteurs qui contrôlent les noeuds de communication concernés. En revanche, si le Service Web est destiné à une utilisation dans un cadre plus élargi, il va rapidement devenir fastidieux pour le fournisseur de ce service de décrire et d'informer chaque consommateur potentiel des caractéristiques fonctionnelles et techniques de ce service. Il sera certainement préférable que ce fournisseur concentre ses ressources sur les aspects commerciaux et contractuels de son offre par exemple.

Ainsi, la spécification WSDL joue un rôle pivot dans une architecture de Services Web. C'est la présence d'un contrat WSDL qui permet d'affirmer que l'on met en œuvre un Service Web.

Le seul usage de SOAP dans la communication entre applications réparties ne suffit pas pour qualifier ces applications de Services Web.

1.3.1.3 Les Schémas XML

Un schéma XML définit les éléments possibles dans un document et les attributs associés à ces éléments avec leur type de données. Les schémas sont standardisés sous forme de recommandation du W3C (XML Schema 1.0) depuis mai 2001. Les schémas XML [17] ont un rôle essentiel dans la création des Services Web. Ils permettent de créer des types propres aux services. Les types de données créés sont soit des types simples, soit des types complexes.

Il existe une large variété de types simples intégrés à XML (*Built-in Simple Type*), dont les entiers, les réels, les chaînes de caractères et les dates sous des formes variées. De plus, les types simples intégrés peuvent être également spécialisés.

La Figure 1-11 représente la hiérarchie des types définis dans le schéma des schémas XML (http://www.w3.org/2001/XMLSchema.xsd).

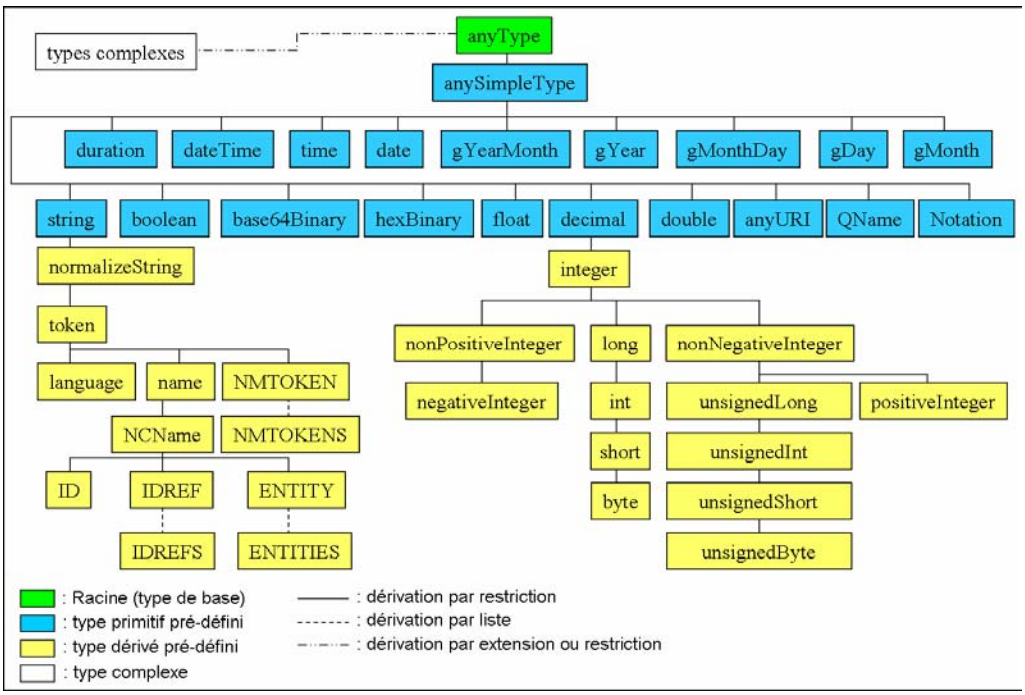

Figure 1-11: La hiérarchie des définitions de type d'XML Schema

Les types complexes sont des types composés d'éléments assemblés à l'aide de constructeurs: séquence (sequence), choix (choice) et tas (all). Ils permettent de définir la structure de documents complexes, de manière quasi similaire aux constructeurs du monde objet. Ils sont cependant, plus limités que ceux du monde objet et plus appropriés à la modélisation d'éléments pour les documents XML. La Figure 1-12 présente ainsi le type complexe retourné par l'opération « *doGoogleSearch* » de Google. Ce type est constitué d'une séquence d'éléments simples et complexes.

```
<xsd:complexType name="GoogleSearchResult">
    <xsd:all>
        <xsd:element name="documentFiltering"              type="xsd:boolean"/>
        <xsd:element name="searchComments"                 type="xsd:string"/>
        <xsd:element name="estimatedTotalResultsCount"     type="xsd:int"/>
        <xsd:element name="estimateIsExact"                type="xsd:boolean"/>
        <xsd:element name="resultElements"                 type="typens:ResultElementArray"/>
        <xsd:element name="searchQuery"                    type="xsd:string"/>
        <xsd:element name="startIndex"                     type="xsd:int"/>
        <xsd:element name="endIndex"                       type="xsd:int"/>
        <xsd:element name="searchTips"                     type="xsd:string"/>
        <xsd:element name="directoryCategories"            type="typens:DirectoryCategoryArray"/>
        <xsd:element name="searchTime"                     type="xsd:double"/>
    </xsd:all>
</xsd:complexType>
```

Figure 1-12: XML Schema - Le type complexe GoogleSearchResult

Les schémas XML possèdent des instructions qui donnent une grande puissance d'expressivité:

- ***Spécialisation de types par restriction :*** Comme avec les sous-classes des langages orientés objet, il est possible de définir des sous-types par héritage. De manière classique, la spécialisation peut s'effectuer sur un type simple ou complexe par ajout de contraintes, appelées restrictions. Ces contraintes peuvent porter sur une propriété de type comme dans l'exemple présenté dans la Figure 1-13. De manière générale, tous les types simples mais aussi les types complexes peuvent être réduits par des contraintes de restriction, appelées facettes (ex : facette « pattern » du type string dans la Figure 1-13). Les facettes définissent les différents types de restrictions applicables sur un type de données (cf. Annexe A.1.3.4).

```
<xsd:simpleType name=« typeCodePostalFR »>
    <xsd:restriction base=«xsd:string »>
        <xsd:pattern value=« \d{5} »>
    </xsd:restriction>
</xsd:restriction>
```

Figure 1-13: Spécialisation de type par restriction

- ***Dérivation de types par extension :*** L'ajout d'information à un type complexe est appelé une extension. Les types ainsi spécialisés sont marqués par une balise <complexContent>. La clause <xsd :extension> introduit cette catégorie de sous-typage avec en attribut le nom du type de base.

En résumé, les schémas offrent une grande variété de types simples surchargeables et la possibilité de construire des types complexes extensibles. Les schémas apportent une définition des types de données de base riche et extensible, un peu comme l'objet. Sans schéma, XML reste un langage de représentation de documents textuels et n'est pas sous-tendu par un véritable modèle de données. Les schémas sont complexes et relativement difficiles à mettre en œuvre. Ils sont toute fois indispensables à la création des services Web et ont une part importante dans la définition du contrat WSDL. Les schémas XML sont spécifiés d'emblée comme le seul outil de définition de format XML dans les Services Web.

1.3.1.4 SOAP

SOAP [1, 18-21] fournit un mécanisme qui permet d'échanger de l'information structurée et typée entre applications dans un environnement réparti et décentralisé. Il ne

véhicule pas de modèle de programmation ou d'implémentation, mais fournit les outils nécessaires pour définir des modèles opérationnels d'échange (styles d'échange) aussi diversifiés que les systèmes de messagerie asynchrone et l'appel de procédure distante (RPC).

Le message SOAP est un document XML. Un message SOAP présente une structure normalisée (cf. Figure 1-14). Il est toujours constitué d'une enveloppe (SOAP-ENV:Enveloppe) munie d'une en-tête (SOAP-ENV:Header) optionnelle et d'un élément (SOAP-ENV:Body) obligatoire contenant le corps du message, suivis d'éventuels éléments applicatifs spécifiques.

La spécification considère explicitement que les en-têtes SOAP sont destinés à la mise en œuvre de couches supérieures et transversales de la technologie des services Web, comme la gestion des transactions, la gestion de la sécurité, etc.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
                   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
                   xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Header></SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <ns1:doGoogleSearch xmlns:ns1="urn:GoogleSearch"
                        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <key xsi:type="xsd:string">dpQNf25QFHLo0i6kv+/IPfxrdgT7aTIh</key>
      <q xsi:type="xsd:string">web</q>
      <start xsi:type="xsd:int">0</start>
      <maxResults xsi:type="xsd:int">10</maxResults>
      <filter xsi:type="xsd:boolean">true</filter>
      <restrict xsi:type="xsd:string"></restrict>
      <safeSearch xsi:type="xsd:boolean">false</safeSearch>
      <lr xsi:type="xsd:string"></lr>
      <ie xsi:type="xsd:string">UTF-8</ie>
      <oe xsi:type="xsd:string">UTF-8</oe>
    </ns1:doGoogleSearch>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 1-14: Requête SOAP de Google

1.3.1.5 UDDI

UDDI [3] est le support d'un système réparti d'annuaires répliqués qui permettent la publication et la découverte de services sur Internet. Un annuaire UDDI est accessible par l'intermédiaire du protocole SOAP. L'API UDDI est un service Web décrit au format WSDL qui permet d'accéder à un annuaire via l'utilisation du protocole SOAP. On peut également faire une analogie entre l'annuaire UDDI et le service de désignation de CORBA.

L'annuaire UDDI est accessible soit via un navigateur Web qui dialogue avec une application Web dédiée, interface spécifique à l'annuaire accédé, soit par programme, en utilisant l'API (*Application Programming Interface*) définie par la spécification.

L'API comporte deux groupes de fonctions :

- ***les fonctions de recherche (Inquiry API) :*** navigation, recherche et consultation des informations de l'annuaire;
- ***les fonctions de publication (Publishing API) :*** publication, création, modification ou suppression des informations de l'annuaire.

Un annuaire UDDI peut être public ou privé. De manière standard, la spécification prévoit qu'un annuaire est distribué sur plusieurs noeuds. Ces noeuds sont synchronisés au moyen du

mécanisme de réplication interne. La réplication n'est pas obligatoire, notamment dans un cadre privé, mais est fortement recommandée pour des raisons évidentes de disponibilité. Cette fonctionnalité est, bien sûr, mise en œuvre par l'annuaire public UDDI, dont les implémentations des opérateurs (IBM, Microsoft, NTT Communications et SAP) se répliquent entre elles.

1.3.2 Installation d'un Service Web

Un service Web n'est rien d'autre qu'un point d'accès atteignable via le Net. Cela signifie qu'aucun détail d'implémentation n'est connu mais qu'en plus l'implémentation peut être faite de différentes manières.

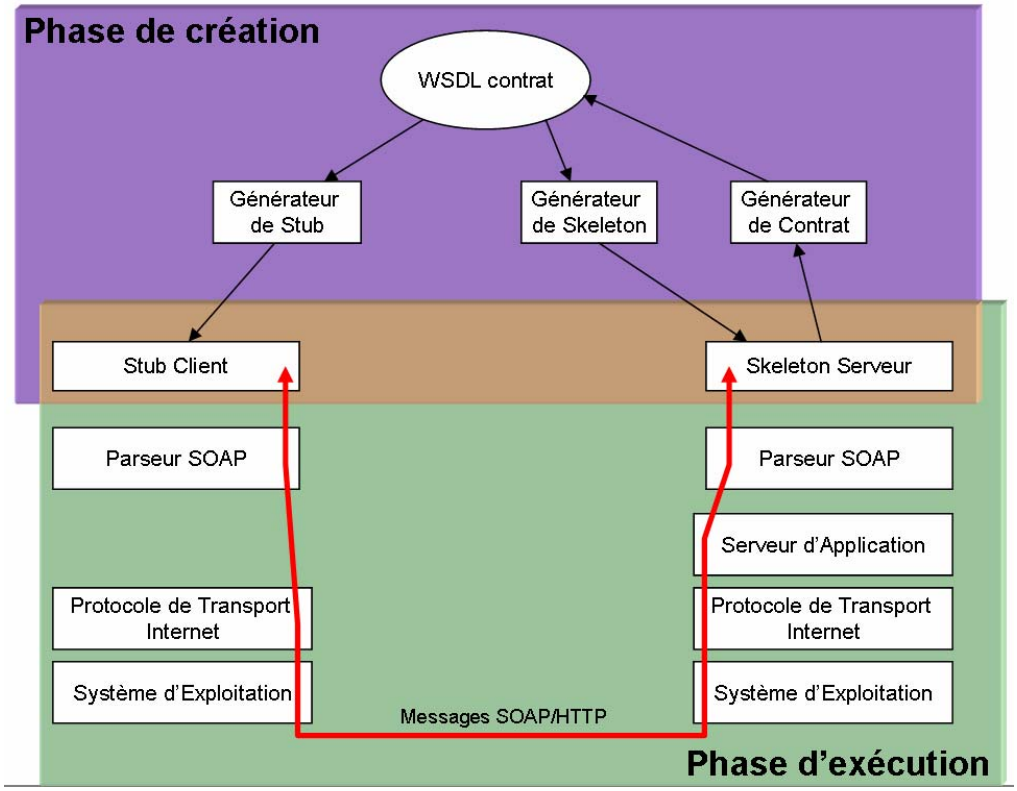

Figure 1-15: Composant fonctionnel des Services Web

La Figure 1-15 représente une communication client-prestataire SOAP en mode RPC. Le protocole de transport Internet utilisé est le protocole standard HTTP.

De façon générale, un service Web est un composant logiciel (ici, le skeleton serveur) reposant sur un parseur SOAP (ex: Axis [22, 23], ASP.NET [24]…etc.). Ce parseur est lui-même intégré, la plupart du temps, dans un serveur d'application. Le serveur d'application agit, en fait, comme un serveur dans un restaurant prenant la requête du client, regroupant les données et appelant les modules capables de la traiter (ici, le parseur SOAP). Les serveurs d'application sont des logiciels occupant le tiers central dans une architecture multi tiers. Les serveurs d'applications existaient bien avant l'apparition des Services Web. Ils effectuent le relais entre les clients et une entité métier: une base de données, un EJB ou une application web (Servlet/JSP). Dans notre cas, le serveur d'application transmet la requête reçue provenant d'un client au parseur SOAP. Celui-ci est en charge de la désérialiser pour la transmettre au Service Web correspondant. Les serveurs d'applications utilisent souvent J2EE ou .Net bien que d'autres plates-formes soient envisageables (Python).

Des Services Web sont, le plus souvent, déployés en employant un logiciel de serveur d'application :

- Axis et le serveur de Jakarta Tomcat [25] (deux projets open source d'Apache Software Foundation),
- ColdFusion MX de Macromedia [26],
- Bibliothèque pour les développeurs de services Web en Java (JWSDP) [27] de Sun Microsystems (basé sur Jakarta Tomcat),
- Serveurs HTTP IIS de Microsoft (avec le framework .NET) [28],
- WebLogic de BEA [29],
- WebSphere Application Server d'IBM (basé sur le serveur d'Apache et la plate-forme de J2EE) [30],
- Oracle Application Serveur d'Oracle Corporation [31],
- ZenWorks de Novell [32],
- Bibliothèque pour les développeurs de Services Web en PHP NuSOAP [33],
- JBoss Application Server de la société JBoss [34]. Composant du JEMS (JBoss Enterprise Middleware System) dont fait également parti le framework de persistance relationnelle Hibernate[4] .

La Figure 1-15 permet donc de mettre en évidence le constat suivant: Rendre un Service Web sûr de fonctionnement nécessite forcément de rendre toutes les couches sur lequel repose le service, sûres de fonctionnement. Les serveurs d'applications sont des composants informatiques fortement soumis aux attaques, et sur lesquels des vulnérabilités ont pu être identifiées. Sun ONE Application Server [35], le serveur d´application de Sun, serait, par exemple, vulnérable à un dépassement de mémoire tampon exploitable à distance. Cette faille grave, est de même nature que celle qui a frappé plus récemment le couple Windows 2000 / IIS de Microsoft. Le Certa[5] (Centre d'Expertise Gouvernemental de Réponse et de Traitement des Attaques informatiques) recense toutes les attaques détectées. Celles-ci concernent la plupart des serveurs d'applications (Codl-Fusion, Tomcat, BEA WebLogic, IBM WebSphere). Au vu de ce constat, l'ajout de mécanismes de détection ou de recouvrement d'erreur semble nécessaire. Bien que nous ne nous intéressions pas dans ce mémoire aux problèmes de sécurité liés au serveur d'application ou au Service Web, il est tout de même bon de noter que ces vulnérabilités peuvent entraîner une indisponibilité du serveur et nuire à l'ensemble des applications reposant dessus.

1.3.3 Services Web : Récapitulatif

Un Service Web est un ensemble de protocoles et de normes informatiques utilisés pour échanger des données entre les applications. Les logiciels écrits dans divers langages de programmation et sur diverses plates-formes peuvent employer des Services Web pour échanger des données à travers des réseaux informatiques comme Internet. Cette

[4] Voir : http://www.hibernate.org/
[5] Voir : http://www.certa.ssi.gouv.fr

interopérabilité est due à l'utilisation de normes ouvertes regroupées au sein du terme générique de SOA (*Service Oriented Architecture ou Architecture orientée services*). L'OSI et le World Wide Web Consortium (W3C) sont les comités de coordination responsables de l'architecture et de la standardisation des services Web. Pour améliorer l'interopérabilité entre les réalisations de Service Web, l'organisation WS-I[6] (Web Services Interoperability) a développé une série de profils pour faire évoluer les futures normes impliquées. Les Services Web se composent d'une collection de standards que l'on regroupe sous le terme : « Web Service Protocol Stack » (cf. Figure 1-9).

En définitive, et compte tenu de ce que nous venons de dire, les Services Web fournissent les avantages suivants :

- L'interopérabilité entre divers logiciels fonctionnant sur différentes plates-formes.
- L'utilisation de standards et protocoles ouverts.
- Les protocoles et les formats de données sont au format texte dans la mesure du possible, facilitant ainsi la compréhension du fonctionnement global des échanges.
- Basés sur le protocole HTTP, ils peuvent fonctionner au travers de nombreux pare-feux sans nécessiter des changements sur les règles de filtrage.

Cependant, ne soyons pas dupes, les Services Web présentent également de gros inconvénients:

- Les normes de services Web dans les domaines de la sécurité et des transactions sont actuellement inexistantes ou toujours dans leur petite enfance comparées à des normes ouvertes plus mûres de l'informatique répartie telle que CORBA.
- Les Services Web souffrent de performances faibles comparées à d'autres approches de l'informatique répartie telles que le RMI, CORBA, ou DCOM.
- Par l'utilisation du protocole HTTP, les Services Web peuvent contourner les mesures de sécurité mises en place au travers des pare-feux. D'après une citation du Gartner Group faite en 2003, *"Les Services Web XML vont rouvrir 70% des chemins d'attaques fermés par les pare-feux lors de la dernière décennie. Ils peuvent transporter virtuellement toutes les données utiles sur le port 80 et le pare-feu ne peut les arrêter."*[7]

Rien ne permet pour l'instant d'assurer la qualité d'exécution d'un Service Web. Il n'y a donc aucune qualité de service (QoS) associée à ces derniers

La suite de ce chapitre sera dédiée à l'analyse de l'état de l'art. Nous allons nous intéresser, dans un premier temps, à dimensionner le problème, c'est-à-dire, à cerner les différents points de sûreté de fonctionnement sur lesquels nous nous concentrons. Nous verrons ensuite les différents travaux qui ont été mis en œuvre pour résoudre certains de ces problèmes ou une partie d'entre eux. Nous analyserons, enfin, en quoi nos travaux diffèrent de ceux des autres.

---

[6] Voir : http://www.ws-i.org

[7] Voir : http://soaj2ee.blogspirit.com/archive/2005/06/03/pourquoi_securiser_l_architecture_orientee_service_soa.html

1.4 Problématique

Cette section a pour but de faire ressortir, en fonction des aspects de sûreté de fonctionnement auxquels nous nous intéressons, les besoins des clients et des prestataires vis-à-vis des Services Web. Nous verrons ensuite, les motivations qui nous ont incité à mettre en place l'architecture proposée dans ce mémoire en la comparant aux travaux actuels.

1.4.1 Dimensionnement du problème

Dans ce mémoire, nous allons uniquement nous intéresser à certaines caractéristiques de la sûreté de fonctionnement vis-à-vis des Services Web. La sûreté de fonctionnement d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre. Selon la, ou les applications auxquelles le service Web est destiné, l'accent peut être mis sur différentes facettes de la sûreté de fonctionnement, ce qui revient à dire que la sûreté de fonctionnement peut être vue selon des propriétés différentes mais complémentaires, qui permettent de définir les attributs suivant:

- Le fait d'être prêt à l'utilisation conduit à la ***disponibilité***;
- La continuité du service conduit à la ***fiabilité***;
- La non-occurrence de conséquences catastrophiques pour l'environnement conduit à la ***sécurité-innocuité***;
- La non-occurrence de divulgations non-autorisées de l'information conduit à la ***confidentialité***;
- La non-occurrence d'altérations inappropriées de l'information conduit à ***l'intégrité***;
- L'aptitude aux réparations et aux évolutions conduit à la ***maintenabilité***.

La maturité des travaux concernant la confidentialité et l'intégrité étant beaucoup plus aboutie que les autres, nous ne nous intéressons pas dans ce mémoire aux travaux basés sur cette thématique. On peut tout de même mentionner qu'actuellement, sur un réseau IP, la confidentialité des échanges, l'intégrité des messages et l'authentification des agents participant à l'échange peuvent être gérées au travers d'une session sécurisée, via l'utilisation de protocoles de sécurité bas niveau tel que SSL, TLS [36] et éventuellement IPSEC [37]. Cependant, bien que la gestion de la sécurité au niveau transport soit nécessaire aux architectures de services Web, elle est insuffisante pour mettre en œuvre des architectures orientées services complexes sur Internet. Il a donc fallu réaliser des technologies d'infrastructure spécialement conçues pour la gestion de la sécurité des Services Web : WS-Security [38], spécification édictée par OASIS, couvre l'ensemble des problématiques de sécurité d'un processus transactionnel comme l'authentification des utilisateurs et la gestion de leurs droits, la gestion de l'intégrité des messages par le biais de certificats ou encore le chiffrement des flux de données.

Ce mémoire se concentre donc sur la problématique plus ouverte de la disponibilité et de la fiabilité des Services Web. Nos travaux portent sur l'étude des mécanismes permettant d'informer les clients ou les prestataires de la disponibilité et de la fiabilité courante du Service Web utilisé, mais surtout d'améliorer sa sûreté de fonctionnement. Cette thématique fait partie des recherches actuelles dans ce domaine.

1.4.2 Le conflit d'intérêt clients-prestataires

Le contrat de service est l'accord passé entre les clients et les prestataires. Cet accord est le point charnière de ce type d'application. Cependant, les deux participants ayant des problèmes et des exigences différents, cet accord est en fait difficile à mettre en place et à maintenir dans le temps.

Du point de vue des **développeurs** de Services Web (les prestataires), le service créé doit être le plus générique possible pour satisfaire le plus de clients possible. Cet aspect générique est essentiel aux fournisseurs pour pouvoir contrôler les coûts de production du service. Il permet également de le rendre plus maintenable et évolutif au besoin du marché. Certes, un Service Web est certainement difficile à réaliser d'un point de vue fonctionnel ("*Quels types de services sont exigés par les clients?*") mais c'est un énorme défi d'un point de vue non fonctionnel ("*Quel niveau de QoS dois je fournir, quel type de contraintes et de mécanismes de sûreté dois-je prévoir pour les divers clients?*")! Il s'agit d'une question ouverte car les besoins du client, l'environnement d'utilisation, le rapport entre les performances et les exigences non-fonctionnelles ne sont pas connues du fournisseur! En effet, au-delà, des mécanismes que peut installer le prestataire pour accroître la sûreté de fonctionnement de ses services, celle-ci peut, dans certains cas et suivant le type d'application, ne pas être adaptée ou suffisante au contexte d'utilisation d'un client particulier. En fait, la concurrence féroce parmi les prestataires de services devrait avoir comme conséquence le développement et le déploiement rapide de services. Cependant, les services offerts par un prestataire, étant gelés par le document WSDL, ne peuvent pas suivre les opportunités du marché.

Du point de vue des **intégrateurs** de Services Web semi-critiques, les clients ont besoin d'assurances supplémentaires sur la qualité et la robustesse des Services Web « boîte noire » qu'ils intègrent au sein de leurs systèmes. Ils ont besoin de plus de confiance. Celle-ci ne peut être acquise que par l'insertion de mécanismes de sûreté spécifiques aux besoins des clients tels que la détection, le signalement ou le recouvrement d'erreurs. Sachant que dans une architecture modulaire, le niveau de sûreté de fonctionnement est très dépendant du composant le plus faible, le choix d'un Service Web répondant parfaitement à ces besoins non-fonctionnels devient primordial. Les clients devraient être capables d'ajuster le niveau de disponibilité et de fiabilité du service en fonction de contraintes imposées par l'application dans laquelle il doit être intégré. Ils ont également besoin d'informations sur les modes de défaillance et sur les canaux de propagation d'erreur introduits par le Service Web, ainsi que des informations quantitatives leur permettant de comparer différentes implémentations candidates du point de vue de la sûreté de fonctionnement, afin de sélectionner le candidat le plus adapté à leurs besoins. Dans la vie courante, l'appréciation d'un prestataire de services (par exemple, pour un prestataire de téléphonie mobile : SFR, Orange, ...etc.) s'effectue en fonction du bouche à oreille, de la publicité ou du rapport d'activité du prestataire en question. De part le manque de maturité à l'heure actuelle des Services Web, de telles informations ne sont pas directement accessibles ou connues. Le choix d'un service passe seulement par la confiance (justifiée ou non) qu'à un client envers le prestataire. En guise de remarque, si l'on se réfère aux contrats d'affaires critiques établis entre deux grandes entreprises internationales, la phase de signature et d'exécution d'un contrat est généralement suivie par des instances extérieures (avocats, huissiers, spécialistes, …etc.) s'assurant du bon déroulement. Ceci a pour but de renforcer la confiance du client envers son prestataire et vice-versa.

Figure 1-16: Le conflit d'intérêt clients-prestataires

En résumé, il y a clairement un fossé entre les objectifs marketing des fournisseurs souhaitant offrir des services génériques ayant des mécanismes de sûreté de fonctionnement globaux, et les besoins spécifiques des clients qui veulent intégrer de tels services dans des applications critiques particulières nécessitant une sûreté de fonctionnement appropriée à leurs applications (voir Figure 1-16). Ce conflit d'intérêt permet de mettre en évidence une première problématique liée à la sûreté de fonctionnement des Services Web : celle-ci doit être adaptable suivant le type d'application et modulable dans le temps.

1.4.3 La sûreté de fonctionnement des Services Web

La sûreté de fonctionnement est un « aspect » fondamental des services Web. Cependant, celle-ci est variable suivant le type d'application et les critères du client. Les caractéristiques non-fonctionnelles sont donc indépendantes du comportement fonctionnel du service Web. Elles peuvent donc dans la plupart des cas, être externalisées et confiées à un module adéquat.

L'état de l'art que nous faisons dans la partie suivante présente les travaux de sûreté de fonctionnement nouveaux ou adaptés aux Services Web qui ont été mis en œuvre pour les architectures orientées services. Nous pensons que bien au-delà des mécanismes de sûreté de fonctionnement qui peuvent être déployés au niveau des Services Web, ceux-ci doivent pouvoir s'adapter au contexte applicatif dans lequel se trouvent les services. C'est sur ce point que nos travaux se distinguent des autres.

L'étude que nous allons effectuer maintenant est constituée de deux phases: la première consiste à détailler les travaux concernant la caractérisation des Services Web tandis que la seconde partie se concentre sur les mécanismes de sûreté de fonctionnement existants liés à cette technologie.

1.4.3.1 La caractérisation des Services Web

En ce qui concerne la caractérisation de la sûreté de fonctionnement des services Web, nous devons considérer les défaillances qui peuvent se produire en opération, et qui peuvent être dues à un certain nombre de types de fautes. Pour nous convaincre, nous avons effectué quelques expériences avec différents Services Web pendant une période courte (72 heures) et nous avons observé que des défaillances sont réellement fréquentes (voir Figure 1-17).

Les défaillances observées sont, la plupart du temps, dues à des fautes d'interaction transitoires et ne reflètent pas la QoS de chaque service cible. En outre, le taux d'échec observé peut énormément évoluer sur une longue période. Néanmoins, ceci montre le type

d'imprévus auquel nous devons faire face en développant une application à grande échelle sur des Services Web. Bien que l'identification des modes de défaillance selon différents modèles de fautes ait été faite pour les systèmes d'exploitation [39], les micronoyaux temps réel [40] et les intergiciels comme CORBA [41, 42], de tels types d'analyses n'ont pas encore été véritablement approfondis dans les Services Web. La raison en est très simple: il n'y a pas d'intergiciel spécifique au développement des Services Web. On peut noter toutefois que la mise en place d'un Service Web est généralement réalisée à partir de la combinaison d'un serveur d'application avec un parseur SOAP. En effet, contrairement aux logiciels CORBA où l'on peut recenser approximativement une dixaine d'ORBs (TAO, ORBacus, Orbix, Mico, ORBit, omniORB, javaORB), le nombre de solutions et donc de supports exécutifs aux Services Web est en fait non-borné.

			Type d'erreurs								
	Nombre de requêtes	Nombre d'erreurs	(502)Bad Gateway	Could not translate. Network problem.	java.lang.NullPointer Exception	Connexion refusée	Connexion ré-initialisée par le correspondant	(404)Not Found	Connexion terminée par expiration du délai d'attente	Aucun chemin d'accès pour atteindre l'hôte ciblé	Service hang
BabelFish	3423	2	0	2	0	0	0	0	0	0	0
FedEx	3406	3	0	0	3	0	0	0	0	0	0
Google	3308	28	28	0	0	0	0	0	0	0	0
MSN Search	3327	0	0	0	0	0	0	0	0	0	0
Amazon – US	2350	0	0	0	0	0	0	0	0	0	0
FraudLabsWebService	3501	1	0	0	0	1	0	0	0	0	0
Temperature ConvertService	3362	260	0	0	7	8	2	202	40	1	0
TimeService	178	1	0	0	0	0	0	0	0	0	1

Figure 1-17: Exemple d'erreurs de Services Web collectées sur une période de 72 heures

Nous pouvons, cependant, mentionner le travail effectué sur la plate-forme d'OGSA basée sur l'utilisation des Services Web, dans lequel les modes de défaillance ont été obtenus en utilisant des techniques d'injection de fautes (SWIFI) [43-45]. Ces travaux ont permis la création de l'outil WS-FIT (*Web Services Fault Injection Tool*) [43]. WS-FIT est un injecteur de fautes qui permet d'évaluer le protocole SOAP. Actuellement, WS-FIT instrumente le parseur SOAP Axis 1.1 en injectant des fautes sur réception d'une requête cliente ou envoie d'une réponse. Cet injecteur a été utilisé sur un système basé sur une simulation d'un chauffage autorégulé.

Ces travaux sur la caractérisation ainsi que les études réalisées dans [46] et [47] ont prouvé que beaucoup de défaillances peuvent altérer des Services Web en exécution. Au delà des fautes physiques de base qui peuvent affecter les noeuds exécutant les services, on peut remarquer que les fautes de communication sont une source d'erreurs significatives dans des applications à grande échelle sur le Net [5, 6]. Plus important, en raison de la complexité des couches de base nécessaires à l'exécution des Services Web, les fautes du logiciel doivent être considérées comme une première source de problèmes. Parmi les divers composants logiciel

et pour donner juste un exemple, l'analyseur SOAP a une importance primordiale et peut être sujet à des fautes de développement comme l'analyse incorrecte des messages ou une mauvaise traduction des types de données, ou ignorer certains codes d'erreurs retournés par le système d'exploitation. De récents travaux (2006) concernant le vieillissement logiciel dans les serveurs basés sur le protocole SOAP [48], ont validé des résultats très intéressants sur un des parseurs SOAP les plus utilisés : le parseur Axis. En effet, Axis 1.3 contient d'énormes fuites de mémoire et donc se dégrade très rapidement sur une courte fenêtre d'exécution menant a de sévères défaillances et à des situations de gel du serveur qui nécessitent une intervention manuelle pour le relancer.

En résumé, le développement d'applications semi-critiques à base de services Web doit prendre en compte beaucoup de sources de fautes possibles :

- Les fautes physiques affectant les ordinateurs et l'infrastructure matérielle de la gestion de réseau ;
- Les fautes du logiciel affectant les composants supportant l'exécution du Services Web (OS, serveurs d'application, parseurs SOAP, etc.) ;
- Les fautes d'évolution liées à une inconsistance entre les versions courantes du documents WSDL et les stubs existants produits par des versions plus anciennes ;
- Les fautes d'interaction agissant sur le point d'accès du service, comme, par exemple, un point d'accès inaccessible, inexistant ou changé;
- Les fautes de communication menant à la perte, l'omission ou la duplication du message;

En bref, le Web, épine dorsale des architectures orientées services, augmente les sources possibles de fautes qui altèrent l'interaction entre les clients et les fournisseurs. Réciproquement, le Web peut également être vu comme une opportunité d'un point de vue de la sûreté de fonctionnement, grâce à la disponibilité de services semblables et des multiples canaux de transmission.

En définitive, on peut se rendre compte que la caractérisation des services Web est un vaste sujet en chantier. A ma connaissance, aucun fournisseur de Services Web n'a officiellement recensé les différents cas de défaillances survenues dans le cycle de vie de son service et comme on a pu le voir, ceux-ci peuvent survenir à n'importe quel moment. De récents travaux ont permis de définir les SLA (*Service Level Agreement*). Un SLA est un accord de qualité de service entre un client et un prestataire. Ce contrat définit les engagements de l'hébergeur quant à la qualité de sa prestation, et les pénalités engagées en cas de manquement. Cette qualité doit être mesurée selon des critères objectifs acceptés par les deux parties (ex : temps de rétablissement du service en cas d'incident). WS-Agreement [49] est une grammaire XML décrivant un SLA dans un environnement de Grid Computing. WS-Agreement est soutenu par le groupe de travail GRAAP (*Grid Resource Allocation and Agreement Protocol*). La spécification n'est encore qu'un « *draft* » à l'heure actuelle. Toutefois, la notion de la « Qualité de service » est un enjeu déterminant au bon développement d'architecture orientée service. De nombreuses recherches se concentrent d'ailleurs sur cette question [50-54].

1.4.3.2 Les mécanismes de sûreté de fonctionnement des Services Web

Parallèlement aux recherches sur la caractérisation et la QoS, la mise en place de mécanismes sûrs de fonctionnement est un aspect prépondérant dans la bonne évolution des Services Web. Elle constitue à l'heure actuelle un frein à leurs développements. De gros

efforts de recherche sont effectués dans ce sens là. Ces efforts doivent se faire sur toutes les couches participant à la réalisation de la communication client-prestataire.

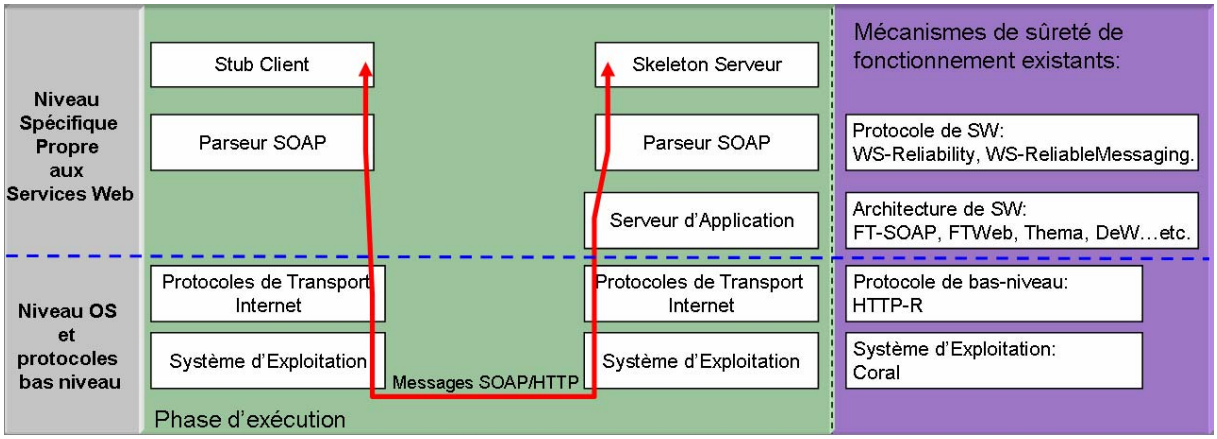

Figure 1-18: Mécanismes de sûreté de fonctionnement existant dans les services Web

La Figure 1-18 présente à quel niveau se placent les différents travaux de sûreté de fonctionnement que nous présentons ci-dessous. Comme on peut le voir, ces travaux peuvent être classifiés en deux catégories:

- les travaux génériques au niveau OS et protocole bas niveau qui permettent de renforcer la fiabilité des couches basses de communication. Ces travaux ne sont pas nécessairement spécifiques aux AOS mais peuvent être parfaitement agrégés à ce type d'application.
- les travaux spécifiques propres aux Services Web. Ces recherches ont permis la création de plates-formes spécifiques pour la mise en place de certains types de recouvrement ainsi que de nouvelles extensions aux protocoles SOAP pour assurer un niveau de sûreté de fonctionnement plus élevé.

Bien que la disponibilité et la fiabilité soient des exigences essentielles pour les applications critiques, les travaux que nous présentons ici sont malgré tout très récents. La section qui suit parcourt toutes les couches fonctionnelles présentées dans la Figure 1-18 et expose les travaux de recherche qui s'y trouvent.

Le modèle de la plate-forme ***CORAL*** (*Connection Replication and Application-Level logging*) proposé dans [55-58] effectue des changements sur le noyau du système d'exploitation (linux) et du serveur Web pour fournir des mécanismes de tolérance aux fautes qui soient transparents pour le client. Les mécanismes implémentés ici sont la réplication active et le « *log* » de messages. Dans ce modèle, chaque requête reçue par le serveur « *leader* » est enregistrée et envoyée à un serveur de secours (« le *follower* »). Les changements effectués au niveau du noyau fournissent l'exécution d'un mécanisme multicast permettant aux requêtes d'être envoyées à la fois au leader et au follower. Les changements effectués sur le serveur Web permettent la manipulation et la génération des réponses aux clients. L'inconvénient majeur de cette plateforme vient du fait que comme elle modifie le système d'exploitation, elle est fortement intrusive.

Le protocole ***HTTP-R*** [59] est un projet ayant pour but de garantir la fiabilité de l'échange entre un client et un prestataire. En effet, HTTP ne prévoit pas de mécanisme permettant de s'assurer qu'un message est bel et bien arrivé à destination et qu'il n'a été délivré qu'une fois. De fait, avec HTTP, des paquets d'informations peuvent être perdus sans qu'il soit donné suite à cette défaillance. Ces "pertes" sont sans gravité quand il s'agit simplement de délivrer à un

utilisateur une page d'information. En revanche, une telle défaillance peut avoir des conséquences fâcheuses si l'émetteur est une application qui attend une confirmation pour valider une transaction. Le problème est ardu: il ne suffit pas d'identifier qu'un message n'a pas été délivré; il faut aussi pouvoir le réexpédier, ce qui suppose d'en avoir conservé une copie quelque part; en outre, cette réexpédition doit, par exemple, respecter un créneau horaire... Pour garantir une telle intégrité, émetteur et récepteur doivent être informés avec précision du statut (de l'état) du traitement. Utiliser le Web comme un canal d'échange de messages entre applications demande donc de lui apporter certains mécanismes. Telle est la vocation de HTTP-R qui intègre, dans l'entête des messages, beaucoup plus d'informations de statut que ne le fait HTTP. En s'appuyant sur ce protocole, les gestionnaires de messages et les différents agents chargés de stocker le statut des messages pourront appliquer des règles afin de réagir aux éventuelles défaillances, qu'elles proviennent d'un serveur ou d'un réseau. Une approche comme HTTP-R n'a de chances de s'imposer à grande échelle que si elle est prise en main par un organisme comme l'IETF (*Internet Engineering Task Force*). Elle pourrait alors se transformer en une extension fiable du protocole HTTP, ce qui en même temps pourrait combler ses lacunes et permettrait ainsi une large diffusion et, à terme, sa banalisation.

L'architecture ***DeW*** (*A Dependable Web Service Framework*) [60] a été proposée pour améliorer la sûreté de fonctionnement des Services Web. Celle-ci peut être comprise comme registre contenant l'adresse des différentes copies d'un Service Web. Ce registre garantit l'indépendance de localisation physique. Ceci permet à une application basée sur les Web Service de continuer à s'exécuter tant que la référence d'une copie disponible est accessible par le registre. Cette architecture permet la continuation d'une opération en présence de défauts affectant les Services Web ou lors de leur migration. Le projet Active-UDDI [61] peut être considéré comme une approche semblable à celle-ci.

FT-SOAP [62] propose une plate-forme permettant à un prestataire d'effectuer une réplication passive sur un groupe de Services Web répliqués. L'utilisation d'intercepteurs dans la couche SOAP du client permet la redirection des requêtes vers les répliques en cas d'une défaillance du primaire. Sur le serveur, des intercepteurs s'ajoutent sur les composants pour effectuer le log, la détection des fautes et la gestion des répliques. Le transfert d'état est effectué de manière interne. A ce titre, la réplication passive de Service Web est également implémentée dans le projet JULIET reposant sur le framework .NET [63] et dans [64]. Ces implémentations imposent l'installation de couches applicatives à la fois chez le client et le serveur pour pouvoir les mettre en œuvre. Elles sont donc intrusives et nuisent à l'interopérabilité des AOS, ce qui est contraire à leur définition.

FTWeb [65] est une infrastructure capable de réaliser la réplication active entre les clients et les prestataires. Ce projet s'appuie sur la norme FT-Corba [66] ainsi que sur un algorithme d'ordonnancement total présenté dans [67] et WS-Reliability [68] pour effectuer l'ordonnancement des messages et assurer la cohérence des répliques. Ce projet nécessitant également d'introduire une couche applicative chez le client et le prestataire, il souffre du même problème d'intruisivité. Cependant, l'inconvénient majeur de cette architecture vient du fait qu'elle est très dépendante du middleware utilisé et impose de ce fait que les services Web des prestataires soient implémentés par des objets Corba. La plate-forme présentée en [69] est une architecture équivalente.

Thema [70] est un middleware pour Services Web capable de tolérer les fautes byzantines. Une couche applicative est insérée sur le client et dans chaque prestataire. Cette couche est une librairie reposant sur les travaux effectués dans [71] permettant d'introduire le canal de

communication adéquat et les politiques de consensus. Les problèmes de consensus sont également abordés dans [72].

WS-Reliability [68], créé par Sun, est le concurrent direct de HTTP-R. L'objectif de WS-Reliability est la prise en compte de la fiabilité de l'échange au niveau message. Le constat de départ est que la simple liaison SOAP/HTTP n'est pas suffisante lorsqu'un protocole de message au niveau applicatif doit aussi se conformer à des contraintes de sécurité et de fiabilité. Ainsi, WS-Reliability garantit la livraison, la non-duplication et l'ordonnancement des messages. L'ambition affichée de WS-Reliability est de constituer une proposition initiale qui permette de démarrer un processus de spécification et d'implémentation du même type que WS-Security. Il est difficile de trancher et de dire si un choix doit être effectué entre le protocole de transport (HTTP-R) ou le protocole de message (WS-Reliability). Tout au plus peut-on se borner à dire que, de façon générale faire descendre une fonction technique dans les couches basses d'une architecture produit toujours une amélioration de la fiabilité et de la performance. En conclusion, aucune des deux spécifications n'a encore atteint le niveau de maturité et d'acceptation nécessaire à une adoption large de ce type de technologie. WS-ReliableMessaging [73] de Microsoft est une spécification analogue qui permet de garantir les mêmes propriétés que WS-Reliability.

WSCAL (Web Service Composition Action Language) est un langage de composition de service basé sur XML qui a été conçu pour la tolérance aux fautes et notamment le recouvrement par poursuite (Forward Error Recovery). Dans le cadre des Services Web, il est préférable d'effectuer un recouvrement par poursuite plutôt qu'un recouvrement arrière. Une des principales raisons est que, en plus de la latence du Web, le verrouillage des ressources jusqu'à la terminaison d'une transaction est en général peu approprié pour les Services Web du fait qu'ils ont potentiellement un grand nombre de clients concurrents qui ne peuvent pas attendre trop longtemps. Certains langages de composition (BPEL4WS [11], WSCI [74]) proposent les transactions compensatoires pour revenir à un état quasi-équivalent et consistant. Cependant, ce mécanisme appliqué à tous les participants d'une transaction peut produire l'effet domino (cascaded roolback - problème bien connu des systèmes distribués). Le recouvrement par poursuite est spécifié en terme d'actions co-opératives, construit sur le concept d'actions atomiques coopératives (CA), s'appuyant sur les mécanismes d'exception de chaque Service Web. Le langage WSCAL proposé dans l'article, crée un co-ordinateur possédant un arbre d'exception. Le coordinateur englobe tous les Services Web de la composition. C'est un Service Web qui va jouer le rôle de proxy (d'intermédiaire) entre le client et les différents services de la composition. Dans le cas d'une levée d'exception sur un service, il sera capable, grâce à l'arbre d'exceptions, de déterminer si cette exception a un impact sur les autres services de la composition ou non. Si c'est le cas, il pourra effectuer des transactions compensatoires ou autres actions de recouvrement, sur les services impactés. Le langage WSCAL n'a à l'heure actuelle aucune implémentation concrète.

1.5 Récapitulatif

En conclusion, les Services Web s'installent pour durer. Ils sont le résultat d'une évolution technologique majeure, et les bases d'une révolution organisationnelle et économique. Le changement est considérable : les Services Web vont modifier en profondeur l'organisation du travail et les processus métier des organisations (entreprises, administrations, associations). En outre, ils vont changer les pratiques des professionnels de l'informatique.

L'évolution technologique des Services Web surgit de la convergence de plusieurs axes de développement (les protocoles Internet, le langage XML, la progression rapide de la

puissance des machines et de la bande passante) vers un ensemble de technologies qui, n'ayant pas un contenu technique révolutionnaire à proprement parler, apportent un niveau de standardisation et d'interopérabilité sans précédent dans des domaines clés comme l'informatique de gestion ou le commerce électronique.

À l'heure actuelle, la plupart des industries utilisatrices ne perçoivent pas encore les opportunités de changement que la disponibilité de cette technologie apporte. Les professionnels de l'informatique ont eux aussi des difficultés à percevoir la profondeur des transformations induites par l'adoption de la technologie des Services Web dans le métier de leurs clients et, par conséquent, dans leur propre métier. Les Services Web sont vus souvent comme « encore une autre technologie de middleware », un concurrent ou au mieux, un successeur de CORBA et DCOM, une trouvaille technologique de Microsoft et IBM pour relancer la croissance et les investissements. Ce point de vue n'est pas simplement réducteur, il est tout bonnement erroné.

En fait, au coeur de l'évolution technologique portée par les Services Web, il y a la montée en puissance du concept de service, qui reste une notion encore mal appréhendée. Nous avons vu que les concepts de services et d'architectures orientées services sont théoriquement indépendants des technologies des services Web. L'exercice qui consiste à concevoir et à décrire un système informatique étendu sous forme d'une architecture orientée services, indépendamment de l'utilisation ou non des technologies des services Web, est sans doute très utile et bénéfique, car il correspond à une bonne préparation à son évolution en vue de son ouverture. Cependant, il n'est pas réaliste de penser que ces deux notions vont rester longtemps séparables, qu'on pourra mettre en œuvre des architectures orientées services au moyen de technologies autres que les Services Web. Les technologies des Services Web (SOAP, WSDL, etc.) vont devenir aussi universelles et omniprésentes que TCP/IP, HTTP, HTML, XML. Pour simplifier, on peut considérer qu'il y a identité entre architectures orientées services et architectures de Services Web et que les technologies des Services Web représentent le moyen d'élection pour mettre en oeuvre des architectures orientées services.

SOAP, WSDL et UDDI constituent une infrastructure de base qui permet la mise en oeuvre de services et d'architectures avec une sécurité point à point, sans exigences poussées de fiabilité, de sécurité ou de gestion transactionnelle. Cette base est en revanche insuffisante pour l'automatisation des processus métier critiques, qui est à terme, la cible des technologies de services Web.

Les nombreux travaux de recherche que nous venons d'exposer dans ce chapitre confirment la jeunesse des Services Web et tente de ce fait de renforcer la sûreté de fonctionnement de cette technologie. Nous pensons que la clé de voûte des Services Web repose sur la notion de contrat. En effet, bien au-delà des mécanismes de recouvrement que l'on peut instaurer ou adapter aux Services Web, nous pensons que ces mécanismes sont dépendants d'un contexte applicatif donné. Ils doivent donc pouvoir être instaurés de manière opportune lorsque cela devient nécessaire. Hormis pour FT-SOAP, où le document WSDL est modifié pour y afficher les répliques utilisées, aucune des autres architectures présentées ne fait référence au contrat de service. Ces travaux fournissent une solution globale pour instaurer un mécanisme de recouvrement qui force ensuite toutes les exécutions du service reposant dessus à consommer les ressources et le temps de calcul indispensables pour mettre en œuvre cette stratégie même lorsque cela n'est pas nécessaire. Un autre inconvénient des propositions soumises vient du fait qu'elles contraignent le client et/ou le prestataire à installer un dispositif particulier sur leur nœud de communication. Ces infrastructures sont donc intrusives et peuvent au final introduire des problèmes d'interopérabilité.

La plate-forme que nous présentons dans ce mémoire, nommée IWSD (*Infrastructure for Web Service Dependability*), est une solution non-intrusive qui s'appuie totalement sur les concepts et les protocoles de base qui forment la pierre angulaire des Services Web. C'est une infrastructure générique pouvant s'exécuter sur un nœud distant différent et indépendamment du dispositif mis en place par le client ou le prestataire. Sur cette plate-forme, un client ou un prestataire peut déployer un connecteur spécifique conçu pour un contexte d'utilisation particulier. Ce connecteur est capable de capturer l'interaction de Services Web et d'effectuer des actions de tolérance aux fautes implémentées spécifiquement pour l'AOS dans laquelle il est défini. Les actions de tolérance aux fautes sont doubles:

- des assertions exécutables effectuant des vérifications d'entrée/sortie pour le confinement d'erreur et,
- des mécanismes de recouvrement d'erreur selon un modèle de défaillances donné.

Les dispositifs de sûreté de fonctionnement qui peuvent être mis en application de cette façon dépendent de plusieurs paramètres, incluant les dispositifs spécifiques du Service Web cible. La vérification peut être différente pour un client qui souhaite se protéger d'informations potentiellement corrompues retournées par un service erroné, ou réciproquement, pour un fournisseur qui veut protéger son service contre des clients mal-attentionnés ou malveillants. Quant au recouvrement, il est dépendant de la nature du fournisseur (avec ou sans état) et évidemment des modèles de fautes et de défaillances considérés. Bien que de tels mécanismes de tolérance de fautes puissent être conçus en fonction du service cible, il est cependant possible de concevoir un cadre global fournissant la séparation des préoccupations (« *separation of concerns* ») entre les applications et les mécanismes de sûreté de fonctionnement. Un des principaux atouts de cette plate-forme est que ces mécanismes de tolérance aux fautes peuvent être définis de manière spécifique pour chaque Service Web et par chaque utilisateur qui le souhaite.

2 IWSD : Une plate-forme pour la sûreté de fonctionnement des Services Web

Ce chapitre présente les différents composants de la plateforme IWSD. Parmi eux, le connecteur joue un rôle capital. Cette section va permettre, au travers de la définition des objectifs et de la spécification d'un tel composant, d'analyser la valeur ajoutée de ce module lorsqu'il est inséré dans une application orientée services.

2.1 Introduction

Le chapitre 1 a permis de mettre en évidence que les Services Web ne sont pas une technologie révolutionnaire mais plutôt évolutionnaire dans le sens où, par rapport, aux technologies voisines telle que la programmation orientée objet avec Corba ou DCOM, ils permettent de réaliser des applications à grande échelle sur le Net. En effet, le principal avantage des Services Web mais aussi le plus gros inconvénient vient du fait qu'ils sont transportés sur des protocoles standards Internet ouverts tel que le HTTP. Ils peuvent ainsi facilement passer les pare-feux puisque ceux-ci ont, généralement, le port 80 ouvert pour lire les pages Web. Cette propriété permet bien sûr de faciliter la mise en place de ce type d'application mais permet également de contourner les technologies de protection anti-intrusion existantes les rendant ainsi obsolètes, ou dans tous les cas inefficaces. Il est clair que recevoir une réponse SOAP, contenant des paramètres de sortie qui seront analysés et interprétés par une application cliente et qui permettront par la suite de lancer, par exemple, des appels systèmes critiques, n'a pas du tout le même impact que de télécharger une page HTML sur son navigateur Web.

De même, un prestataire défaillant ou malveillant pourrait fournir à ses clients un contrat WSDL erroné, ne respectant pas les engagements qu'il prétend définir (ex : le Service Web de l'addition pourrait effectuer par moment une soustraction). Les Services Web n'étant rien d'autre que des points d'accès, des ressources atteignables via le Net, celles-ci peuvent être à tout moment corrompues, défaillantes ou détruites.

Notre proposition se base sur la notion de « connecteur ». Un connecteur est un composant logiciel qui relie un client à un prestataire. Il permet dans un premier temps de fournir à ces deux acteurs des moyens pour équiper les Services Web de mécanismes adaptés et efficaces de détection d'erreurs, c'est-à-dire, pour transformer un Service Web en un composant logiciel auto-testable [7, 8]. Dans un deuxième temps, il réalise des mécanismes intégrés à base de répliques pour effectuer le recouvrement d'erreur. Dans le cas où la couverture d'hypothèses telle que celle de silence sur défaillances est forte, la procédure de recouvrement peut s'implémenter par une commutation simple vers un autre composant disponible. Si le modèle de défaillance prend en compte les fautes en valeur alors des stratégies de masquage deviennent nécessaires.

Ces connecteurs spécifiques de tolérance aux fautes (CSTF) sont définis par l'utilisateur. L'utilisateur peut être un client, un fournisseur de Services Web ou n'importe quel utilisateur tiers intéressé par des mécanismes de tolérance aux fautes. Celui-ci est défini en tant que composant logiciel capable de capturer les interactions de Services Web et d'effectuer des actions de tolérance aux fautes. Son rôle est triple (voir la Figure 2-1) :

1. il effectue des assertions exécutables en appliquant des vérifications sur les requêtes d'entrée-sortie pour le confinement d'erreur,
2. il réalise des actions de recouvrement pour la tolérance aux fautes, grâce à un modèle de défaillance donné et selon des caractéristiques de gestion d'état des Services Web ciblés,
3. il contrôle, enfin, la disponibilité des points d'accès utilisés.

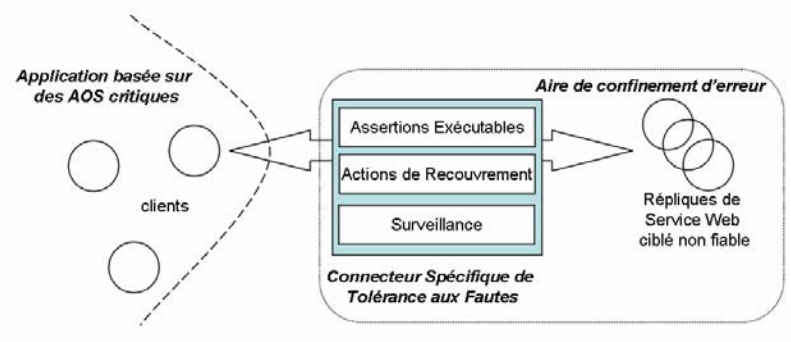

Figure 2-1: Rôle du connecteur

Les caractéristiques de sûreté de fonctionnement, qui peuvent être implémentées de cette façon, dépendent de plusieurs paramètres, dont des caractéristiques spécifiques du Service Web qui est sous contrôle. La vérification de l'interaction pourrait être différente pour un client qui vise à se protéger contre une information corrompue retournée par un service, ou réciproquement, pour un fournisseur souhaitant protéger son service contre des clients malveillants.

2.2 Présentation de la plate-forme

La gestion et l'exécution des connecteurs s'effectuent sur une plate-forme spécifique qui est une infrastructure insérée entre les clients et les fournisseurs. Le stockage, la recherche, le chargement et l'exécution des connecteurs spécifiques de tolérance aux fautes sont soutenus par la plate-forme IWSD (*Infrastructure for Web Services Dependability*). A l'exécution, la plate-forme fournit le support pour exécuter les mécanismes de tolérance aux fautes inclus dans le connecteur.

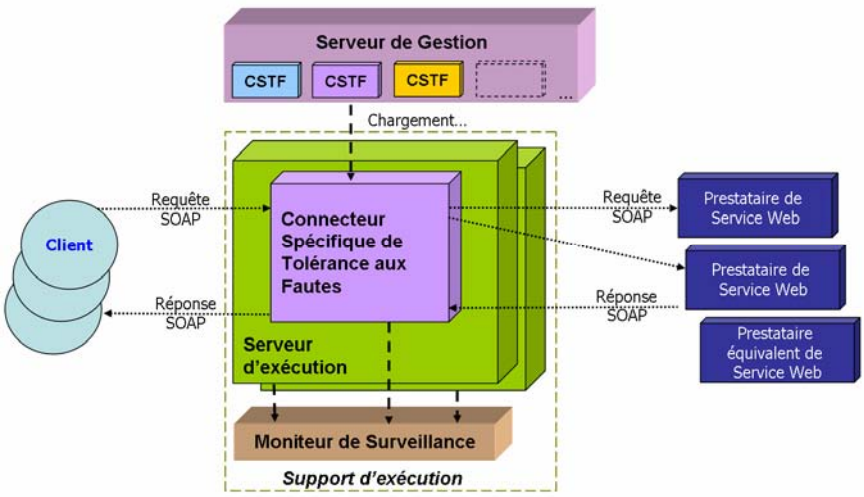

Figure 2-2: IWSD, une infrastructure pour la sûreté de fonctionnement des Services Web

La plate-forme IWSD (voir la Figure 2-2) est donc constituée de composants logiciels, chacun ayant des responsabilités bien définies:

- Le ***connecteur*** est l'élément central de la plate-forme. Il traite les requêtes et les réponses fournies par les clients et les prestataires contactés en effectuant les actions de tolérance aux fautes définies par l'utilisateur.
- Le ***support d'exécution*** peut être considéré comme une machine virtuelle sur laquelle s'exécutent les connecteurs. Il est composé de deux modules distincts :
    - Le ***serveur d'exécution*** intercepte les requêtes des clients et charge le connecteur souhaité à partir de l'analyse du message reçu.
    - Le ***moniteur de surveillance*** est en charge de vérifier l'état de la plate-forme IWSD et d'évaluer l'état courant des Services Web en exécution au travers de messages (rapport d'erreur ou autre) émis par les connecteurs.
- Le ***serveur de gestion*** permet de créer et gérer les comptes utilisateurs. Il permet également la création et le stockage des connecteurs associés à chacun d'eux.

Cette plate-forme ayant pour but d'assurer la sûreté de fonctionnement d'application à base de Service Web doit, de ce fait, être tolérante aux fautes. Pour cela, il est impératif d'utiliser des solutions de sûreté de fonctionnement déjà bien établies. Par exemple, elle peut être implémentée en mode duplex pour tolérer les fautes de crash. Comme nous maîtrisons son implémentation, elle peut être rendue tolérante à des modèles de fautes plus complexes, ce qui n'est pas en soi un sujet de recherche car des solutions classiques peuvent être mises à profit.

Cette architecture doit obéir également à une contrainte importante : la solution proposée ne doit pas être intrusive ni pour le client, ni pour le fournisseur de Service Web. La raison de cette contrainte forte s'appuie sur le fait que les Services Web se basent sur des protocoles ouverts (cf. chapitre 1) et sont totalement détachés de toute implémentation. Les applications clientes et prestataires sont donc considérées comme des boîtes noires. Cette contrainte nous impose de considérer les applications participantes comme des entités non modifiables. Nous posons donc comme simple hypothèse que les clients et les prestataires possèdent un parseur SOAP pour pouvoir sérialiser ou désérialiser les messages SOAP.

Les phases d'enregistrement, de consultation et d'exécution d'un connecteur sont totalement prises en charge par le IWSD. Le serveur de gestion permet d'enregistrer et de rechercher un connecteur pour un Service Web spécifique. Pendant la phase d'exécution, le connecteur est lancé sur un serveur d'exécution qui le décompose en deux sous-traitements :

- les ***actions de pré-traitements*** : ces actions sont effectuées sur réception d'une requête cliente et permettent, entre autres, de faire de la détection d'erreur.
- les ***actions de post-traitements*** : ces actions sont exécutées sur réception de la réponse du ou des prestataire(s) correspondant à la requête reçue. Elles sont principalement chargées d'effectuer un signalement ou un recouvrement d'erreur.

Ces actions sont spécifiques à un connecteur. En effet, un Service Web peut être impliqué dans diverses applications où la criticité est un facteur variable dans le temps, suivant le client qui l'utilise et le type d'application. Pour chacune de ces contraintes, on peut rattacher au Service Web un ou plusieurs connecteurs.

L'utilisateur doit donc avoir la possibilité de créer et d'insérer dans son connecteur, des actions qu'il a lui-même prédéfinies en fonction du type d'application dans lequel il souhaite

intégrer son service. La section qui suit est consacrée à présenter les objectifs et caractéristiques des connecteurs.

2.3 La notion de connecteur

2.3.1 Objectifs et Spécifications

La notion de connecteur est un concept classique des ADLs (*Architecture Description Language*) permettant de rendre explicite les interactions entre composants [75]. L'objectif du connecteur, dans notre contexte, est d'améliorer la sûreté de fonctionnement des applications à base de Services Web en insérant, dans la communication entre le client et le prestataire, des actions de tolérance aux fautes. Le connecteur doit donc être capable de répondre à certaines exigences techniques. Il doit être capable de réaliser les actions suivantes:

- La ***détection d'erreur*** : vérification de type, assertion exécutable sur les messages SOAP d'entrée/sortie, ...etc,
- Le ***signalement d'erreur*** : levée et envoi d'une exception SOAP au client,
- Le ***recouvrement d'erreur*** : mise en place de différentes stratégies de réplication,
- Le ***diagnostic d'erreur*** : collecte d'informations d'erreurs sur les points d'accès contactés.

Ces actions doivent être définies par n'importe quel utilisateur de la plate-forme pour n'importe quel service ciblé et dans n'importe quel contexte d'applications donné. Afin de garantir la sûreté de fonctionnement de la plate-forme IWSD, le connecteur, lui-même, doit, bien sûr, être fiable. Enfin, il doit également pouvoir être intéropérable en s'interfaçant avec n'importe quel client de Services Web. Pour cela, un contrat WSDL rattaché au connecteur créé est fourni. En effet, n'importe quel Service Web étendu avec des mécanismes de sûreté de fonctionnement au moyen de connecteurs peut être considéré comme une version sûre de Services Web. Cela signifie que les fonctionnalités de cette version étendue peuvent être rapportées dans un contrat WSDL. Cette version plus sûre du service peut être décrite dans ce document et rendue visible aux utilisateurs dans un annuaire UDDI. La conclusion très importante de ce fait, est que les utilisateurs peuvent ainsi tirer bénéfice de ces services Web sûrs s'ils répondent aux exigences non fonctionnelles de leur application.

Chaque connecteur possède donc son propre contrat WSDL. Ce document est une version plus sûre du document WSDL original incluant, entre autre, la description de pré et post actions mises en oeuvres par l'utilisateur ainsi que les stratégies de recouvrement possibles et les répliques utilisées (cf. Figure 2-3).

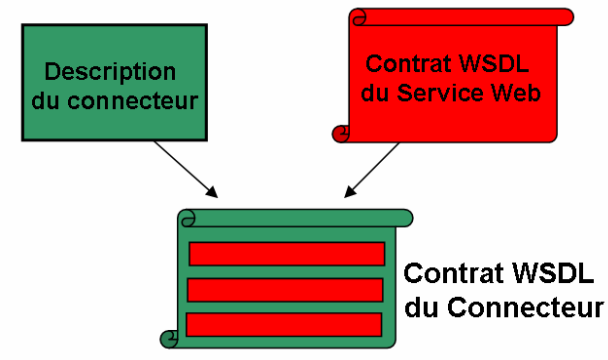

Figure 2-3: Le contrat WSDL du connecteur

Plusieurs documents WSDL peuvent donc exister pour le même Service Web fonctionnel: (i) le document original décrivant seulement l'interface du service, et (ii) les documents produits par IWSD décrivant plusieurs instances de connecteurs. Ainsi, d'autres clients recherchant un tel Service Web ont trois options à leur disposition:

- Utiliser le service web original, ou probablement, le connecteur du prestataire.
- Utiliser une version publique d'un connecteur déjà développée par un client.
- Créer un nouveau connecteur répondant à leurs besoins.

Lors de la production du document WSDL du connecteur, un serveur d'exécution est indiqué par défaut pour créer le point d'accès du service empaqueté. Cependant, le cadre proposé ici permet à des clients de connaître et d'entrer en contact dynamiquement avec d'autres instances de serveur d'exécution.

2.3.2 Développement d'un connecteur

La description du connecteur est l'élément clé de cette architecture. Celle-ci est définie par l'utilisateur. Il est impératif qu'elle soit la plus sûre possible. La spécification de cette description doit donc garantir deux conditions essentielles :

- S'assurer que le connecteur créé à partir de cette description est exempt de fautes de développement. En effet, il n'est pas concevable d'insérer un composant potentiellement défaillant pour assurer la sûreté de fonctionnement d'une application;
- Permettre à un utilisateur de décrire facilement les actions de tolérance aux fautes appropriées aux Services Web. Cette propriété fondamentale conditionne la complexité de la description des connecteurs.

Pour respecter cette spécification, notre approche est basée, ici, sur les concepts des langages dédiés ou DSL (*Domain Spécifique Language*). En effet, les DSLs ont été étudiés depuis longtemps dans divers domaines d'application et ont montré des avantages significatifs en termes d'expressivité, de concision et de sûreté. Parmi eux, nous pouvons notamment citer des langages tels que Devil [76, 77] (un langage dédié à la spécification d'interfaces de programmation de périphériques), Plan-P [78] (un langage dédié permettant aux applications de programmer les routeurs du réseau), Spidle [79] (un langage pour la spécification d'application à base de « flux de données »), …etc.

Etant donné que les DSLs sont une solution bien reconnue pour résoudre les problèmes d'une famille de programmes [80, 81], ils peuvent être employés pour fournir les mêmes bénéfices pour décrire et implémenter des connecteurs spécifiques de sûreté de fonctionnement.

Ainsi, nous proposons un langage dédié nommé DeWeL (*DEpendable WEb Service Language*). DeWeL vise à décrire et à rendre plus fiable le développement des connecteurs spécifiques au cas-par-cas. Ce langage est essentiel car il permet de garantir que le code qui sera par la suite inséré dans le serveur d'exécution sera exempt de fautes. Dans notre architecture, un programme DeWeL compilé représente un connecteur, il contient toutes les actions que devra effectuer le serveur d'exécution lorsque celui-ci sera activé. La conception du langage DeWeL a deux objectifs principaux :

- Eviter l'introduction de fautes logicielles critiques dans les connecteurs créés pour garantir la sûreté de fonctionnement de l'application client et surtout, de la plate-forme IWSD sous-jacente ;

- fournir un ensemble fini de construction pour déclarer des stratégies de recouvrement et écrire des assertions exécutables afin de pouvoir réaliser les actions prévues dans la spécification du connecteur.

Le premier objectif est un problème bien connu des systèmes critiques. Plusieurs compagnies industrielles (par exemple IBM, HP, etc.) ont d'ailleurs classifié les fautes de développement logicielles usuelles. Par exemple, dans ODC (classification orthogonale des fautes) d'IBM [9], les types de fautes sont classifiés comme suit : assertions, vérifications de données et/ou de paramètres incorrects, problèmes d'exactitude algorithme, synchronisation des ressources, fonctions incorrectes. L'introduction de ces fautes peut être limitée par la réduction des possibilités du langage en termes de types de données et constructions algorithmiques. C'est exactement ce qui est imposé par des normes de conception et de codage pour le logiciel critique de sûreté, comme CENELEC 50128 pour les chemins de fer (c'est-à-dire, « restrictions des constructions de langage comme les objets dynamiques, la récursivité, les pointeurs, les entrées/sorties etc. ») ou recommandé dans la section des normes logicielles de DO-178/EUROCAE pour l'avionique. DeWeL offre les abstractions et notations appropriées menant à un pouvoir expressif restreint (pas de pointeurs, pas de fonctions, pas de boucles conditionnelles, seulement des expressions arithmétiques et logiques simples). Ceci permet à un utilisateur d'améliorer considérablement la qualité du code et de vérifier des propriétés critiques à la compilation comme la terminaison et l'usage prévisible des ressources. Ceci a également le mérite du point de vue de la sûreté de fonctionnement d'avoir un haut niveau de confiance dans le connecteur créé. Les restrictions du langage sont très utiles pour effectuer une vérification statique efficace et permettent de limiter à l'exécution le nombre de vérifications dynamiques à effectuer. La Figure 2-4 classifie les restrictions imposées au langage DeWeL et présente pour chacune d'entre elle, la propriété critique que l'on cherche à assurer.

Restrictions	Propriétés critique vérifiées
- Pas d'allocation dynamique - Pas de pointeur - Pas de référence	Contrôle total de la mémoire et des ressources
- Pas de création de fichier	
- Pas d'indexation de tableaux	
- Pas de boucle standard (while, for)	Terminaison
- Pas de fonction - Pas de surcharge de méthodes	
- Pas de construction récursive	
- Pas d'accès externe aux données des autres espaces utilisateurs ou aux ressources du système	Non-interférence

Figure 2-4: Les principales caractéristiques de DeWeL

Le deuxième objectif est de permettre à un utilisateur, malgré ces restrictions, de pouvoir décrire, concevoir et déployer un connecteur capable de réaliser les mécanismes de tolérance aux fautes cités ci-dessous:

- ***Programmation défensive :*** Le but est de s'assurer que l'utilisateur peut écrire des assertions sur les messages d'entrées/sorties, sans que les restrictions n'altèrent l'expressivité.

- ***Gestion des exceptions :*** Les erreurs détectées en provenance du prestataire doivent être capturées et traitées dans le connecteur en utilisant les expressions restreintes permises par le langage ou expédiées au client pour les traiter à un niveau d'abstraction supérieur.
- ***Sélection de la stratégie de recouvrement :*** Concernant les stratégies de recouvrement, seules des instructions déclaratives sont permises. Dans un connecteur, l'utilisateur voit des stratégies de recouvrement en tant que composants logiciels intégrés qui peuvent seulement être paramétrés. Ces stratégies sont discutées dans le chapitre 4. La détection d'erreur repose sur la vérification d'assertions d'une part, et la gestion des exceptions d'autre part. Celle-ci est à l'origine du déclenchement du mécanisme de recouvrement présélectionné par le client.

La façon de décrire ces différentes actions de tolérance aux fautes précitées dans un programme DeWeL est détaillée dans le chapitre 3. Ce chapitre présente également le processus de compilation pour générer un connecteur à partir d'un programme DeWeL.

Ces différentes actions sont essentielles pour pouvoir assurer la sûreté de l'application. Elles peuvent être communes à toutes les opérations d'un même service ou bien spécifiques et différentes à chaque opération.

La conception d'un langage implique donc de définir les constructions appropriées et de lui donner une sémantique. La description d'un programme DeWeL s'appuie ainsi sur un canevas qui peut être automatiquement généré à partir du document WSDL original du Service Web cible sur lequel s'appuie le connecteur. Un canevas est un moyen de séparer le fond (le contenu informationnel) de la forme (la manière dont il est présenté). Il agit comme un modèle dans lequel seuls certains éléments sont modifiables (le contenu). Cela facilite la conception et la mise à jour du contenu. La Figure 2-5 détaille la description de ce canevas. Elle expose également le canevas généré à partir du document WSDL partiel du service de Google présenté dans le chapitre 1 (cf. Figure 1-10). Celui-ci est ainsi composé de deux sections définies ci-dessous:

- ***Déclaration de traitements globaux :*** les instructions insérées par l'utilisateur dans cette zone impacteront toutes les opérations du connecteur.
- ***Déclaration des traitements spécifiques :*** Ce sont des traitements dans lesquels l'utilisateur décrit les actions à réaliser pour une seule opération de service. Ainsi, le point d'entrée d'un traitement spécifique est la signature de l'opération elle-même. Ces traitements sont destinés :
    - à la vérification et la manipulation des requêtes et des réponses SOAP de l'opération (***Pre-Processing*** et ***Post-Processing***),
    - à la gestion d'erreurs provenant d'un prestataire. Celles-ci peuvent être de deux types :
        - Soit une incapacité à dialoguer avec le prestataire (rupture de la communication, prestataire injoignable, message reçu incompréhensible par le connecteur, …etc.): ***CommunicationException***.
        - Soit un message d'erreur SOAP directement retourné par le prestataire suite à une erreur de son service : ***ServiceException***.
    - à la sélection des modes de recouvrement (***RecoveryStrategy***).

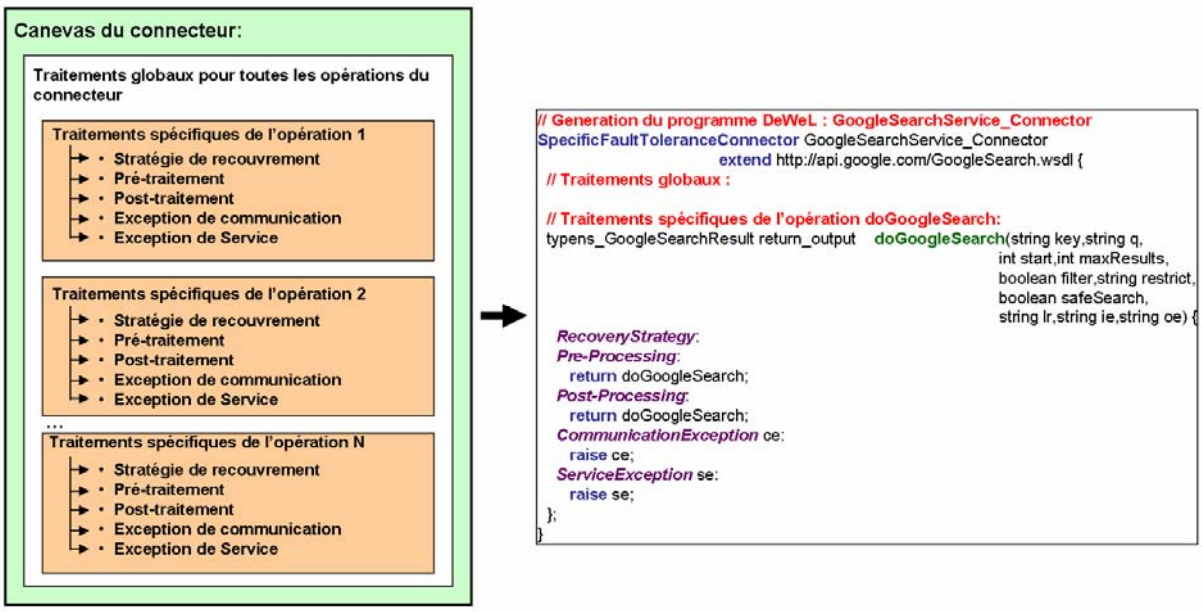

Figure 2-5: Le canevas du connecteur

Le canevas DeWeL est indispensable pour permettre à un utilisateur d'insérer ses propres actions de tolérance aux fautes. Cependant, on peut remarquer qu'un connecteur généré à partir d'un canevas sans assertion ni mécanisme de recouvrement définis par l'utilisateur, comme celui présenté en Figure 2-5, a quand même une utilité majeure. En effet, celui-ci est doté de mécanismes d'observation capables de monitorer le service ciblé. Il peut ainsi collecter des informations capitales (disponibilité, temps de réponse, fréquence et taux de défaillance du service) qui pourront par la suite lui permettre d'adapter les actions de tolérances aux fautes à insérer en fonction des données obtenues.

La sémantique dynamique des constructions du langage décrit le comportement d'un programme à l'exécution. Par exemple, la sémantique dynamique du langage C [82] définit la présence obligatoire d'une fonction de nom « *main* » qui sera la première appelée lors de l'exécution du programme. La sémantique dynamique d'un langage s'appelle le modèle d'exécution. Dans le cadre des programmes DeWeL, il existe plusieurs modèles d'exécution. Chacun de ces modèles est conditionné par la stratégie de recouvrement mise en place par l'utilisateur dans la section « ***RecoveryStrategy*** ». Ceux-ci déterminent quand et dans quel ordre seront effectués les différents traitements présentés dans le canevas.

Dans le cas standard où aucun mécanisme de recouvrement n'est activé, le modèle d'exécution par défaut est linéaire. Ce mode est présenté sur la Figure 2-6. Ce mode n'effectue aucun retour en arrière. Le traitement nominal est le suivant : Si le pré-traitement (***Pre-Processing***) a pu être effectué, la requête est envoyée au prestataire. Dans le cas contraire, cela signifie qu'une assertion exécutable définie par le client a levé une exception. Celle-ci est directement retournée au client. Sur réception d'un message SOAP correspondant à la requête, on effectue le post-traitement (***Post-Processing***) avant d'envoyer une réponse au client. Si une exception de communication (***CommunicationException***) ou de service (***ServiceException***) est levée, le traitement correspondant de l'utilisateur spécifié dans le programme DeWeL est réalisé avant d'envoyer une exception SOAP au client.

Une des spécificités des connecteurs dans ce modèle, c'est qu'ils sont éphémères. Ils sont chargés et actifs juste le temps du traitement de la requête et de la réponse associée. Cette

particularité se retrouve dans tous les modèles. Il est important de noter également qu'après l'envoi d'une requête sur une réplique et réception de la réponse, le connecteur se retrouve forcément sur un des trois traitements définis par l'utilisateur (*Post-Processing*, *CommunicationException* ou *ServiceException*). Ceci est vrai pour n'importe quel mécanisme de recouvrement mis en œuvre.

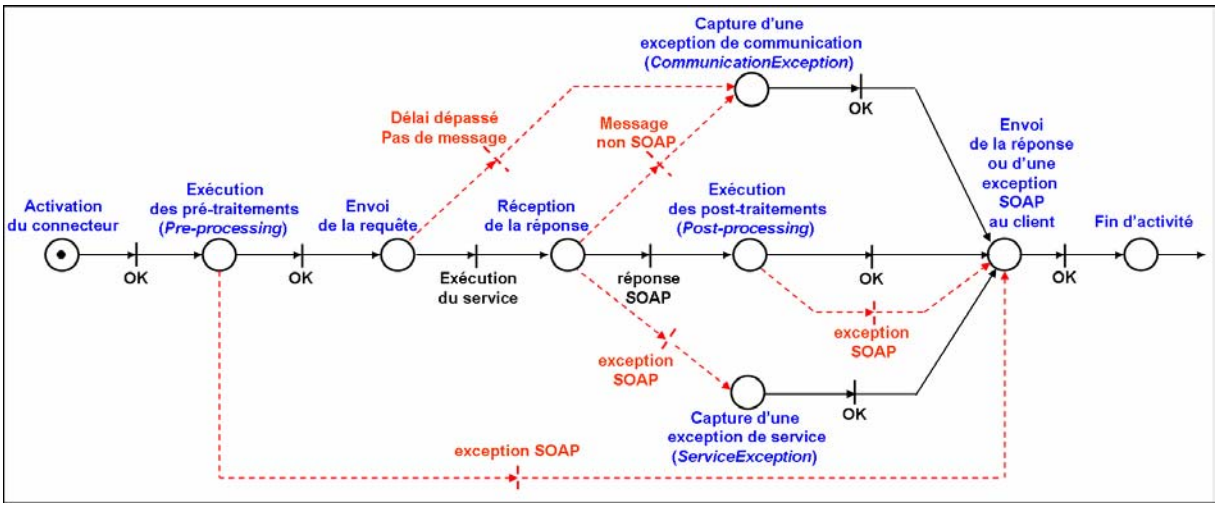

Figure 2-6: Le modèle d'exécution linéaire

La section qui suit présente ces différents mécanismes de recouvrement réalisables par le connecteur, les conditions et hypothèses nécessaires à leurs mises en œuvre ainsi que les modèles d'exécution rattachés à chacun d'eux.

2.3.3 Les mécanismes de recouvrement

En plus d'effectuer des actions de détection et de signalement d'erreur, le connecteur est capable d'appliquer des mécanismes de recouvrement déclarés par un utilisateur dans un programme DeWeL. La mise en place de mécanisme de recouvrement est un processus complexe qui nécessite de prendre en compte de nombreux problèmes. Le premier d'entre eux, est de pouvoir identifier et sélectionner des répliques incluses dans le recouvrement. En effet, pour être tolérant aux fautes, une des premières contraintes imposées au composant est d'être redondant. Notre domaine d'application étant Internet, nous nous retrouvons désormais face à une nouvelle opportunité : la redondance de ressources inhérente au Web. Internet est le substrat qui contient le plus de ressources mondiales. Les recherches actuelles sur les Ontologies [83, 84] ont pour but de donner du sens à ces ressources et de pouvoir par la suite les classifier. Cette approche, aussi appelée « *Web Semantic* », va permettre de rechercher et d'identifier plus facilement des ressources identiques ou équivalentes. Les Services Web, n'étant qu'en réalité, un point d'accès (c'est-à-dire une ressource sur le net), les travaux précités s'intègrent parfaitement à cette technologie. Les autres problématiques concernent les mécanismes eux-mêmes. Quels mécanismes peut-t-on proposer ? Sous quelles hypothèses et conditions ? Comment les rendre spécifiables par l'utilisateur ? Toutes ces questions seront discutées dans la suite de cette section.

2.3.3.1 Réplication et équivalence de services

Nous considérons, dans notre contexte, deux types de services qui peuvent être utilisés pour mettre en œuvre des mécanismes de recouvrement.

Tout d'abord, les ***services identiques*** qui correspondent à un unique document WSDL, avec des points d'accès différents (par ex, les répliques d'Amazon : US, FR, CA, etc.). Différentes implémentations du service peuvent aider à tolérer les fautes transitoires du service en exécution. Un simple basculement d'une réplique à une autre peut être fait dans ce cas précis.

Plus important, nous considérons également ce que nous appelons ici, les ***services équivalents*** : les documents WSDL sont différents mais peuvent être considérés comme fournissant une spécification similaire au service original. Dans le but de tirer avantage des services équivalents, nous introduisons la notion de ***Service Web Abstrait (SWA)***. Un SWA n'a pas de réalité fonctionnelle mais possède un document WSDL ; c'est une abstraction de plusieurs Services Web similaires. Le connecteur associé à un SWA doit convertir ce que nous appelons les requêtes abstraites en requêtes concrètes et vice-et-versa pour les réponses. Tous ces éléments vont être par la suite détaillés. L'objectif de nos travaux, ici, est de réaliser l'interface de chaque opération abstraite de manière à, d'une part, minimiser le nombre de paramètres d'entrée à fournir et donc permettre à l'utilisateur de faciliter son déploiement et, d'autre part, de maximiser le nombre d'informations communes produites par chaque service concret.

Pour créer de tels connecteurs, plusieurs définitions doivent être introduites:

- ***paramètre*** : un paramètre « p » est un couple (nom, type) noté nom : type.

  Exemple : keyword : string

- ***Instance d'ensemble de paramètres*** : une instance d'un paramètre est l'association d'une valeur à un nom tel que la valeur appartienne au type du paramètre. Soit P un ensemble de paramètres p,…, pn. Une instance d'un ensemble de paramètres est un ensemble d'associations nom-valeur noté $\{inst_1, \ldots, inst_n\}$ tel que $\forall\ i \in [1,n]$, $inst_i$ est une instance de p_i. Notons « inst(P) », l'ensemble d'instances de P.

  Exemple: inst(P)={ (keyword,'JB007') ; (language,'fr') ; (maxResults,'10') }

- ***Opération concrète*** : Une opération concrète « Op » est une fonction qui à chaque requête associe une réponse.

- ***Interface d'une opération*** : L'interface d'une opération représente un ensemble de paramètres. La description d'une opération est composée de deux interfaces :

  - Une interface d'entrée « InterfaceE(Op) » contenant les paramètres d'entrée,
  - Une interface de sortie « InterfaceS(Op) » contenant les paramètres de sortie.

- ***requête***: une requête « Req » est une instance de l'interfaceE(Op). C'est un ensemble d'instances des paramètres d'entrée de Op tel que pour chaque paramètre d'entrée de Op, il y ait exactement une instance.

- ***réponse*** : une réponse « Rep » est une instance de l'interfaceS(Op) générée par Op en fonction de « Req » tel que Rep = Op(Req).

- ***Opération abstraite*** : Une opération abstraite « Op_{abs} » est construite à partir d'un ensemble d'opérations concrètes Op_1 … Op_n appartenant à différents services. La description d'une opération abstraite possède également deux interfaces nommées respectivement « $InterfaceE_{abs}$ » pour l'interface abstraite d'entrée et « $InterfaceS_{abs}$ » pour l'interface abstraite de sortie.

- ***Fonction de mappage*** : une fonction de mappage est une fonction de conversion permettant de traduire une requête issue d'une opération abstraite en une requête d'une opération concrète et inversement pour les réponses. Nous noterons :
    - ***reqMap*** : une fonction de mappage permettant de traduire une requête abstraite en une requête concrète
    - ***repMap*** : une fonction de mappage permettant de traduire une réponse concrète en une réponse abstraite

Nous définirons donc un ***chemin d'équivalence*** comme suit :

> *$\forall Op_i \in [1,n]$, et $\forall Req_{abs}$ issue de $InterfaceE_{abs}(Op_{abs})$, il existe $reqMap_i$ et $repMap_i$ tels que :*
>
> *$repMap_i(Op_i(reqMap_i(Req_{abs})))$ est une réponse syntaxiquement valide pour Op_{abs}*

Figure 2-7: Définition de l'équivalence d'opération

Ces chemins d'équivalence permettent en fait d'interroger n'importe quel service concret à partir d'une requête abstraite. Ainsi, une paire triviale (« $interfaceE_{abs}$ », « $interfaceS_{abs}$ ») pour l'opération abstraite « Op_{abs} » pourrait donc être construite de la façon suivante :

- l'union de toutes les interfaces d'entrée
- l'intersection de toutes les interfaces de sortie

L'objectif de nos travaux ici est de créer une interface pour une opération abstraite de manière optimale, en réduisant au maximum l'effort demandé à l'utilisateur pour décrire la requête. Le but est donc de minimiser la taille de l'interface abstraite d'entrée « $interfaceE_{abs}$ » et de fournir le plus d'informations possibles en maximisant la taille de l'interface abstraite de sortie « $interfaceS_{abs}$ ».

Nos contraintes sont donc les suivantes :

Soit Ω l'ensemble des opérations concrètes équivalentes tel que $\Omega = \{ Op_1, \ldots, Op_n \}$,

$$InterfaceE_{abs} \subseteq \bigcup_{Op \in \Omega} InterfaceE(Op)$$

$$\bigcap_{Op \in \Omega} InterfaceS(Op) \subseteq InterfaceS_{abs}$$

La création d'une interface d'une opération abstraite s'effectue en deux étapes chacune introduisant les notions suivantes.

1. définition des relations d'équivalence (définies par l'utilisateur) et vérification de la cohérence des paramètres générés à partir de ces relations.
2. génération des interfaces abstraites : $InterfaceE_{abs}$ et $InterfaceS_{abs}$

2.3.3.1.1 Définition des relations d'équivalence

Une relation d'équivalence *f* est une relation sémantique entre deux instances d'ensemble de paramètres. Bien que syntaxiquement différents, deux paramètres peuvent représenter la même information sémantique. Ces relations d'équivalence sont établies par l'utilisateur lorsqu'il estime que la sémantique de $inst(P_1)$ est équivalente à celle de $inst(P_2)$ du point de vue d'une requête à Op_{abs}. Elle est définie manuellement par l'utilisateur et est supposée cohérente.

La Figure 2-8 permet de visualiser différentes relations d'équivalences sur un exemple. Les trois interfaces d'entrée concrètes qui y sont présentées, dans cet exemple, permettent respectivement de contacter un fournisseur d'achat en ligne d'ordinateur. Les trois mécanismes de recherche sont par contre sensiblement différents. Le service A sélectionne un ordinateur en fonction de paramètres qualitatifs tels que le mode de fonctionnement (ordinateur de jeux, de stockage, serveur de données, …etc), la gamme de prix, la taille et le type de l'écran. Le service B n'effectue sa recherche qu'en fonction de la marque (HP, Sony, Compaq, DELL, …etc.) et du modèle de l'ordinateur. Enfin, le service C effectue sa recherche en fonction des pièces de base constituant celui-ci.

En utilisant les relations d'équivalence définie par l'utilisateur, on peut ainsi construire respectivement les interfaces d'entrée et de sortie du Service Web Abstrait à partir de services concrets. Dans l'exemple de la Figure 2-8, seulement l'interface minimale du Service Web abstrait est représentée. Celle-ci peut être générée en fonction de la marque, du modèle et de l'écran de l'ordinateur. En effet, à partir de ces trois paramètres, on peut générer toutes les interfaces concrètes :

- la marque et le modèle de l'ordinateur peuvent être directement utilisés pour interroger le service B,
- les relations d'équivalence n°1, n°2 et n°4 peuvent être utilisées pour générer les paramètres du service A et
- la relation d'équivalence n°3 peut être employée pour générer les données d'entrée du service C.

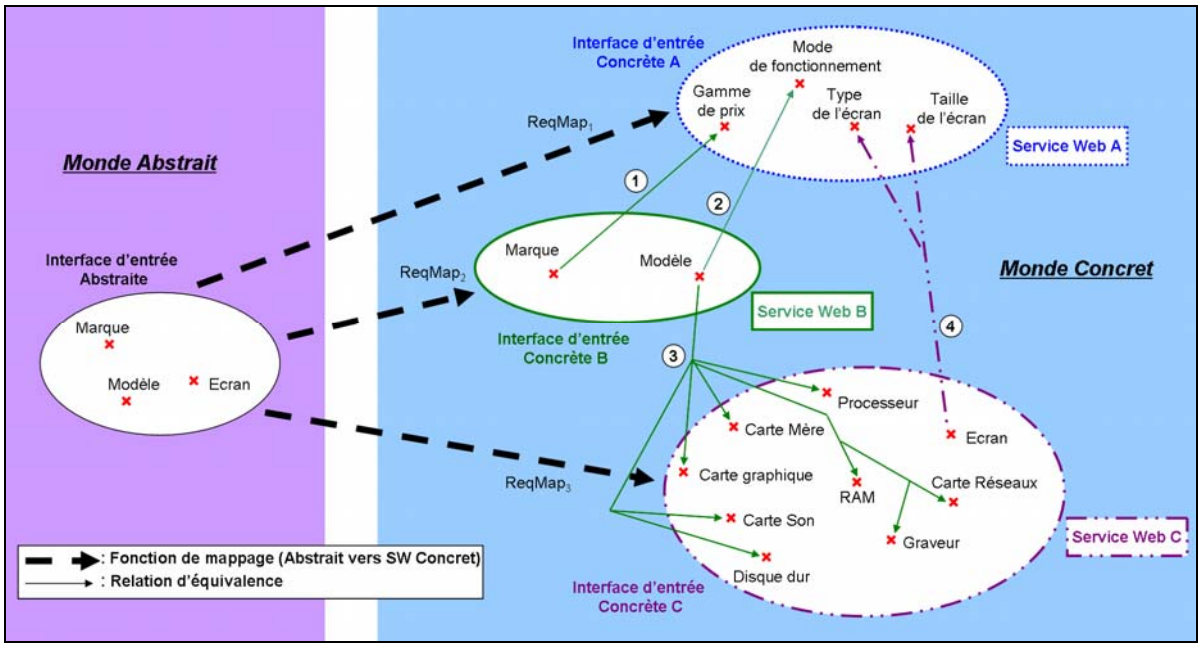

Figure 2-8: Notion de Service Web abstrait

Ces relations d'équivalence permettent de déduire ainsi trois types de paramètres:

- les paramètres dits « ***producteurs*** » qui peuvent générer des paramètres d'un nouvel ensemble. Par exemple, la marque du service B peut générer la gamme de prix du service A.

- les paramètres dits « ***induits*** » qui peuvent être déduits à partir de paramètres producteurs. La carte mère du service C peut être déduite ici en fonction du modèle du service B.
- les paramètres ni producteurs, ni induits sont dis « ***privés*** » : Ces paramètres sont propres et obligatoires pour accéder à une opération concrète spécifique. Il n'y a pas de relation d'équivalence pour convertir les instances d'un ensemble vers des instances d'un autre ensemble (par exemple, les paramètres d'authentification propres à chaque service). Ceux-ci ne sont pas représentés dans l'exemple de la Figure 2-8

Les relations d'équivalences étant fournies par l'utilisateur, certaines incohérences peuvent être détectées lors de la création des interfaces abstraites. Ces incohérences se produisent essentiellement lorsque l'on utilise une relation d'équivalence qui régénère un paramètre déjà obtenu par l'utilisation d'une relation antérieure. Pour assurer la cohérence globale des interfaces abstraites, il est important d'éviter les conflits lors de la génération des paramètres produits. Pour cela, lorsqu'une relation d'équivalence génère un ensemble de paramètres, on s'interdit d'utiliser d'autres équivalences sémantiques qui pourraient à nouveau générer un paramètre de cet ensemble. On obtient, au final, plusieurs ensembles de paramètres correspondants à des interfaces abstraites de services valides construits à partir d'un enchaînement différent de relations d'équivalence. L'utilisateur choisit ainsi celles qu'il souhaite utiliser pour réaliser son service abstrait.

2.3.3.1.2 Génération des interfaces abstraites : InterfaceE$_{abs}$ et InterfaceS$_{abs}$

Le but de ces travaux est de pouvoir automatiser la création de l'interfaceE$_{abs}$ et l'interfaceS$_{abs}$ à partir des relations d'équivalence fournies par l'utilisateur.

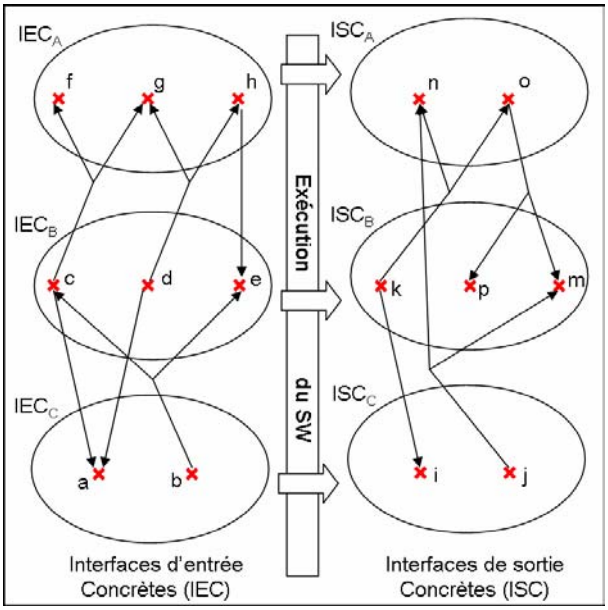

Figure 2-9: Résolution d'une interface abstraite

La Figure 2-9 représente respectivement trois interfaces d'entrée concrètes et les interfaces de sortie associées. L'ensemble des points et des relations d'équivalence de toutes les interfaces (entrées ou sorties) représente un hypergraphe. Dans cette modélisation, les relations d'équivalence constituent les hyper-arcs. L'objectif est de construire l'interface d'entrée minimale et l'interface de sortie maximale tout en garantissant la cohérence. Nous présentons, ici, un algorithme minimal permettant de résoudre ce problème.

a) génération de l'interface d'entrée : InterfaceE$_{abs}$

Pour générer l'interface d'entrée minimale, nous parcourons tous les hyper-arcs et nous ne conservons que les chemins qui n'entraînent pas d'incohérence. Nous ne récupérons ensuite que les chemins qui ont été construits à partir d'un minimum de paramètres. Ces ensembles de paramètres constituent des interfaces d'entrée possibles de l'opération abstraite. En guise de remarque, on peut constater que tous les paramètres privés sont toujours intégrés dans l'interface d'entrée abstraite car ceux-ci ne peuvent être générés par d'autres paramètres.

Dans l'exemple que nous donnons en Figure 2-9, l'algorithme produit deux interfaces qui peuvent être mises en œuvre avec un ensemble de paramètres distincts :

- Ensemble de paramètres [*f,d,b*] : *d* permet de générer les paramètres [*g,h,a*] et *b* génère les paramètres [*e,c*]. On obtient ainsi tous les paramètres qui nous permettent de contacter n'importe quelle opération.

- Ensemble de paramètres [*h,d,b*] : *b* permet de générer les paramètres [*e,c*] et *c* génère les paramètres [*f,g,a*]. On obtient également tous les paramètres qui nous permettent de contacter n'importe quelle opération.

b) génération de l'interface de sortie : InterfaceS$_{abs}$

Pour générer l'interface de sortie maximale, nous parcourons également tous les hyper-arcs en évitant les situations d'incohérence. Nous récupérons ensuite les chemins qui ont généré un maximum de paramètres. Ces ensembles de paramètres constituent les interfaces de sortie possibles de l'opération abstraite. On peut remarquer, ici, que les paramètres privés ne peuvent jamais être fournis dans l'interface de sortie abstraite.

Dans l'exemple donné en Figure 2-9, l'algorithme ne nous fournit qu'un seul résultat possible. Seul les paramètres *n* et *m* peuvent être intégrés dans l'interface de sortie. En effet, quel que soit le service Web utilisé pour générer une réponse, on peut, à partir de celle-ci fournir ces deux paramètres en sortie :

- Si le service A répond : *n* est présent et *m* peut être généré à partir du paramètre *o*.
- Si le service B répond : *m* est présent et *n* peut être généré à partir du paramètre *k*.
- Si le service C répond : *i* est utilisé pour générer les paramètres *n* et *m*.

Nous présentons, en annexe A.3, l'algorithme minimal réalisé en Prolog permettant de générer les différentes interfaces abstraites d'entrée et de sortie possibles à partir de relations d'équivalence définies par l'utilisateur.

L'annexe A.4 présente un contrat de service abstrait réalisé à partir de deux services de moteurs de recherches : Google et MSN. Ce document précise l'interface d'entrée et de sortie d'une opération abstraite nommée « *doAbstractSearch* ». Il contient également les points d'accès concrets à contacter ainsi que les scripts de mappage. Ces scripts contiennent les relations d'équivalence utilisées pour générer les requêtes concrètes et les réponses abstraites. Ils sont écrits en XSLT (*eXtended Stylesheet Language Transformations*) [85]. XSLT est un dialecte XML permettant de décrire des transformations à appliquer à des documents XML.

Une fois les répliques sélectionnées, elles sont enregistrées dans la plateforme IWSD au travers du service de gestion. Cet enregistrement s'effectue à l'aide du document WSDL correspondant à la réplique. L'utilisateur doit également insérer les scripts de mappage dans le cas où la réplique n'est pas identique.

A l'exécution, sur réception d'une requête, le connecteur crée une liste de tous les points d'accès disponibles. L'ordre des répliques correspond à l'ordre de leur enregistrement dans le service de gestion. C'est cette liste qui sera par la suite manipulée par les différents mécanismes de recouvrement.

2.3.3.2 Le type des opérations

Les Services Web sont par définition sans état (stateless). Chaque requête est traitée par un nouvel objet (par exemple, une servlet [86]) créée pour lui répondre. Cela ne signifie pas pour autant que la prestation ne comporte pas d'état. Les tâches qui produisent les unités de prestations comportent en réalité deux niveaux de gestion d'état, qui correspondent directement aux deux catégories listées ci-dessous :

- **Les prestations sans état (stateless):** la tâche correspondante ne produit aucune transition d'état. Rentrent dans cette catégorie les exemples cités de calcul intensif et de recherche sur des bases d'informations réparties et, en général, tout traitement dont le seul résultat est la restitution d'informations. La même occurrence de tâche peut être exécutée plusieurs fois de suite sans effets autres que des effets temporaires comme l'occupation et, éventuellement, le verrouillage des ressources de calcul, de mémoire, d'information. Lors de la répétition de la tâche, les informations livrées comme résultat de l'exécution peuvent rester invariantes, ce qui arrive lorsque l'on demande l'exécution d'un calcul avec les mêmes données et les mêmes paramètres. Les résultats de la tâche peuvent aussi varier au cours des répétitions, car si la tâche est sans état, l'application, elle, peut gérer des états qui évoluent (par exemple, la base de données de réservation des places d'avion) et l'interrogation est d'ailleurs effectuée pour obtenir l'état le plus « frais ».
- **Les prestations avec gestion d'état interne (stateful):** la tâche correspondante produit des transitions d'état directement gérées par les applications prestataires. En règle générale, l'accomplissement, plus d'une fois, d'une occurrence de la tâche produit des transitions d'état successives : l'état final n'est pas le même que celui qui est produit lorsque la tâche n'est exécutée qu'une seule fois (par exemple, lorsque l'on passe deux fois une écriture comptable qui correspond au retrait d'une somme d'argent d'un compte). Le propre de la tâche avec gestion d'état interne est qu'elle peut toujours, indépendamment de la complexité des traitements requis et si l'on a pris les précautions nécessaires, être défaite. C'est-à-dire que l'on peut toujours revenir à l'état précédent. Cela implique en général de défaire non seulement l'occurrence de la tâche en question mais aussi toutes celles qui ont suivi et qui ont provoqué depuis des transitions d'état.

Parmi les prestations avec gestion d'un état interne, on peut distinguer, les prestations qui modifient l'environnement externe. La tâche correspondant à cette prestation produit des actions qui changent l'état de l'environnement, en cohérence avec une transition d'état interne. Ces prestations utilisent parfois des effets de bord pour changer l'état de l'environnement. Lors de l'impression d'un billet avec réservation, le document produit n'est pas seulement un support d'information, il constitue également un titre qui donne certains droits et obligations à son possesseur. Par ailleurs, le changement de l'état de l'environnement est indépendant de l'action physique d'impression du billet (le « billet électronique », de plus en plus répandu, en est la preuve s'il en faut) : dans ce cas, le changement de l'état interne du système représente directement le changement de l'état de l'environnement. Le propre de l'état de l'environnement, du point de vue de l'application prestataire, est que, contrairement

aux états internes, il est irréversible. Les états de l'environnement ne peuvent pas être défaits par l'application prestataire, mais ils peuvent parfois être compensés, ce qui signifie que l'application peut prendre des dispositions qui amènent le système global (les applications et l'environnement) à un état que l'on peut considérer proche de l'état auquel on veut revenir (les deux états ne sont jamais identiques). Par exemple, il est possible d'effectuer la réservation d'une place, puis d'annuler celle-ci. La similitude s'arrête là : dans le laps de temps écoulé entre la réservation et l'annulation, l'avion s'est rempli, et certains voyageurs, qui veulent être certains de pouvoir voler, ne se sont pas mis en liste d'attente et se sont tournés vers une compagnie concurrente.

Ainsi, le mécanisme de recouvrement mis en place par l'utilisateur doit tenir compte du type de l'opération sur laquelle il s'appuie. Suivant l'opération ciblée, sa complexité peut varier considérablement.

2.3.3.3 Les stratégies de recouvrement

Les stratégies de recouvrement font parties des points prépondérants de cette plate-forme. Elles sont définies dans la Figure 2-10. Pour chacune d'entre elles, cette figure présente les hypothèses faites sur les répliques de services utilisées ainsi que les types de fautes qui peuvent être détectées par les mécanismes mis en jeu.

Les stratégies de **réplication passive** impliquent d'envoyer la requête à une seule réplique qui la traite. En cas d'erreurs détectées (par les assertions des post-traitements ou la capture d'exceptions), une réplique disponible est utilisée pour exécuter la requête. Au delà de l'exécution des assertions, le connecteur fournit la détection d'erreur et le routage (primaire inaccessible en raison du crash du noeud, du crash du service ou du gel du service). Ce mécanisme s'apparente aux blocs de recouvrement (Recovery Blocks) proposés par [9, 87]. Si les résultats fournis par la réplique primaire ne satisfont pas le test d'acceptation (ici, les Post-Processing), la réplique secondaire est exécutée, et ainsi de suite jusqu'à la satisfaction du test d'acceptation ou l'épuisement des répliques disponibles, auquel cas le bloc de recouvrement est globalement défaillant.

Concernant les stratégies de **réplication active**, le connecteur envoie en multicast la réplique aux N répliques du Service Web une fois les assertions de pré-traitement effectuées. Deux approches sont possibles:

- ***Sans vote*** : le connecteur reçoit les réponses des N répliques, et envoie au client la première réponse qui passe les assertions de post-traitement. Ce mécanisme tolère le crash du noeud, le crash de service ou le gel de service. Dans la pratique, la réplication active a été utilisée dans nos expériences avec toutes les répliques ***identiques*** existantes du Service Web d'Amazon.
- ***Avec vote*** : le mécanisme de recouvrement avec vote (majoritaire) peut tolérer des fautes de valeur (corruption de données). Le connecteur reçoit les réponses fournies par les *2f+1* répliques du service, *f* fautes peuvent être tolérées dans ce cas. Une bibliothèque des variantes d'algorithme de vote (au niveau du bit, moyenne, médiane, etc.) peut être fournie à l'utilisateur pour la configuration de vote dans le connecteur.

Ces stratégies de recouvrement peuvent être adaptées aux besoins du client par l'utilisateur, c'est-à-dire, elles peuvent être paramétrées ou des variantes peuvent être dérivées de celles existantes (par exemple inspirées par des approches comme les *Recovery Blocks*, *N-Version Programming, N-Self-Checking Programming*). Clairement, tous ont une liste des répliques comme premier paramètre. Les autres paramètres concernent des valeurs de temporisation,

des variantes de point de reprise, des fonctions de décision, etc. Des stratégies plus avancées peuvent également être créées pour aborder d'autres aspects non fonctionnels, y compris des stratégies de recouvrement dépendantes de l'application.

Stratégie de recouvrement	Hypothèse		Fautes détectées
	Opérations sans état	Opérations avec état	
Réplication Passive	- Services à silence sur défaillances	- Services à silence sur défaillances - Restauration possible de l'état sur une réplique de secours	Fautes temporelles
Réplication Active sans vote	- Services à silence sur défaillances	- Services à silence sur défaillances - Déterminisme des répliques - Ordonnancement total des messages - Connexion fiable	
Réplication Active avec vote	- Services à défaillances en valeur	- Services à défaillances en valeur - Déterminisme des répliques - Ordonnancement total des messages - Connexion fiable	- Fautes temporelles - Fautes en valeur

Figure 2-10: Les stratégies de recouvrement

Parmi les hypothèses présentées dans la Figure 2-10, certaines, comme la gestion de l'état et l'ordonnancement des messages, peuvent être assurées par les connecteurs avec toutefois une collaboration du prestataire. Ces différentes problématiques ainsi que des détails approfondis sur les modes de recouvrement et la façon de les spécifier dans un programme DeWeL par l'utilisateur sont présentés dans le chapitre 4.

2.3.3.4 Les modèles d'exécution du connecteur

Le modèle d'exécution détermine le flot de séquences d'instructions que devra effectuer le connecteur pendant son cycle de vie. Ce flot de séquences est conditionné par la stratégie de recouvrement mise en œuvre. Cette stratégie est définie par l'utilisateur aux travers de fonctions spécialisées liées au langage DeWeL. Ces mécanismes peuvent être sélectionnés pour chaque opération en les spécifiant dans la section « RecoveryStrategy ».

Nous ne présentons dans ce chapitre que les modèles d'exécution suivant :

- Le modèle d'exécution de la réplication passive pour les opérations sans état (cf. Figure 2-11).
- Le modèle d'exécution de la réplication active pour les opérations sans état (cf. Figure 2-12).

La prise en compte de l'état ou la mise en place d'un vote majoritaire dans la réplication active n'étant que des variantes de ces deux modèles de base, ceux-ci seront détaillés dans le chapitre 4.

La Figure 2-11 représente le modèle d'exécution appliqué aux connecteurs pour mettre en œuvre la réplication passive sur des opérations sans état. Le cas nominal est identique au modèle linéaire présenté précédemment. Par contre, lors d'une erreur survenue après avoir contacté la réplique primaire (message invalidé par les post-traitements de l'utilisateur, ou levée d'une exception de service ou de communication), une nouvelle réplique est sélectionnée pour traiter la requête. Ce système boucle jusqu'à ce que la réception d'une

requête soit validée par le post-traitement ou jusqu'à épuisement de toutes les répliques. Dans ce dernier cas, une exception SOAP est retournée au client.

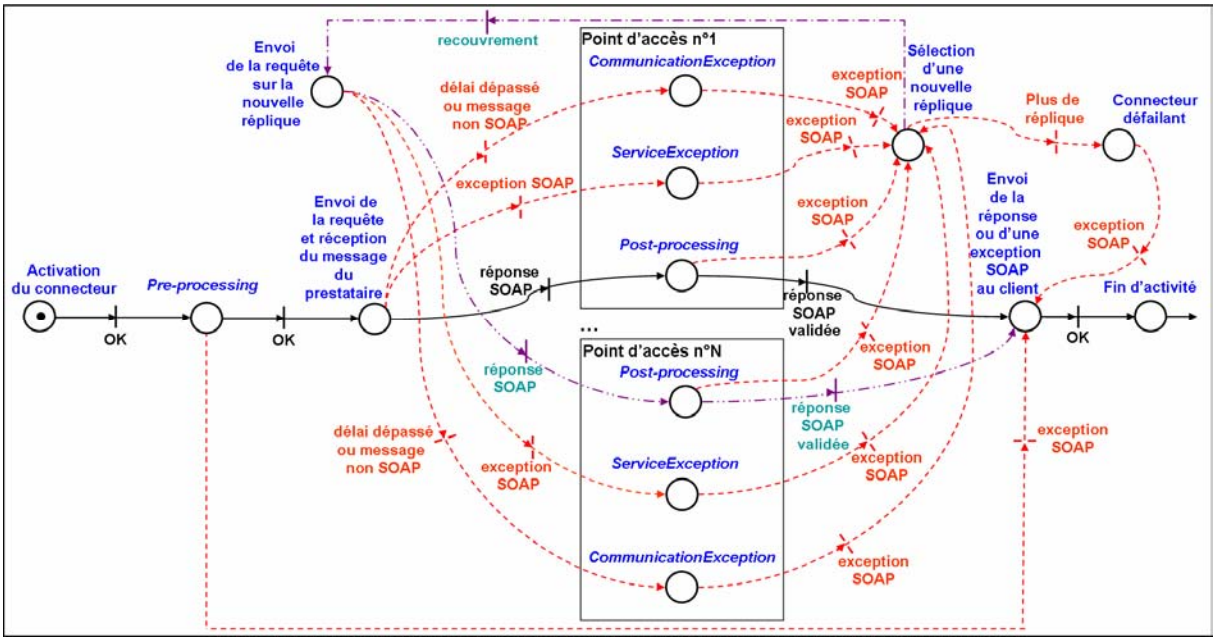

Figure 2-11: Le modèle d'exécution de la réplication passive sans état

Dans le modèle d'exécution de la réplication active pour les opérations sans état présenté dans la Figure 2-12, la requête est envoyée à toutes les répliques simultanément. Lorsque le connecteur reçoit une réponse validée par le post-traitement, celle-ci est envoyée au client. Les autres réponses valides sont ignorées. Si toutes les répliques produisent une erreur, le connecteur est défaillant, une exception SOAP est retournée au client. Ce modèle s'applique essentiellement sur des services sans état, qui ne nécessitent pas un ordonnancement total des messages pour garantir la cohérence des répliques. Pour les autres modèles basés sur lui, ces problématiques doivent, bien évidemment, être prises en compte.

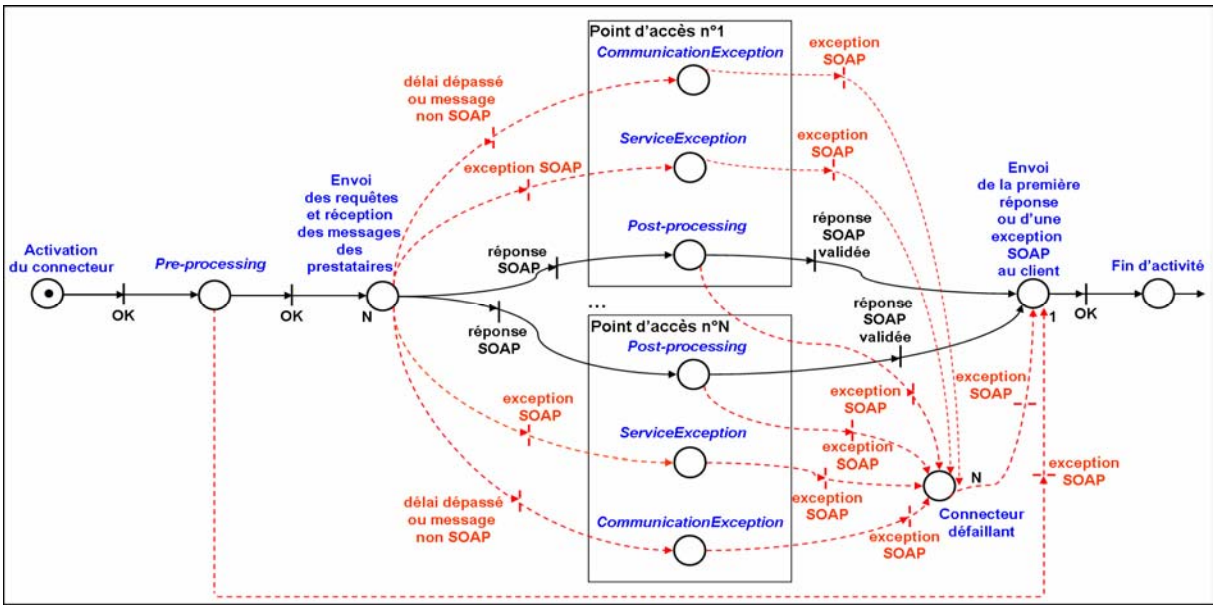

Figure 2-12: Le modèle d'exécution de la réplication active sans état

Pour des raisons de clarté, les figures présentées ne prennent pas en compte certaines caractéristiques comme la création et la mise à jour de la liste des répliques, l'envoie de messages au moniteur de surveillance ou la levée d'erreur interne (ex : impossible d'activer un connecteur à cause d'un manque de mémoire du support d'exécution). Dans le cas où la requête reçue par le connecteur est une requête abstraite, ces modèles ne prennent pas non plus en compte les traductions de ces requêtes en requêtes concrètes et vice-et-versa pour les réponses.

2.4 Le support d'exécution

Une fois spécifié par l'utilisateur et généré par le compilateur DeWeL, le connecteur peut être déployé sur le support d'exécution.

2.4.1 Le serveur d'exécution

Le serveur d'exécution est le support exécutif des connecteurs spécifiques de tolérance aux fautes. C'est une machine virtuelle constituée de plusieurs services capables de charger et de contrôler l'activité des connecteurs. Dans cette section, sont détaillés différents aspects du serveur d'exécution à savoir : son installation, son mode de fonctionnement et les différents composants qui le constituent.

Pour améliorer la sûreté de fonctionnement des architectures orientées services, la plate-forme doit garantir trois propriétés essentielles :

- Les connecteurs créés doivent permettre de rajouter des mécanismes non-fonctionnels de sûreté de fonctionnement spécifiques à un contexte d'application donné.
- La génération de connecteurs doit être exempte de fautes.
- Le serveur d'exécution doit être, lui-même, tolérant aux fautes.

Les deux premières propriétés concernent l'expressivité, le processus de compilation des programmes DeWeL ainsi que le développement des mécanismes de tolérance aux fautes. Elles seront détaillées dans le chapitre 3. La troisième sera analysée dans cette section.

2.4.1.1 Dimensionnement et Administration

Le serveur d'exécution s'installe de façon traditionnelle de la même manière que, par exemple, le serveur apache. Un script de configuration est fourni à l'administrateur permettant de dimensionner son serveur (nombre maximum de connexion possible), le port d'écoute et de sélectionner le mode de recouvrement désiré. Le moniteur de surveillance associé au serveur d'exécution doit également être précisé dans ce script. Chaque serveur d'exécution doit être connu du serveur de gestion. Ceci permet aux utilisateurs d'avoir une vue de tous les serveurs d'exécution existants et de décider sur lequel leurs connecteurs seront exécutés par défaut.

2.4.1.2 Les composants fonctionnels du Serveur d'exécution

Le serveur d'exécution fournit trois services présentés sur la Figure 2-13: le **Service d'exécution,** le **Gestionnaire des connecteurs actifs** et le **Chien de garde**. Ces composants sont responsables de la gestion et du contrôle des connecteurs et du serveur d'exécution.

Le **Chien de garde** est un composant autonome qui a pour seule fonction d'envoyer des messages de type « I'm alive » au moniteur de surveillance et à son serveur de secours. Ce module permet de détecter si le serveur d'exécution est en arrêt ou non.

Le **Gestionnaire des connecteurs actifs** est le mécanisme de stockage interne des connecteurs propres à un serveur d'exécution. Il contient les connecteurs déjà utilisés. Ces connecteurs sont téléchargés à partir du serveur de gestion. Une défaillance de ce module empêcherait le serveur d'exécution de charger le connecteur. Il serait dans l'impossibilité de réaliser les actions de pré-et-post traitements demandées par le client et lui retournerait automatiquement une exception.

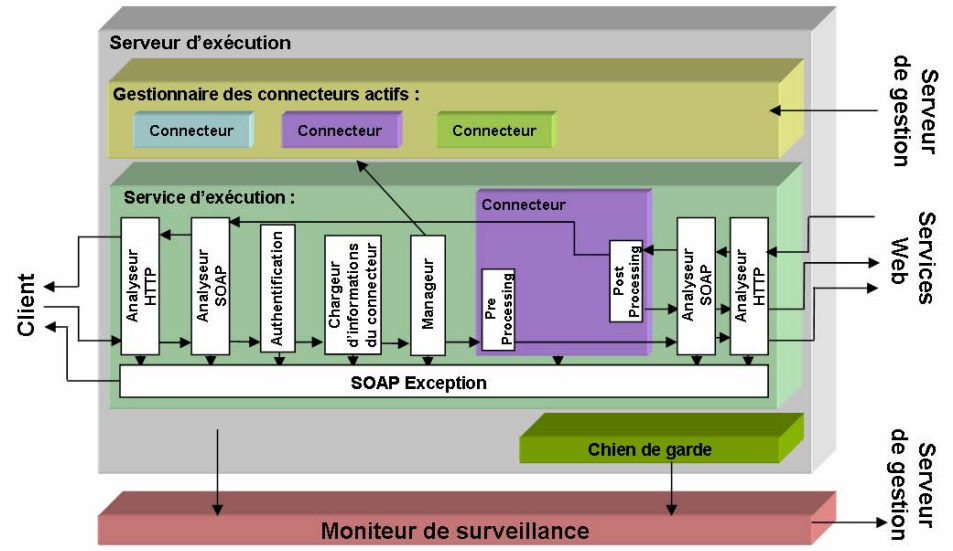

Figure 2-13: Le Serveur d'exécution

Le **Service d'exécution** est le cœur de notre système. Il est activé quand le Serveur d'exécution accepte une nouvelle connexion en provenance d'un client. A ce moment là, une nouvelle session est créée, plusieurs fonctions et services (cf. Figure 2-13) y sont activés :

1. Les **analyseurs HTTP et SOAP** vérifient la syntaxe des messages et les désérialisent. Une exception SOAP est automatiquement générée dans le cas d'une réception d'un message mal-formaté.
2. Le module d'**authentification** a pour rôle de récupérer l'identité de l'utilisateur. Cette identification peut être faite à partir de l'en-tête HTTP et/ou de l'en-tête SOAP. Si le message présente une identification au niveau HTTP et SOAP, seul le message d'authentification SOAP est pris en compte. Si aucune identification n'a pu être récupérée cela signifie que le connecteur devra avoir le statut public.
3. Le rôle du **chargeur d'informations du connecteur** est de récupérer toutes les informations relatives au connecteur désiré auprès du serveur de gestion et notamment des informations de contrôle d'accès. Le contrôle d'accès définit le mode d'utilisation du connecteur (statut « public » ou « privé »). Un connecteur public peut être utilisé par n'importe quel client, un connecteur privé nécessite une authentification avant de pouvoir être chargé et exécuté. Ce module charge donc, en interrogeant le serveur de gestion, toutes les informations relatives à ce connecteur.
4. Le **manageur** vérifie si la version du connecteur présente dans le gestionnaire du serveur d'exécution correspond à la dernière version mise à jour et enregistrée sur le

serveur de gestion. Si ce n'est pas le cas, il demande au gestionnaire de la télécharger. Dans le cas où il serait impossible de télécharger la dernière version du connecteur, une exception est retournée au client. La connexion entre le gestionnaire du serveur d'exécution et le serveur de gestion étant totalement privée, celle-ci peut être effectuée au travers d'un tunnel SSL pour garantir la sécurité et l'intégrité des messages transférés.

5. Le **connecteur** effectue les pré et post traitements sur le Service Web interrogé et active les mécanismes de sûreté de fonctionnement définis par l'utilisateur.

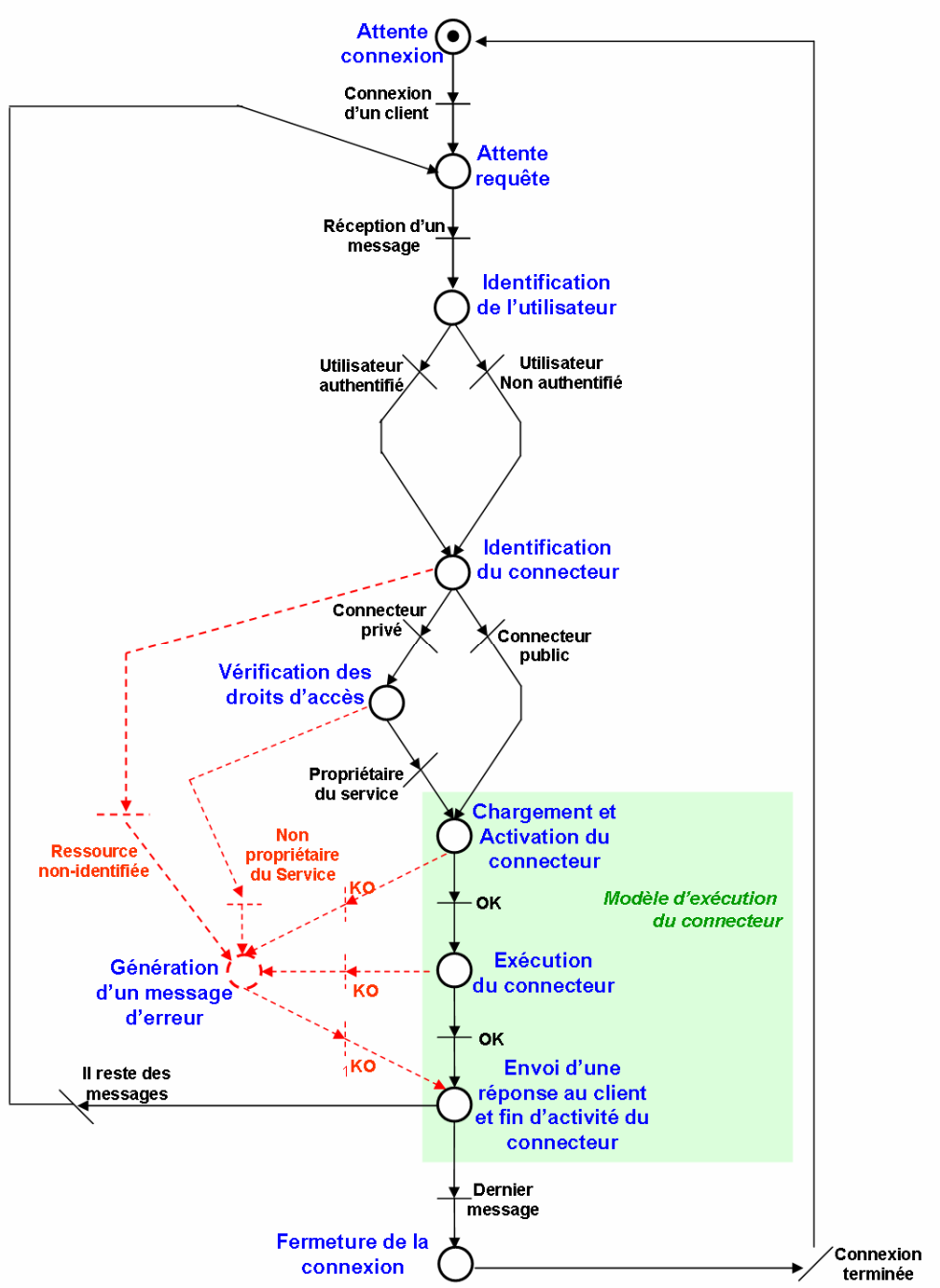

Figure 2-14 : Déroulement d'une connexion cliente sur le serveur d'exécution

Cette phase d'exécution est essentielle. Son déroulement est présenté sur la Figure 2-12. On peut remarquer ici que les opérations réalisées pour un ou N client(s) sont les mêmes, exécutées en parallèles. Définir le déroulement d'une connexion cliente permet donc de définir le déroulement de toutes les connexions en cours.

La Figure 2-14 présente le déroulement d'une connexion client s'appuyant sur le protocole HTTP. Celle-ci reprend partiellement les explications définies plus haut avec notamment l'identification de l'utilisateur et du connecteur associé à la requête reçue. Ce schéma permet d'analyser à quel moment les modèles d'exécution présentés précédemment sont insérés dans le modèle global. En effet, suivant la stratégie de recouvrement définie par l'utilisateur, l'état dénommé « exécution du connecteur » est remplacé par le modèle d'exécution correspondant. Les traitements en pointillés rouges représentent le fonctionnement du service d'exécution en mode dégradé. L'envoi de messages au moniteur de surveillance n'est pas représenté ici.

Bien que le protocole HTTP soit sans état, les requêtes étant indépendantes les unes des autres à moins d'utiliser des cookies [88], il permet d'envoyer plusieurs requêtes sur une même connexion. Cette caractéristique est représentée sur la Figure 2-14 et est, bien sur, dépendante du protocole de transport. Le fait d'utiliser un autre protocole de transport peut modifier considérablement le déroulement d'une connexion. Suivant le port d'écoute en attente d'une connexion (80 pour le port HTTP, 25 pour le port SMTP, …etc), ce déroulement peut changer du tout au tout.

2.4.1.3 Un serveur tolérant aux fautes

Pour garantir la sûreté de fonctionnement des connecteurs, il faut bien entendu rendre le support d'exécution sûr de fonctionnement. Bien que cette problématique ne soit pas réellement au cœur des travaux de cette thèse, différentes recherches ont été effectuées sur ce thème [56, 60, 62] et nous allons énoncer rapidement une solution inspirée des travaux de Coral présentés dans le chapitre 1.

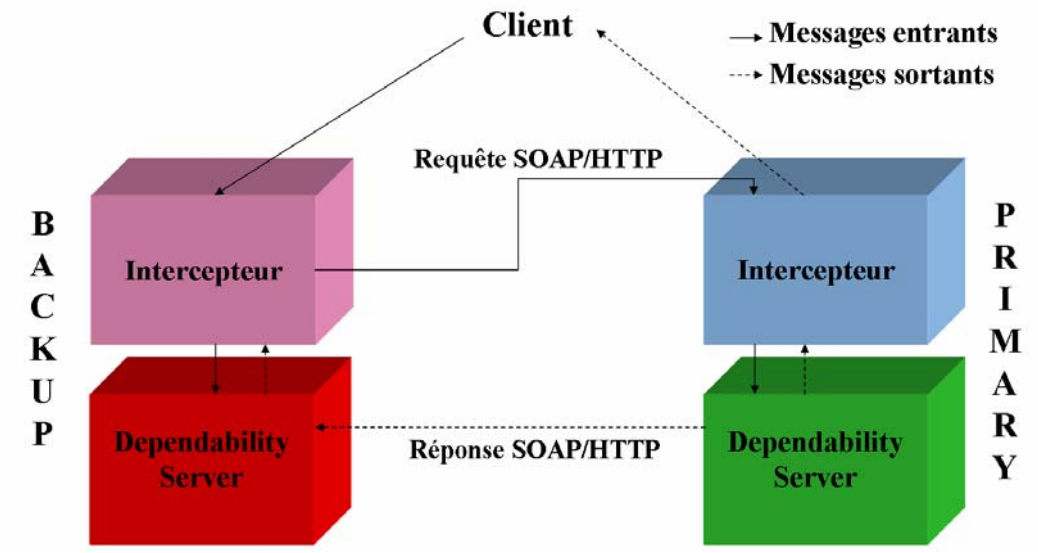

Figure 2-15: Le mode duplex du Serveur d'exécution

L'implémentation du serveur d'exécution en mode duplex, nécessite l'ajout d'intercepteurs (cf. Figure 2-15). Ceux-ci peuvent directement être implémentés soit au niveau TCP du système d'exploitation sous jacent, soit à un niveau supérieur ou par un composant externe.

L'idée de base de cette architecture est d'utiliser ces intercepteurs pour s'assurer que le serveur de secours, le backup, dispose d'une copie de chaque requête envoyée par le client avant de l'envoyer au primaire. Sur réception des requêtes, le serveur de secours envoie une copie de celles-ci au serveur primaire. Les adresses sources des requêtes conservent l'adresse du client, de sorte que les paquets apparaissent au serveur primaire comme étant des paquets directement envoyés par le client. Ainsi, si le serveur primaire défaille avant de générer la réponse, le serveur de secours peut exécuter la copie de la requête, générer une réponse et l'envoyer au client. Dans l'état nominal, la réponse est normalement générée par le primaire, une copie complète est envoyée au serveur de secours. Une fois, celle-ci reçue par le serveur de secours, le primaire envoie la réponse au client. Si le primaire défaille avant la transmission de la réponse au client, le backup peut envoyer sa copie de la réponse au client. Si le primaire défaille pendant l'envoie de la réponse au client, les intercepteurs sont utilisés pour s'assurer que la partie non-envoyée de la réponse sera envoyée par le backup.

Bien sûr, la faisabilité de cette architecture nécessite de faire l'hypothèse que le primaire et le backup sont à arrêt sur défaillance. La défaillance de l'un des deux serveurs est détectée par des messages périodiques de type « I'm alive » échangés entre le primaire et le backup grâce à leur chien de garde respectif.

2.4.2 Le moniteur de surveillance

Si le Service d'exécution est le cœur de notre architecture, le **moniteur de surveillance** en est, quant à lui, le poumon. Sa mission consiste à surveiller le système. Pour cela, il recueille en permanence, et de manière passive, tous les signaux en provenance du serveur d'exécution. Le moniteur de surveillance est en charge de collecter toutes les informations de défaillances et les rapports d'erreurs de chaque composant du Service d'exécution et du gestionnaire des répertoires actifs pour évaluer le niveau courant de sûreté de fonctionnement du serveur d'exécution. Le moniteur de surveillance est aussi en charge de superviser l'état à l'exécution des connecteurs. Il collecte toutes les erreurs détectées par les pré-et-post traitements qui sont analysés. Il permet ainsi de fournir un diagnostic précis du comportement du système et de garantir son caractère opérationnel vis-à-vis des considérations de sûreté requises. Cette collecte d'informations est transmise périodiquement au serveur de gestion au moyen d'une connexion sûre. Une fois enregistré, l'utilisateur peut ainsi consulter ces informations soit au moyen du serveur de gestion, soit par l'intermédiaire d'une requête SOAP envoyée directement au connecteur associé.

La gestion des erreurs est nécessaire pour une application visant à un apport de sûreté de fonctionnement. Chaque composant de notre plate-forme a été conçu à partir d'un ensemble de classes d'exceptions hiérarchisées créées dans le but de faire remonter les erreurs au niveau décisionnel de l'application. Le concept de « *piggy backing* » consiste à munir un outil existant de nouvelles fonctionnalités embarquées. Toutes sortes d'exceptions sont d'ores et déjà générées sur détection d'erreurs, permettant de signaler les états de défaillance du système et de se prémunir contre eux. L'idée est d'utiliser le principe de « *piggy backing* » sur les objets d'exception, en leur ajoutant la capacité de communiquer l'erreur qu'ils représentent à un « serveur d'erreurs », le moniteur de surveillance. Ces classes sont transformées en capteurs d'erreurs de l'instance du serveur en cours.

2.5 Le serveur de gestion

Le serveur de gestion est le module permettant d'enregistrer, de conserver et de consulter les informations nécessaires et relatives aux différents administrateurs, utilisateurs et connecteurs. Il est donc composé de deux modules capables:

- De faire intérargir ces acteurs avec le système : les ***Services de Gestion.***
- De stocker des informations persistantes indispensables au fonctionnement du système : le ***Service de stockage persistant.***

Les **services de gestion** constituent l'interface entre les différents acteurs (administrateurs, utilisateurs et connecteurs) et le système. Ces services sont au nombre de trois :

1. Le ***service de gestion des administrateurs*** permet à un acteur de s'enregistrer en tant qu'administrateur. Une fois enregistré, l'administrateur peut télécharger et installer la dernière version du serveur d'exécution et du moniteur de surveillance sur des nœuds appropriés.
2. Le ***service de gestion des utilisateurs*** permet à un utilisateur de créer, de supprimer ou de mettre à jour les informations personnelles de son compte. Cette phase est indispensable avant de pouvoir créer un connecteur.
3. Le ***service de gestion des connecteurs*** n'est accessible que par un utilisateur enregistré dans le système. Il permet à celui-ci de créer et d'installer un connecteur. Le service de gestion des connecteurs possède également deux sous-services indispensables à la création et à l'utilisation des connecteurs :
    - Le **service de génération d'interface des connecteurs** produit, à partir du programme DeWeL, le document WSDL (le contrat de service) correspondant au connecteur créé. Ce document est indispensable pour les clients qui souhaitent utiliser pleinement toutes les possibilités du connecteur. A l'aide de celui-ci, il peut générer le stub correspondant lui permettant d'accéder au connecteur associé.
    - Le **service de gestion des répliques** permet à l'utilisateur d'enregistrer les points d'accès répliqués associés à plusieurs instances du Service Web original. Ce service est indispensable pour pouvoir mettre en œuvre des mécanismes de recouvrement appropriés.

Le ***Service de stockage persistant*** conserve toutes les informations récupérées au travers des différents services de gestion et nécessaires au fonctionnement de la plate-forme. Ce service est implémenté à l'aide d'une **base de données** qui contient toutes les informations sur les administrateurs, utilisateurs du système ainsi que sur les points d'accès de tous les Services Web et répliques utilisées. De plus, Le **répertoire de sauvegarde des connecteurs** est un disque responsable du stockage des connecteurs créés par chaque utilisateur. Ce composant est un module critique de la plate-forme puisqu'il conserve des informations vitales au bon fonctionnement de la plupart des mécanismes. Chaque module qui le compose doit donc être redondant pour assurer une certaine disponibilité de service. Ceci peut être fait au travers de technique traditionnelle comme les disques miroirs par exemple.

2.6 Mise en place d'un connecteur dans une application

Lors du développement d'une architecture orientée service, le connecteur spécifique de tolérance aux fautes peut être introduit de deux façons différentes :

- Soit les composants de type connecteur sont prévus dès la spécification de l'architecture orientée service afin d'adapter la sûreté de chaque Service Web utilisé.
- Soit ils sont insérés dans une application déjà développée, lorsque la sûreté de fonctionnement d'un ou plusieurs Services Web ne satisfait plus les besoins du client.

Dans la première solution, les stubs clients sont générés à partir des documents WSDL des connecteurs. Chaque connecteur étant créé spécifiquement pour chaque Service Web en fonction du contexte et de la criticité de l'application, cette solution permet de construire facilement une zone adaptable de confinement d'erreur.

Dans la seconde solution, les connecteurs sont créés à posteriori, les stubs clients étant déjà créés pour chaque Service Web, trois possibilités se présentent à nous :

- Soit le client régénère le stub correspondant au contrat du connecteur. Celui-ci permet d'accéder aux fonctionnalités avancées comme la gestion de session ou la récupération de rapport de diagnostic sur le connecteur et les points d'accès utilisés. Cette solution l'oblige cependant à réimplémenter et tester son application.
- Soit le client modifie simplement l'url du point d'accès d'origine du Service Web pour le substituer par le point d'accès du connecteur créé. Ce cas nécessite de posséder le code source du stub client pour pouvoir le recompiler. Les fonctionnalités avancées du connecteur sont alors inatteignables. Les connecteurs utilisés dans ce cas sont donc forcément « publics » puisque aucune authentification n'est insérée dans le message.
- Soit il demande à l'administrateur de son système d'installer un module d'interception, le ***service d'écoute***, capable de router les requêtes de type Service Web vers un serveur d'exécution.

Dernier élément de cette architecture, le ***service d'écoute*** est le seul élément installé sur la machine cliente. C'est un composant qui capture les requêtes SOAP/HTTP et les redirige vers le serveur d'exécution sélectionné. Les réponses SOAP/HTTP délivrées par le service ciblé par le biais du serveur d'exécution sont retournées au client en passant par le service d'écoute.

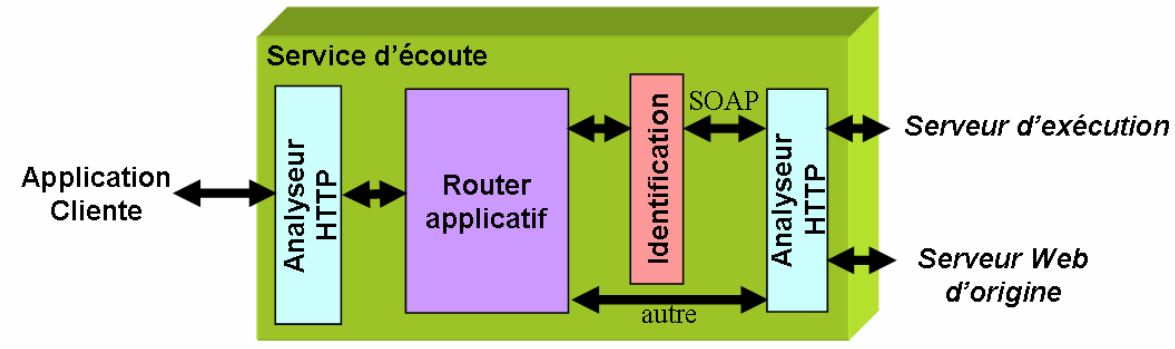

Figure 2-16: Le Service d'écoute

Quand le service d'écoute reçoit une requête, plusieurs étapes majeures sont effectuées :

1. Premièrement, l'**analyseur HTTP** examine la requête et récupère son en-tête pour l'envoyer au routeur applicatif.

2. Le **routeur applicatif** est le composant décisionnel. Celui-ci a deux fonctions :
    - Il analyse l'en-tête (le champ content-type de la requête HTTP) pour vérifier si le contenu de la requête est de type SOAP. Dans l'affirmative, elle est transmise au module d'identification. Dans le cas contraire, elle est transmise au serveur d'origine sans modification.
    - Lorsque la requête est de type SOAP, le routeur applicatif modifie l'url contenue dans l'en-tête HTTP pour que celle-ci soit transmise par la suite au connecteur et au serveur d'exécution désirés. Une table de traduction entre les adresses originales et les connecteurs est alors utilisée.
3. Le module d'**identification** permet d'insérer dans l'en-tête HTTP les informations qui permettront au serveur d'exécution qui recevra cette requête d'identifier l'émetteur correspondant comme un utilisateur.

Le rôle du service d'écoute est essentiel pour les applications existantes. Il peut être conçu comme un routeur derrière un pare-feu modifié qui filtrerait les requêtes HTTP clientes pour les router sur ce module. Il peut donc être inséré au niveau système sans modifier l'application cliente source. Cette architecture est donc non-intrusive pour les applications clientes déjà existantes. Situé côté client, ce module doit être portable sur n'importe quel système d'exploitation.

2.7 Récapitulatif

Placé entre les clients et les prestataires, les connecteurs jouent un rôle essentiel dans la communication entre ces deux acteurs. Ils insèrent une zone de confinement d'erreur mais également de confiance, dans laquelle ils peuvent se mettre d'accord sur les formats des messages qu'ils attendent, les différentes actions de tolérance aux fautes à réaliser ainsi que le mécanisme de recouvrement à mettre en place sur détection d'erreur.

La plateforme IWSD est le support d'enregistrement et d'exécution de ces connecteurs. Elle est constituée de plusieurs composants chacun ayant un rôle bien défini.

Le serveur d'exécution est le cœur de notre architecture. C'est lui qui exécute les connecteurs de chaque utilisateur c'est-à-dire, les actions de pré-et-post traitements. Les principaux atouts de cette plate-forme résident dans le fait qu'elle est non-intrusive et que les connecteurs sont définis par l'utilisateur. Cette dernière caractéristique est très intéressante puisqu'elle permet à un non-spécialiste de décrire facilement des mécanismes de tolérance aux fautes complexes, au travers du langage DeWeL, et de se prémunir de Services Web dans lequel il aurait une confiance limitée. Ce langage, détaillé dans le chapitre suivant, permet de spécifier les mécanismes non-fonctionnels associés au connecteur. Par ailleurs, les mécanismes de recouvrement mis en jeux doivent tenir compte d'un certains nombres d'hypothèses et du modèle de défaillances du service ciblé. En fonction de ces caractéristiques, le rôle des connecteurs ainsi que la collaboration des clients et des prestataires peut fortement varier. Ces actions de tolérance aux fautes ainsi que les propriétés de l'interface du connecteur (son contrat WSDL) seront détaillées dans le chapitre 4.

Le moniteur de surveillance permet, quant à lui, de surveiller et d'analyser l'état du système et des connecteurs. Les décisions suite à cette analyse incombent aux clients qui peuvent à tout moment basculer sur un autre serveur d'exécution ou connecteur, ou changer de prestataire.

Le serveur de gestion est composé de services réalisant, entre autre, la consultation et la gestion des administrateurs, des utilisateurs et des connecteurs. Il peut être considéré comme

un annuaire UDDI de connecteurs. Toutes ces fonctionnalités peuvent être implémentées sous forme de Services Web.

Bien qu'attractive, cette plate-forme présente tout de même un inconvénient, elle rajoute un intermédiaire dans le réseau, ce qui va diminuer les performances. La tolérance aux fautes à de toute façon un prix ! Notons en outre que les performances strictement temporelles ne sont pas, à la base, un argument commercial dans l'utilisation des Services Web sur Internet. En effet, basés sur le format XML, les Services Web sont voués à échanger de gros formats de données contrairement à leurs concurrents du monde objet (Corba ou DCOM) qui utilisent un protocole de transport privé au format binaire [89]. Nous tenterons de montrer dans le chapitre 5 que le surcoût temporel induit par cette plate-forme est en fait négligeable.

Une des caractéristiques importantes introduite par cette plate-forme est la notion de Service Web Abstrait. Ces services n'ont pas de réalité fonctionnelle mais permettent de contacter différents services sur le Net et ainsi de tirer profit de la redondance des ressources inhérentes au Web. Cette plate-forme offre, en fait, de nouvelles opportunités en ce qui concerne la disponibilité et la fiabilité des services.

3 DeWeL : Un langage dédié pour la sûreté de fonctionnement des Services Web

Dans ce chapitre, nous nous concentrons sur le développement du langage dédié DeWeL (DEpendable WEb service Language) permettant de décrire les mécanismes de tolérance aux fautes. Ce DSL (Domain Specific Language) permet de réaliser des connecteurs robustes pour la tolérance aux fautes des Services Web.

3.1 Introduction

DeWeL [90, 91] est un langage dédié contenant les abstractions et notations spécifiques permettant de configurer de façon **sûre** des propriétés à vérifier sur les opérations de Services Web et de commander la plate-forme IWSD sous jacente en actionnant des mécanismes de recouvrement ou de signalement d'erreur. Il a été conçu de sorte que le code soit robuste afin que le connecteur généré soit fiable.

DeWeL s'apparente aux méthodes et techniques telles que *Design by Contract* popularisée par le langage Eiffel [92], qui repose sur la définition formelle des préconditions et des postconditions d'un traitement. DeWeL a, cependant, été spécifiquement conçu pour créer des connecteurs en s'interfaçant avec des contrats WSDL des Services Web ciblés.

On peut également faire ici une analogie entre DeWeL et la programmation par aspect (Aspect Oriented Programming) représentée par des langages comme AspectJ [93-95] qui popularise le principe de « séparation des préoccupations » (Separation of concerns). Le grand intérêt ici est de pouvoir distinguer dans des objets logiciels différents, le code fonctionnel du code non fonctionnel. Dans le monde objet, idéalement, chaque tache spécifique devrait être la seule responsabilité d'une classe. Or, il suffit de lire le code source de n'importe quelle classe écrite dans un langage objet (C++, Java, C#, VB.NET…) pour se rendre compte que beaucoup de lignes de code sont consacrées à la synchronisation des accès aux ressources, à l'optimisation de l'utilisation des ressources, à la sauvegarde de certaines informations dans des bases de données ou autres (comme la persistance), au log des états du programme, à la vérification des paramètres entrants, au traitement des exceptions, …etc. (le terme anglais pour designer toutes ces activités est crosscutting concerns). Ces lignes de code ont tendance à être réécrites dans la plupart des méthodes des classes d'une application. Il en résulte que les classes sont peu réutilisables telles quelles par d'autres applications (autrement dit les classes sont couplées avec l'application). Il en résulte aussi que le changement de politique sur un de ces aspects de l'application (par exemple si on décide de ne plus loguer certaines informations) oblige la modification de nombreuses lignes de code éparpillées dans le code source, ce qu'on pourrait décrire comme un problème de localité textuelle: tout code concernant le même aspect devrait être regroupé dans un même endroit. Cela se rapproche du concept de modularité.

Cette problématique n'échappe pas, bien sûr, au monde des services. Elle est même amplifiée puisque le prestataire n'est pas censé connaître, à l'avance, dans quel contexte d'utilisation, son service sera utilisé. Cela dépend du type de clients et d'applications qui sont mis en jeux. Ces mécanismes ne peuvent donc pas être décidés à l'avance. Ils peuvent même varier dans le temps suivant la disponibilité des services (ressources) sur le Net.

Ainsi, DeWeL s'inspire de ces différentes approches et permet, à l'aide de connecteurs, de développer des mécanismes non-fonctionnels, des « aspects » de sûreté de fonctionnement, qui seront appliqués à l'exécution sur un Service Web spécifique suivant un contexte d'utilisation précis.

3.2 Définition et conception d'un DSL

Pour réaliser la conception de ce langage, nous nous sommes basés sur la méthodologie citée dans [96], qui consiste à effectuer en premier lieu, une étude préliminaire du domaine et de la famille considérée. Les informations recueillies par ces analyses sont ensuite utilisées comme données d'entrée principales du processus de conception d'un langage dédié.

Lors de la phase de conception du langage, la terminologie, les objets, les points communs et les variations provenant de l'analyse de domaine et de famille sont combinés avec les contraintes exprimées sur le langage. La Figure 3-1 empruntée à [96] montre une vue schématique de ce processus de conception tel qu'il est décrit par Thibault [97].

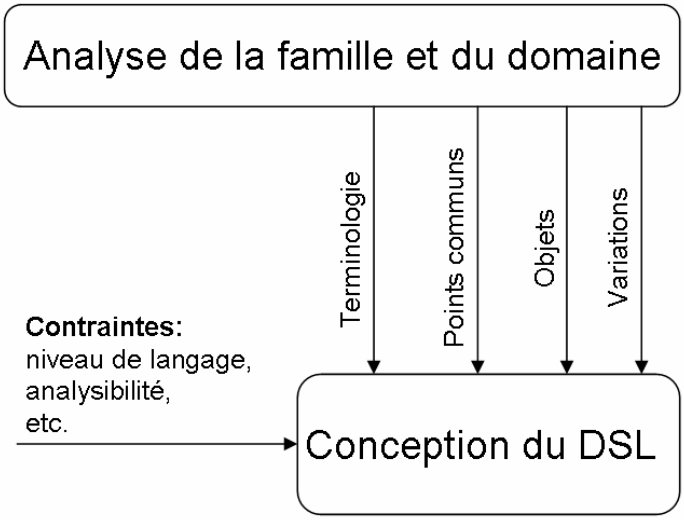

Figure 3-1: Processus de conception de DSL

L'analyse du domaine consiste à étudier un domaine et à en déterminer les éléments (ou objets) d'information réutilisables [98]. Une des principales limitations de l'analyse de domaine est qu'elle ne s'intéresse qu'aux points communs qui existent à l'intérieur d'un domaine, mais non pas aux variations. L'analyse de famille diffère donc de l'analyse de domaine en ce sens qu'elle s'intéresse autant aux variations qu'aux points communs. Ces analyses permettent d'identifier les éléments suivants:

- La **terminologie**: C'est le dictionnaire des termes généralement utilisés entre concepteurs. Elle sert à nommer les constructions du langage dans le but de les rendre familières aux experts du domaine.
- Les **objets** : Ce sont les éléments ou abstractions directement manipulables par le langage. Ils sont généralement traduits en types de données.
- Les **points communs**: Ce sont des besoins ou des hypothèses qui sont vrais pour tous les membres d'une famille de programmes.

- Les **variations**: Elles définissent comment les membres d'une famille de programmes peuvent varier. La forme finale des constructions introduites pour chaque point commun dépend des variations.

Le processus de conception prend également en considération les contraintes concernant le langage. Ces contraintes peuvent, par exemple, exprimer des besoins en analyse. Il peut s'agir dans ce cas de rendre décidable la vérification de propriétés critiques pour les programmes du domaine ou de rendre possible des optimisations automatiques. Il est important de noter que les besoins en vérification sont souvent à l'origine du développement d'un langage dédié. Les propriétés qui doivent être vérifiables sur un programme constituent les principales contraintes à respecter pendant le processus de conception.

A partir de cette étude, nous avons pu extraire les mots clés, les différents opérateurs et fonctionnalités du langage ainsi que les types et objets manipulables.

3.3 Les principales contraintes de DeWeL

L'analyse automatique de programmes reste trop souvent limitée par des résultats théoriques d'indécidabilité et de complexité algorithmique. Un des avantages d'un langage dédié est qu'il permet de réaliser certaines analyses qui seraient indécidables ou irréalisables dans le cas d'un langage généraliste. Et ce, en restreignant le langage ou en enrichissant les informations fournies par l'utilisateur. Le but de toutes ces vérifications est de détecter le plus grand nombre d'erreurs potentielles et cela le plus tôt possible dans le processus de développement. En effet, la sûreté et la robustesse d'une application dépendent en partie du temps nécessaire avant qu'une erreur soit détectée pendant la phase de développement [99]. Dans notre cas, plusieurs propriétés sont vérifiables à différents moments du processus de développement.

Dans le cadre de notre étude, il est, par exemple, possible de vérifier, à l'intérieur d'un même niveau d'abstraction, la non redéfinition d'un objet ou la cohérence des contraintes exprimées sur cet objet. Par exemple, le type associé à une variable doit être compatible à sa définition contenue dans le schéma XML.

La sûreté de fonctionnement est la contrainte majeure de notre langage. Le code généré par DeWeL doit être aussi exempt d'erreur que possible. La raison en est toute simple : le code compilé s'exécutant sur la plate-forme propriétaire IWSD, une erreur dans celui-ci pourrait faire défaillir l'application cliente ou pire, faire défaillir tout le système provoquant ainsi l'arrêt de toutes les applications en cours d'exécution. En effet, quand la famille des services ciblés est liée à l'utilisateur, comme dans le cas des Services Web, le développeur peut ne pas être un programmeur expérimenté. Par conséquent, DeWeL doit garantir des propriétés spécifiques pour préserver l'intégrité du serveur sous-jacent (la plate-forme IWSD) et pour empêcher un connecteur défectueux de corrompre l'application. On peut remarquer que la plupart de ces conditions ne seraient pas réalisables dans le contexte des langages usuels (ou *GPLs : General Purpose Languages*) en raison de leur expressivité sans restriction [80]. Voyons maintenant les restrictions imposées à DeWeL (ces restrictions sont reprises dans la Figure 3-2) :

Utilisation appropriée des ressources et terminaison: DeWeL doit utiliser des quantités appropriées de ressources comme la CPU, la mémoire, le stockage, ou la bande passante. Ceci suppose qu'un programme de DeWeL se termine. Bien que, l'arrêt soit indécidable en général, le DSL peut être conçu de sorte que cette propriété soit garantie, comme illustrée par divers DSLs existant (par exemple, Pems [100], Plan-P [78] et Devil [77, 101]).

Non-Interférence: Un programme DeWeL ne doit pas pouvoir examiner les données d'autres utilisateurs ou modifier arbitrairement les fichiers sur le serveur. La non-intervention avec d'autres aspects du serveur est garantie par une utilisation appropriée des ressources comme cela est mentionné ci-dessus. Par exemple, un programme DeWeL est appelé sur un serveur spécifique; les seules opérations possibles sont réalisées à partir de primitives contrôlées et permettent par exemple, la sélection et la manipulation des requêtes/réponses associées à un client spécifique pour un Service Web particulier.

Afin de simplifier son utilisation et d'éviter à un programmeur d'apprendre un nouveau langage, DeWeL emprunte sa syntaxe de base au C (pour les actions conditionnelles, les expressions arithmétiques et logiques, …etc.). Cependant l'analogie s'arrête là. En fait, DeWeL diffère énormément des langages de programmation usuels (GPLs) comme le C; les stratégies de recouvrement doivent pouvoir être seulement déclarées et paramétrées et les assertions exécutables définies en utilisant une syntaxe limitée. Les assertions définies par l'utilisateur correspondent à des pré et post actions qui encapsulent l'exécution du service. La Figure 3-2 donne les restrictions appliquées, le type d'erreurs correspondant qu'elles empêchent, et le type de propriétés critiques que DeWeL doit assurer. Enfin, des dispositifs spécifiques sont proposés dans DeWeL pour surmonter certaines de ces limitations.

<table>
<tr><th>Restrictions</th><th>Evitement d'erreurs</th><th>Propriétés vérifiées</th><th>Caractéristiques spécifiques</th></tr>
<tr><td>- Pas d'allocation dynamique
- Pas de pointeur
- Pas de référence</td><td>Bus error, Segmentation fault, not enough memory</td><td rowspan="3">Contrôle total de la mémoire et des ressources</td><td></td></tr>
<tr><td>- Pas de création de fichier</td><td></td><td>Fichier de log pour chaque utilisateur et service</td></tr>
<tr><td>- Pas d'indexation de tableaux</td><td>Dépassement de tableaux</td><td rowspan="2">Boucle contrôlée (foreach)</td></tr>
<tr><td>- Pas de boucle standard (while, for)</td><td rowspan="3">- Boucle infinie
- Gel du service</td><td rowspan="3">Terminaison</td></tr>
<tr><td>- Pas de fonction
- Pas de surcharge de méthodes</td><td>- fonctions prédéfinies
- méthodes spécifiques aux objets</td></tr>
<tr><td>- Pas de construction récursive</td><td></td></tr>
<tr><td>- Pas d'accès externe aux données des autres espaces utilisateurs ou aux ressources du système</td><td>Corruption de données</td><td>Non-interférence</td><td></td></tr>
</table>

Figure 3-2: Les principales caractéristiques de DeWeL

Les restrictions du langage sont très utiles pour effectuer une vérification statique efficace. Ces restrictions n'altèrent pas l'expressivité du langage et permettent de limiter à l'exécution le nombre de vérification. Par ailleurs, on peut noter ici que ce type de restrictions est reconnu et largement employé dans le monde du développement de logiciels critiques tels le ferroviaire (avec CENELEC 50128) ou l'avionique (DO-178/EUROCAE). En outre, le compilateur produit également automatiquement du code pour la vérification dynamique des paramètres des messages SOAP. Cela sera discuté par la suite.

3.4 Le langage DeWeL

L'exemple présenté dans la Figure 3-3 permet d'exhiber les possibilités du langage. Le but de cet exemple est de servir de guide pour définir, décrire et illustrer les caractéristiques de DeWeL. Celles-ci vont être exposées tout au long de cette section. Le chapitre 2 nous a permis de voir qu'un connecteur DeWeL était développé à partir d'un canevas lequel est composé de deux sections définies ci-dessous (cf. Figure 3-3):

- les ***traitements globaux*** s'appliquant à toutes les opérations du Service Web.
- les ***traitements spécifiques*** s'appliquant à une opération particulière (ItemSearch dans l'exemple de la Figure 3-3)

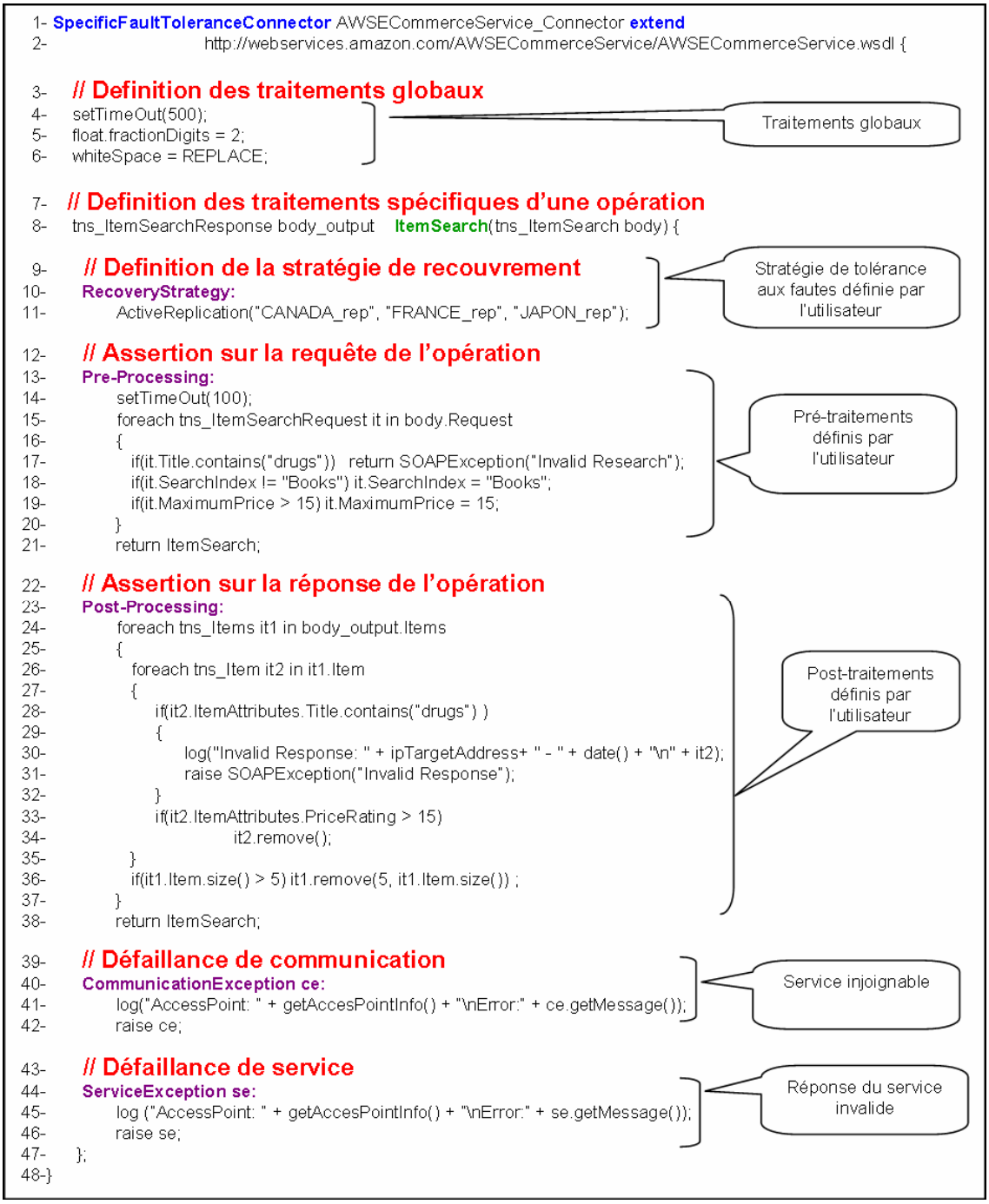

Figure 3-3: Exemple d'un programme DeWeL (pour Amazon)

Dans l'exemple de la Figure 3-3, les instructions insérées par l'utilisateur dans la zone de déclaration de traitements globaux (ligne 4-6) impacteront toutes les opérations du connecteur. Les traitements spécifiques sont quant à eux définis par l'utilisateur pour chaque opération. Ainsi, le point d'entrée à un traitement spécifique est la signature de l'opération elle-même (ici, *ItemSearch*). Celle-ci peut contenir, bien évidemment, des paramètres d'entrée et de sortie qui pourront être analysés, vérifiés ou modifiés dans la plupart des sous-traitements présentés ci-dessous:

- ***RecoveryStrategy*** (ligne 10 à 11): cette zone permet de spécifier le mécanisme de recouvrement ainsi que le nombre et les répliques qu'il souhaite utiliser.
- ***Pre-Processing*** (ligne 13 à 21): cette section permet d'effectuer des vérifications et modifications sur la requête reçue.
- ***Post-Processing*** (ligne 23 à 38): cette section permet d'effectuer des vérifications et modifications sur la réponse correspondante.
- ***CommunicationException*** (ligne 40 à 42): cette section permet de spécifier les actions à réaliser dans le cas où le service cible est injoignable.
- ***ServiceException*** (ligne 44 à 46): cette section permet de spécifier les actions à réaliser dans le cas où le service répond par un message d'erreur (une exception SOAP).

Ce langage doit être compris comme un moyen fiable permettant d'écrire des stratégies de tolérance aux fautes (détection, recouvrement, gestion des fautes, …) propre à un contexte d'utilisation donné. Le canevas lui-même est automatiquement généré à travers une analyse du document WSDL correspondant, permettant de capturer les opérations des services utilisant le protocole SOAP. La Figure 3-3 montre le canevas du connecteur dans DeWeL et la façon d'encapsuler un Service Web. L'exemple donné ici est un simple connecteur pour le Service Web d'Amazon, dans lequel la réplication active[8] a été sélectionnée (ligne 11) et différentes assertions simples ont été implémentées (ligne 13 à 21 pour les pré-traitements et 23 à 38 pour les post-traitements).

3.4.1 Définition d'un connecteur DeWeL

La définition d'un connecteur passe par l'utilisation du mot clé « `SpecificFaultToleranceConnector` ». Un connecteur hérite toujours d'un document WSDL. Cet héritage se fait grâce à l'utilisation du mot clé « `extend` » (ligne 1-2). Cette description s'apparente en fait à la définition d'une classe. Dans le corps du connecteur, l'utilisateur peut instaurer des traitements de sûreté de fonctionnement globaux ou spécifiques. Pour chaque opération du service, il peut manipuler des instructions, des fonctions ou des variables de types spécifiques au langage DeWeL. Les instructions et les fonctions sont prédéfinies par le langage. Les types sont, quant à eux, soit prédéfinis, soit construits à partir du schéma XML inclus dans le contrat d'origine.

[8] Parmi les six contrats disponibles d'Amazon, trois répliques du service Web d'Amazon ont été utilisées dans cet exemple (respectivement nommé CANADA_Rep, FRANCE_Rep, JAPON_Rep) fournissant ainsi une meilleure robustesse vis-àvis des fautes matérielles et logicielles.

Pour des raisons de clarté, la grammaire du langage (BNF - Backus-Naur Form) est présentée en annexe (cf. Annexe A.1). Nous ne présenterons ici que certaines particularités du langage au travers de l'exemple présenté en Figure 3-3.

Les traitements globaux (ligne 4-6) permettent d'installer une politique commune à toutes les opérations du Services Web. La fonction interne « `setTimeOut` » est spécifique au connecteur. Celle-ci permet de configurer la fenêtre de temps dans laquelle doit répondre le prestataire et ainsi de détecter un gel de service (cf. ligne 4). L'attribut « `totalDigits` » appliqué aux types « `float` » permet de spécifier la précision maximale des nombres décimaux (cf. ligne 5). Tous les réels supérieurs à cette précision génèreront par conséquent une exception. La variable interne prédéfinie « `whiteSpace` » peut être appliquée de manière globale, en supprimant par exemple, tous les espaces blancs inutiles contenus dans la requête ou la réponse SOAP (cf. ligne 6). Ceci aurait pour but de diminuer la taille des messages et ainsi d'augmenter la vitesse de transfert et de traitement des messages côté client ou prestataire.

Comme on peut le constater, la conception d'un connecteur passe par la manipulation de variables. Ces variables sont toutes issues d'un type spécifique du langage DeWeL. Ceci est abordé dans la section suivante.

3.4.2 Les types DeWeL

Les types DeWeL sont une traduction des types définis dans les schémas XML manipulables par le connecteur. Ces types sont des classes au sens des langages objets. Ils portent les mêmes noms que ceux contenus dans le schéma XML associés au service cible. La correspondance entre ces deux modèles est forte. La manipulation des variables issues des types DeWeL se déduit donc de la lecture des schémas XML correspondant

Il existe deux catégories de types :

- les ***types primitifs*** sont les types de base du langage. Ils sont déduits du schéma des schémas XML vu en Figure 1-11. La description de tous les types de base est présentée en annexe A.1.
- les ***types construits*** sont les types créés dynamiquement à partir du contrat du Service Web. Ils sont propres à chaque connecteur.

Chaque type peut posséder plusieurs facettes. Les facettes sont des attributs associés aux types. Elles permettent de contraindre l'espace lexical[9] ou l'espace de valeur[10] des types primitifs. En effet, les valeurs d'un type de données primitif peuvent répondre à des critères déterminés rigoureusement à l'aide de facettes. Une facette est un aspect de la définition d'une valeur simple. Les valeurs peuvent être restreintes par rapport à une liste de possibilités, à des limitations, à un modèle d'expression régulière, à une longueur et à un comportement. Ainsi, agir sur l'espace de valeurs consiste à modifier les caractéristiques d'un type, en imposant, par exemple, que les nombres soient inférieurs à une certaine valeur, en définissant la précision

[9] Un espace lexical est l'ensemble des littéraux valides pour un type de donnée.
[10] Chaque valeur dans l'espace de valeurs du type de donnée est dénotée par un ou plusieurs littéraux dans son espace lexical.

des nombres décimaux ou déterminant la longueur maximale des chaînes de caractères comme montré dans l'exemple ci-dessous[11]:

```
string.maxLength = 30 ;
```

Les facettes s'appliquent directement sur les types. Elles imposent des contraintes sur le domaine de valeur. Les différentes facettes applicables sur les types de base sont présentées en annexe A.1.

Dans les schémas XML, les facettes permettent également de réduire le champ de valeur d'un type. Cependant, une fois qu'un prestataire a réduit un type et l'a inséré dans un document WSDL, celui-ci se retrouve figé.

Ces restrictions correspondent généralement aux limites des prestataires. Elles sont appliquées pour tous les clients et donc dans toutes les applications se basant sur ce service Web. Un des avantages de DeWeL est de pouvoir effectuer cette restriction et vérification à l'exécution pour chaque connecteur créé à partir d'un document de base. Dans un souci de réaliser proprement la séparation des préoccupations, la définition de ces facettes devrait donc se situer dans les documents WSDL des connecteurs plutôt que dans le contrat WSDL d'origine. Le contrat du connecteur du prestataire devient alors le contrat de référence, remplaçant ainsi le contrat d'origine voué à disparaître.

3.4.3 Les variables

Les variables DeWeL sont les éléments (ou objets) centraux de notre langage. Les assertions exécutables utilisent les valeurs de ces variables et permettent de détecter des incohérences qui sont alors signalées.

3.4.3.1 Caractéristiques des variables

Les variables manipulées dans le connecteur sont de plusieurs catégories :

- ***variables internes prédéfinies:*** ce sont des variables propres à tous les connecteurs créés à partir du langage DeWeL (exemple : ipSourceAddress permettant de connaître l'adresse source du client).
- ***variables internes spécifiées:*** ce sont des variables créées par l'utilisateur dans le but de réaliser des vérifications non-fonctionnelles.
- ***variables externes :*** ce sont les variables récupérées à travers les requêtes et réponses SOAP. Elles sont donc toutes construites à partir d'un nœud XML.

Les variables internes et externes ne sont donc pas de même nature. En effet, les variables internes sont déclarées directement dans le connecteur. Elles ne proviennent pas d'un nœud XML. Bien que n'étant pas de même nature, ces variables se manipulent pourtant de la même façon. Lors d'affectations successives entre des types de même nature mais de catégorie différente, les manipulations internes ne sont pas du tout les mêmes. Modifier la valeur d'une variable externe, implique de modifier le nœud XML correspondant dans le message reçu.

[11] L'accès a une facette d'un type se fait à l'aide de l'opérateur « **.** » .

Toute cette mécanique est en fait cachée par une abstraction de plus haut niveau fournie par le langage DeWeL évitant ainsi certaines erreurs maladroites.

3.4.3.2 Manipulations des variables

La manipulation des variables du langage peut se faire à l'aide de plusieurs composants :

- ***Les opérateurs:*** Les types DeWeL sont dotés d'un certain nombre d'opérations de base exprimées par des opérateurs, comme par exemple l'addition. Le nombre d'opérateurs et leur signification varie, bien sûr, suivant le type (tous ces opérateurs sont listés en annexe A.1).

  Ex : positiveInteger b = 2 ;

  positiveInteger a = 3 + b ;

- ***Les méthodes:*** Chaque type DeWeL possède en plus, des méthodes spécifiques permettant de résoudre des problèmes associés au domaine (ici, la sûreté de fonctionnement). La méthode « *contains* » associée au type *string* permet par exemple de vérifier qu'une sous-chaîne de caractères est incluse dans la chaîne de base.
- ***Les attributs:*** Seuls les variables de types construits possèdent des attributs. Les types construits proviennent des types complexes issus des schémas. Les attributs sont en fait des sous variables de types primitifs ou construits[12].

3.4.3.3 Portées des variables

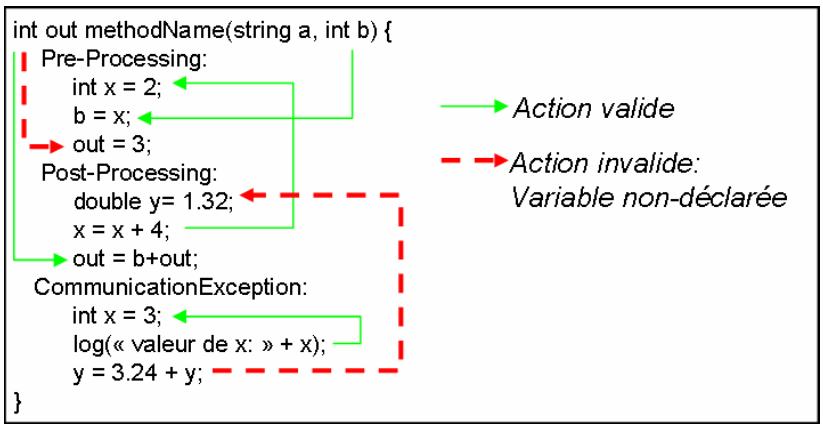

Figure 3-4: Portée des variables

Les variables internes peuvent être soit globales au connecteur soit locales à une opération. Pour qu'une variable soit globale, il faut qu'elle soit déclarée et définie dans la partie des traitements globaux (ligne 4-6 de la Figure 3-3). Pour qu'une variable soit locale, il faut la définir et la déclarer en priorité dans un des sous-traitements associés à une opération. Les règles de portée des variables au sein des traitements spécifiques d'une opération s'effectuent de la façon suivante (cf. Figure 3-4):

[12] L'accès a ces attributs et aux méthodes allouées à la variable se fait à l'aide de l'opérateur : « **.** ».

- une variable interne définie dans un Pre-processing est connue dans tous les autres sous-traitements de l'opération sauf RecoveryStrategy.
- une variable interne définie dans un Post-processing, CommunicationException, ServiceException ou RecoveryStrategy n'est connue que de lui-même.
- les paramètres d'entrée de l'opération sont accessibles dans les traitements Pre-Processing, Post-Processing, CommunicationException et ServiceException.
- le paramètre de sortie n'est connu que du Post-Processing.

Ces règles permettent de s'assurer qu'on ne peut pas utiliser une variable qui n'aurait pas précédemment été initialisée.

3.4.3.4 Les variables à mémoire

Le connecteur étant éphémère, les variables créées jusqu'à présent sont sans mémoire. Elles sont donc supprimées dès la fin de l'activité du connecteur. Afin de pouvoir réaliser des assertions plus complexes prenant en compte des informations de requêtes passées, l'utilisateur peut vouloir mémoriser certaines variables pour les réutiliser par la suite.

Fichier : AWSECommerceService_Connector.dwl

```
1- SpecificFaultToleranceConnector AWSECommerceService_Connector
     extend http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl {

2-  tns_CartAddResponse body_output    CartAdd(tns_CartAdd body) {
3-    Pre-Processing:
4-      string memory::clientID;
5-      string memory::panierID;
6-      string memory::itemID;
7-      optional{
8-         if((body.AWSAccessKeyId == clientID)&&
9-           (body.CartId==panierID)&&
10-          (body.Item.ASIN==itemID))
11-              return SOAPException("This item is already in the cart");
12-      }
13-      if(body.Item.Quantity > 1)  {
14-           log("Error in request: " + body);
15-           body.Item.Quantity = 1;
16-      }
17-      clientID= body.AWSAccessKeyId;
18-      panierID= body.CartId;
19-      itemID= body.Item.ASIN;
20       return CartAdd;
21-   };
22- }
```

Fichier : Connector_Memory.xml

```
<connector name=«  AWSECommerceService_Connector  »>
  <globalProcessing />
  <operation name=« CartAdd »>
      <pre-processing>
            <clientID type=« string »> RU5768TIYU54</clientID »>
            <panierID type=« string »>145673</panierID>
            <itemID type=« string »>ISBN-798584</itemID>
      </ pre-processing>
      <post-processing />
      <communicationException />
      <serviceException />
  </operation>
</ connector>
```

Figure 3-5: Exemple d'utilisation des variables à mémoire

Pour cela, DeWeL offre la possibilité de créer des variables persistantes. Cela peut être effectué à l'aide du mot clé « `memory` » appliqué sur la déclaration d'une variable. Un exemple est fourni dans la Figure 3-5. Dans cet exemple, le pré-traitement s'effectue sur une autre opération du Service Web d'Amazon nommé « CartAdd ». Celle-ci permet d'ajouter un article dans un panier d'achat. Suite à une erreur de l'opérateur (par exemple, un double clic sur l'application cliente), deux requêtes successives identiques sont envoyées au service et ajoutent ainsi deux fois le même élément dans le panier. Le pré-traitement présenté ici permet d'éviter ce problème en sauvegardant le dernier message reçu.

Celui-ci sauvegarde trois éléments de la requête : l'identifiant du client (clientID, ligne 4), l'identifiant du panier (panierID, ligne 5) et l'identifiant de l'article à insérer au panier (itemID, ligne 6). Cette sauvegarde permet de vérifier que le même article n'est pas inséré deux fois de suite dans le même panier. En effet, après réception de la requête, on peut vérifier si l'article qu'elle contient n'est pas identique à celui-ci de la requête précédente (ligne 8 à 11) au travers de la variable mémorisée : ItemID. Nous verrons au paragraphe 3.4.6 pourquoi ces instructions sont contenues dans un bloc optionnel. On peut également remarquer que le connecteur vérifie à chaque requête, la quantité d'articles mis dans le panier (cf. ligne 13 à 16).

Les variables persistantes peuvent être déclarées soit en début du traitement global soit au début de traitements spécifiques d'une opération. Contrairement aux variables internes spécifiées, elles ne doivent être pas initialisées lors de la déclaration (ligne 4 à 6). Lors de la compilation, un fichier de sauvegarde associé au programme DeWeL est créé. Ce fichier contient toutes les variables à mémoire. A la génération de code, les lignes de chargement de ces variables (synchronisation, ouverture du fichier, lectures des variables, …etc.) sont substituées aux lignes d'initialisation écrites dans le programme DeWeL (ligne 8 à 11). Lorsque l'on utilise des variables à mémoire, ce fichier devient une ressource critique partagée entre les différents connecteurs actifs associés. Il est donc soumis au problème d'accès concurrent sur lequel plusieurs solutions de la littérature courante fournissent des réponses valables (problème du lecteur-écrivain, section critique, …etc.). La gestion d'accès à ce fichier est entièrement prise en charge par DeWeL lors de la génération de code. Ceci étant bien sûr totalement transparent pour le développeur.

Le fichier « Connector_Memory.xml » présenté dans la Figure 3-5 contient la valeur des variables mémorisées après une passe dans le pre-processing. L'exemple ci-dessus permet de voir l'utilité de ce type de variable en réalisant des assertions à partir de requêtes passées.

3.4.4 Les fonctions internes

Les variables issues de DeWeL peuvent être manipulées aux travers d'opérateurs ou méthodes. En plus de ces propriétés, le langage DeWeL fournit une suite de fonction permettant à l'utilisateur de récolter des informations non-fonctionnelles qui vont l'aider dans sa prise de décision et dans la création du connecteur.

On peut distinguer deux catégories de fonction :

- les ***fonctions privées*** (private): ces fonctions ne peuvent être utilisées qu'à l'intérieur du connecteur (dans le programme DeWeL).
- les ***fonctions publiques*** (public): ces fonctions sont utilisées à l'intérieur du connecteur mais peuvent aussi être appelées de l'extérieur. Cela signifie que ces fonctions sont en fait des opérations propres au connecteur. Elles peuvent donc être décrites dans un contrat au même titre que les opérations de base que l'on encapsule.

Elles seront donc incorporées dans la création du contrat WSDL associé au connecteur et appelables par un client au travers de requêtes SOAP.

Une liste de fonctions non-exhaustives est présentée ci-dessous. Leur statut est spécifié entre parenthèse pour chacune d'entre elle :

- getCreationDate (public) : permet de récupérer la date de création du connecteur.
- getLastUpdate (public) : permet de récupérer la dernière date de mise à jour du connecteur.
- getDate (private) : permet de récupérer la date courante.
- getResponseTime (private) : permet de fournir à l'utilisateur le temps de réponse du service (cette fonction retourne -1 si elle est appelée ailleurs que dans un traitement autre que Post-Processing ou ServiceException).
- getAverageResponseTime (public) : donne le temps de réponse moyen du service (renvoi -1 si l'opération n'a jamais atteint un Post-Processing ou un Service-Exception).
- getCommunicationExceptionNumber (private) : donne le nombre total de Communi-cationException survenu depuis le début du cycle de vie du connecteur.
- getAccessPointInfo (private) : fournit des informations sur le point d'accès concret qui a produit la réponse.
- SOAPException (private) : génère une exception spécifique de type SOAP.
- log (private) : permet de créer un historique.
- ActiveReplication (private) : sélectionne la réplication active comme mode de recouvrement (voir chapitre 4).

La technologie des Services Web et les protocoles extensibles utilisés n'ayant pas encore atteint totalement leur maturité, le langage DeWeL ne peut être figé. Ainsi, d'autres fonctions publiques ou privées peuvent être insérées dans le langage. Ces fonctions peuvent être insérées sous forme de librairie. On peut prévoir, par exemple, l'essor de fonctionnalité concernant le traitement des pièces jointes qui peuvent être incluses dans le message SOAP (détection de virus, compression d'images, conversion audio), d'envoi de SMS, d'email ou de fax pour prévenir par exemple le prestataire d'une défaillance sur son service. Ces librairies doivent, bien sûr, être incorporées dans la plate-forme sous-jacente (le support d'exécution).

3.4.5 Les instructions

Les zones *`RecoveryStrategy`*, *`Pre-Processing`*, *`Post-Processing`*, *`CommunicationException`* et *`ServiceException`* autorisent des instructions supplémentaires facilitant ainsi le traitement et la manipulation des données.

Ainsi, ces sous-traitements possèdent des instructions proches du C, telles que les instructions conditionnelles comme le « *`if`* (*`else`*) » ou le « *`switch`* ». Le « *`switch`* » est toutefois contraint puisqu'il faut ou moins un « *`case`* » et que chacun soit terminé par un « *`break`* ».

Le choix d'un langage impératif plutôt que déclaratif a été décidé afin de pouvoir écrire des assertions complexes. De plus, d'après l'expertise que l'on a pu faire sur l'utilisation des Services Web, les deux principales plates-formes pour développer des Services Web (.NET et Java) génèrent côté client, des stubs utilisant les langages à objet tels que C++, Java, C#

possédant ce même type d'instruction. Il nous a donc semblé logique de garder ce mode et de ne pas contraindre un développeur inexpérimenté, à apprendre un nouveau type de programmation.

Les instructions « `return` » ou « `raise` » permettent toutes les deux d'interrompre l'exécution du connecteur. Le mot clé « return » permet de retourner :

- la requête au prestataire en sortie du Pre-Processing,
- la réponse au client en sortie du Post-Processing,
- un message d'erreur au client. La stratégie de recouvrement, si elle est précisée, est, dans ce cas, annulée, car elle peut être considérée comme non appropriée.

L'instruction « raise » permet de lever une exception et ainsi de solliciter le mécanisme de recouvrement défini par l'utilisateur. Si aucune stratégie de recouvrement n'est précisée, un message d'erreur est retourné au client.

Les instructions traditionnelles telles que le « `for` » ou le « `while` », sont prohibées en DeWeL dans le but d'éviter les boucles infinies. Pour les mêmes raisons, la récursivité est proscrite. Il est donc impossible de créer une fonction, d'appeler une autre opération du service de base ou d'un Service Web externe. Cette dernière contrainte a pour but d'éviter un effet cascade en enchaînant une succession de pré-traitements de connecteurs différents ou identiques. Ainsi, le parcours de séquence d'éléments XML ne peut se faire que par l'intermédiaire de l'instruction « `foreach` » permettant de contrôler chaque objet.

Les pointeurs et l'indexation de tableau sont également interdits pour éviter un dépassement de tampon (buffer overflows). DeWeL est donc un langage proche du C tout en imposant des restrictions fortes afin d'éviter des erreurs critiques au même titre que le langage Cyclone [102].

3.4.6 Les instructions optionnelles

Les schémas XML permettent de spécifier certains attributs qui octroient des propriétés particulières aux éléments définis. Ainsi, l'attribut « *minOccurs* » (lorsqu'il est égal à 0) et « *nillable* » (lorsqu'il est égal à vrai) permettent respectivement à un élément d'être optionnel ou d'avoir une valeur nulle. Ces éléments sont donc non-définis. Il en est de même pour les variables à mémoire, qui sont déclarés sans être définis.

L'utilisation sans précaution d'un sous-élément ou d'une méthode de cette variable ou attribut DeWeL à l'exécution entraîne dans un langage naturel une erreur irrécupérable (« segmentation fault » en C ou « NullPointerException » en Java). Ainsi, lors de la génération de code, le compilateur DeWeL encapsule les instructions faisant intervenir l'utilisation de ce type de variable par des assertions (automatiquement générées) capables de vérifier que l'accès aux attributs ou aux méthodes souhaités de l'élément est correct. Si lors de l'écriture du programme, cette instruction est préfixée par le mot clé « `optional` », elle sera ignorée lors de l'exécution si elle ne peut être vérifiée (cf. Figure 3-5 ou Figure 3-7). Dans le cas contraire, une exception SOAP sera levée et retournée au client. Pour remarque, seuls les éléments optionnels ou les éléments inclus dans des tableaux ou des listes peuvent être ajoutés ou supprimés des messages SOAP. Pour ce faire, DeWeL fournit l'opérateur d'affectation (=) pour créer et affecter une valeur à un nœud et la méthode « *remove* » pour supprimer ce nœud. Lors de la compilation, l'utilisateur est prévenu par des messages d'alerte que certaines instructions utilisent des variables optionnelles et donc que ces assertions ne pourront être effectuées que si l'élément existe.

3.4.7 Le paramétrage des connecteurs

Afin de pouvoir réaliser des assertions et autres mécanismes de sûreté à partir de propriétés non-fonctionnelles spécifiques au client, DeWeL offre la possibilité à un utilisateur d'étendre une opération de base en lui injectant des paramètres d'entrée supplémentaires. Ces paramètres permettent de configurer le connecteur selon des critères spécifiques propres au client. Cette possibilité est assurée par les mots clés « `extend with` » (cf : Figure 3-6). Si les types insérés correspondent à des éléments non primitifs, les schémas correspondants doivent être intégrés en débutant le programme du connecteur par le mot clé « `import` ».

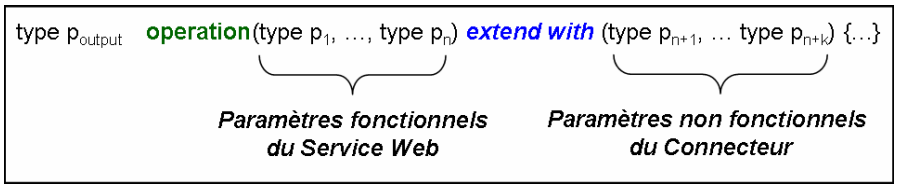

Figure 3-6: Paramétrage du connecteur

La Figure 3-7 présente un exemple de connecteur pour le Service Web « *DictService* » de la société Aonaware[13]. Celui-ci contient une opération « *define* » permettant de rechercher la définition d'un mot dans un dictionnaire. Dans cet exemple, la variable « engine » de type construit (ClientEngine) est un paramètre non-fonctionnel propre au connecteur. Ce type (ClientEngine) est connu du langage grâce à l'importation du schéma « non_functional_schema.xsd » (ligne 1). Il permet d'adapter la réponse suivant les ressources disponibles du client. Ici, on réduit la taille de la définition du mot passé en paramètre de la réponse (cf. ligne 5 - Figure 3-7).

```
1-  #import "non_functional_schema.xsd"

2- SpecificFaultToleranceConnector DictService_Connector extend http://services.aonaware.com/DictService/DictService.asmx?WSDL {

3-    String defineResult  define(string word)  extend with (ClientEngine engine) {
4-      Post-Processing:
5-           optional { if((engine.type = "PDA")&&(engine.resource.level == "very low")) defineResult.resize(50); };
6-           return define;
7-     };
8- }
```

Figure 3-7: Exemple de programme DeWeL pour DictService

Ces paramètres propres au connecteur seront bien sûr précisés dans son document WSDL. Ils seront insérés dans les en-têtes des requêtes SOAP clientes puis consommés par la plateforme IWSD lors de la réception de la requête. Cela signifie que la requête transmise au prestataire ne contient plus que les paramètres précisés dans le contrat de service original.

La paramétrisation des connecteurs est en fait nécessaire dans le cas où certaines assertions ne sont effectuées qu'en fonction de critères propres au client. Elle permet aux connecteurs de jouer pleinement leur rôle : contrôler et vérifier des propriétés côté prestataires mais également côté clients.

[13] Voir : http://services.aonaware.com/

3.5 Le Processus de Génération de code et compilation

Cette section décrit comment, à partir d'un contrat WSDL, on peut générer un connecteur dédié. La Figure 3-8 présente le processus de génération de code composé de 4 étapes. Chacune d'elles implique un outil spécifique.

Parmi les quatre outils participant à ce processus, trois (présentés respectivement dans les trois premières étapes) ont dû être spécifiquement développés pour s'assurer de la fiabilité et de la robustesse des connecteurs créés.

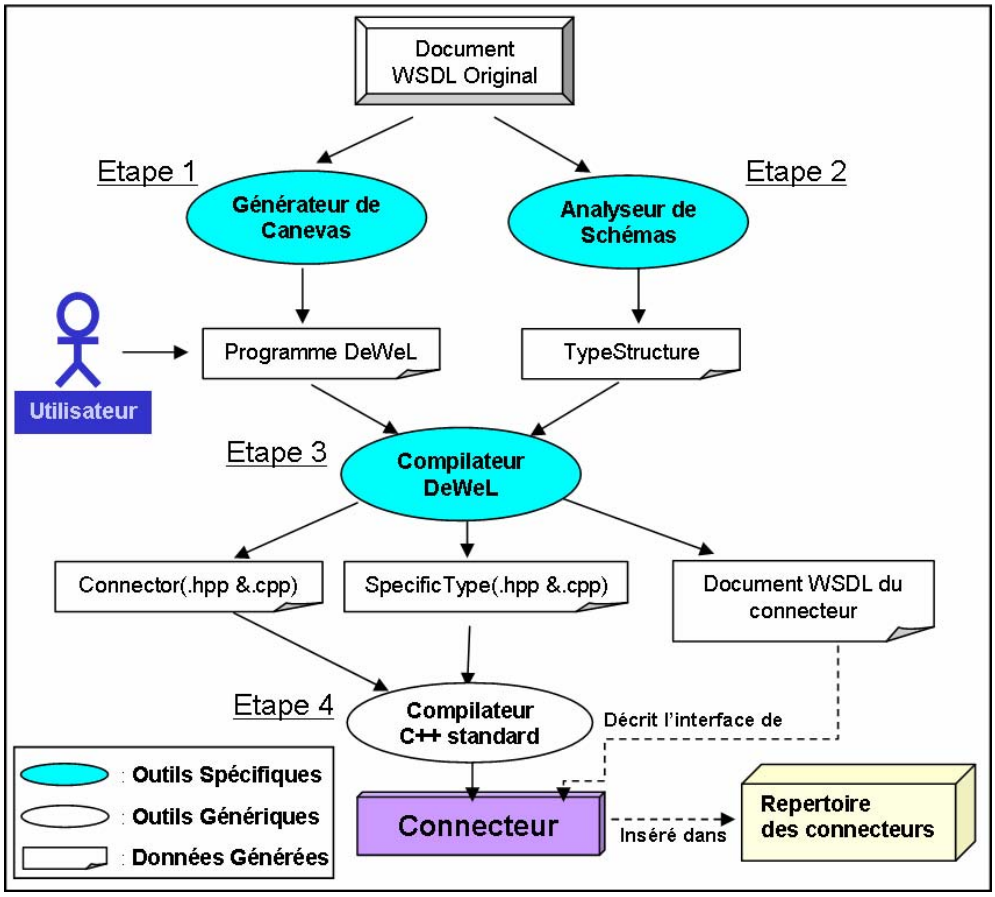

Figure 3-8: Le processus de génération de code

3.5.1 Génération du canevas

Tout d'abord, la validité du document WSDL et toutes les opérations internes fournies par le Service Web correspondant, sont vérifiées. Cette phase permet d'examiner que le contrat est syntaxiquement correct. Elle valide aussi l'existence de tous les types utilisés et la non-duplication d'opérations. Le canevas DeWeL du connecteur est produit à partir de la signature de ces opérations. Par la suite, l'utilisateur peut insérer ses propres mécanismes de sûreté de fonctionnement dans les sections appropriées du canevas. C'est le ***Générateur de canevas*** qui est en charge de cette étape.

Cet outil analyse le document WSDL et récupère pour chaque service qui y sont décrits les différentes opérations SOAP/HTTP qu'ils supportent. Il peut donc générer plusieurs canevas (un pour chaque service représenté dans le document).

La Figure 3-9 présente, à travers les exemples de Google et Amazon, les styles d'échange SOAP qui sont principalement supportés par l'outil (ici, RPC ou document) ainsi que le protocole de transport utilisé par défaut (HTTP).

```
Pour Google:
<binding name="GoogleSearchBinding" type="typens:GoogleSearchPort">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
  ...
 </binding>

Pour Amazon:
<binding name="AWSECommerceServiceBinding" type="tns:AWSECommerceServicePortType">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
  ...
 </binding>
```

Figure 3-9: Les liaisons (<binding>) de Google et Amazon

En fait, SOAP permet de mettre en œuvre plusieurs styles d'échange (cf. Figure 3-10) :

- **Message à sens unique (one-way)** : il correspond à l'envoi unidirectionnel d'un message SOAP. Bien qu'il puisse être supporté par DeWeL, celui-ci ne représente en fait qu'un intérêt mineur puisque seuls les pré-traitements sont autorisés.
- **Requête/réponse** : ici, l'envoi d'un message de requête est suivi d'un message de réponse. Ce style est en fait défini implicitement (via la liaison générique SOAP/HTTP). Le style requête/réponse s'applique notamment à la mise en oeuvre de l'appel de procédure distante (RPC : Remote Procedure Call), pour lequel SOAP définit une représentation spécifique. Le style document est aussi défini implicitement comme le complément du RPC : il désigne les requêtes/réponses au format standard SOAP qui ne sont pas une représentation explicite de l'appel de procédure distante.

A l'heure actuelle, notre implémentation ne supporte que les styles d'échanges transportés par le protocole HTTP.

L'analyse du document WSDL est donc une analyse descendante. Pour chaque service, on recherche les liaisons SOAP/HTTP. A partir de ces liaisons, on récupère les opérations du service. Ces opérations comportent, entre autres, un message d'entrée et un message de sortie. Chacun de ces messages contient des paramètres. Cette analyse permet donc d'extraire les signatures des méthodes potentiellement empaquetables par l'utilisateur.

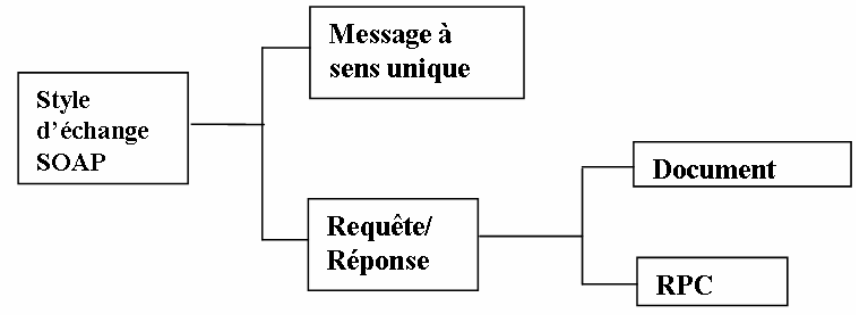

Figure 3-10: Taxonomie des styles d'échange SOAP

3.5.2 Analyse et création de la TypeStructure

Cette deuxième étape est seulement nécessaire quand le document WSDL inclut des schémas XML décrivant les types spécifiques définis dans le service Web. Dans ce cas, une

structure de données particulière nommée « ***TypeStructure*** » est produite pour contrôler la manipulation de ces types spécifiques dans le canevas du connecteur. L'***Analyseur de Schémas*** que nous avons développé est en charge de créer cette structure de données. La TypeStructure inclut tous les types présents dans le Service Web et contrôle l'accès à ces paramètres spécifiques dans la requête SOAP. La TypeStructure a donc deux fonctions essentielles :

- Valider les assertions de l'utilisateur utilisant des types spécifiques dans le programme DeWeL. Cette phase est nécessaire lors de la vérification des types effectuée dans le compilateur. Par exemple, les schémas XML autorisent, à l'aide de l'attribut « *fixed* », de contraindre une valeur à l'élément. Une modification de cette variable dans le programme DeWeL génèrera une erreur lors de la compilation. Cette erreur sera détectée par cette structure.
- Générer ces types spécifiques de manière à ce qu'ils soient compréhensibles et manipulables par le connecteur.

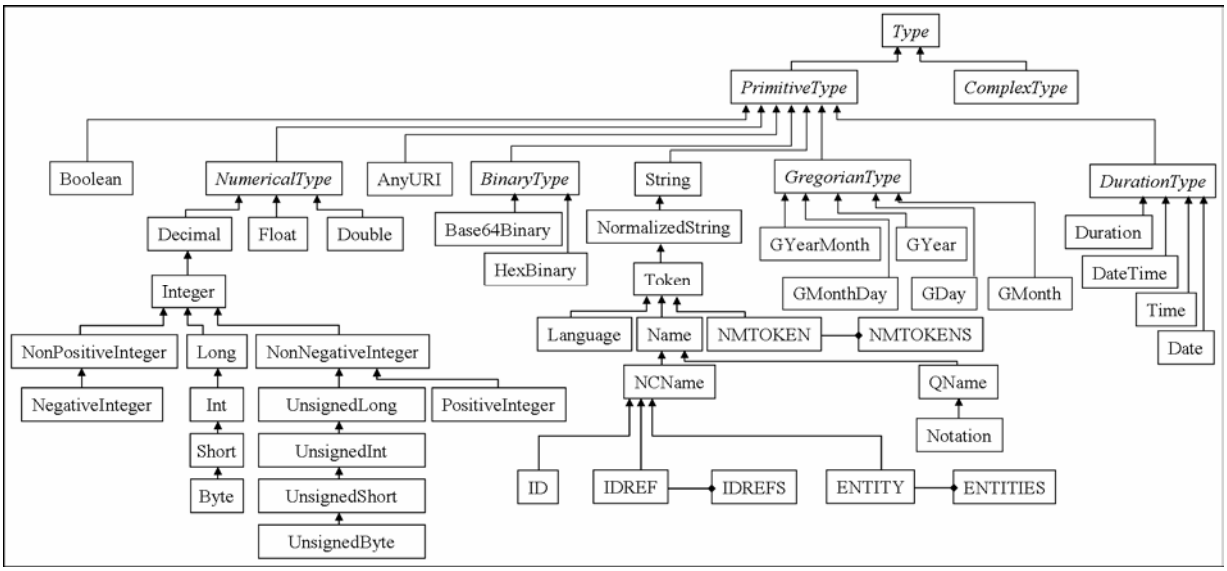

Figure 3-11: Diagramme UML de la hiérarchie des types C++

Cette phase de génération de type est une phase complexe et indispensable. Lors de la génération de code, chaque type DeWeL est traduit en une classe C++ correspondante. Ainsi, les variables DeWeL sont traduits en objets de classe C++ prédéfinis. La Figure 3-11 présente la hiérarchie des différentes classes C++ existant dans la plate-forme. Ces classes ont été directement construites à partir de l'analyse du schéma des schémas XML[14] (cf : Figure 1-11). Pour des raisons de clarté, la relation d'héritage qui unit les types NMTOKENS, IDREFS et ENTITIES à la classe *ComplexType* n'est pas représentée.

DeWeL peut être perçu comme un langage intermédiaire entre les schémas XML et les classes C++ présentées dans la Figure 3-11. En effet, DeWeL permet de manipuler de façon simple et sûre les nœuds XML de la requête SOAP correspondant aux paramètres d'entrée/sortie. Cette manipulation ne peut pas se faire directement au niveau des classes C++

[14] Voir: http://www.w3.org/2001/XMLSchema.xsd.

car l'utilisateur aurait alors la possibilité d'effectuer des manipulations dangereuses telle que l'effacement du nœud ou pire accéder au nœud racine et supprimer toute la requête. Il pourrait également insérer des valeurs invalides aux paramètres qui seraient par la suite transmis au service correspondant. Enfin, les instructions permises en C++ permettraient à un développeur d'introduire de manière maladroite ou malicieuse des morceaux de code réalisant des boucles infinies ou de dépassement de limites de tableaux, par exemple. Ce genre d'instructions critiques est à proscrire absolument car elles sont une source d'erreur entraînant des défaillances graves. Grâce au langage DeWeL, l'utilisateur est donc contraint de manipuler des types de plus haut niveau (types DeWeL) et des instructions restreintes et contrôlées. Chacun de ces types ayant ses propres facettes et opérateurs (cf. Annexe A.1), les instructions décrites plus haut sont soit interdites par le langage (pas d'instruction « *delete* »), soit contrôlées à la compilation (processus de vérification de types).

```
class Type {

private:
    DOMNode* node;

public:
    // Constructeur
    Type(DOMNode* n)  throw (XMLTypeException);

    // Destructeur
    ~Type();

    // Accesseur
    DOMNode* getNode();
    void setNode(DOMNode* n);

    string toString();
};
```

Figure 3-12: La classe de base - *Type*

Lors de la génération de code, chaque variable du programme DeWeL est translatée en objet C++ de la classe correspondante. La classe mère à tous les types C++ est la classe *Type*. Celle-ci est présentée partiellement en Figure 3-12. La classe « *Type* » possède une variable d'état « *node* » de type « *DOMNode* ». Ce type est le type de base de la librairie Xerces C++ [103] utilisée dans ce projet. Cette variable « *node* » contiendra à l'exécution le nœud correspondant au paramètre de la requête SOAP à analyser. Des méthodes spécifiques à chaque sous-classe permettront de récupérer, de convertir en type équivalent C++ et de manipuler la valeur de l'élément simple. C'est cet élément qui sera directement manipulé au travers des types DeWeL.

```
class PrimitiveType: public Type {

private:
    string soapValue;

public:
    // Constructeur
    PrimitiveType(DOMNode* n)  throw (XMLTypeException);

    // Destructeur
    ~PrimitiveType();

    // Accesseur
    string getSoapValue();
    void setSoapValue(string s);

protected:
     virtual void convert_soapValue2cppValue() = 0;
     virtual void convert_cppValue2soapValue() = 0;
};
```

Figure 3-13: Le type abstrait: PrimitiveType

Ainsi, la manipulation d'un type DeWeL masque des instructions complexes de conversion et de vérification de types.

La classe abstraite *PrimitiveType* permet de décrire tous les types primitifs de base autorisés et manipulables par la plate-forme. Cela signifie que l'élément XML n'a qu'un seul fils et que celui-ci représente en fait la valeur attribuée à ce paramètre :

```
<maxResults xsi:type="xsd:int">10</maxResults>
```

Si l'on regarde plus attentivement la classe PrimitiveType (cf. Figure 3-13), on peut constater que celle-ci possède désormais la variable d'état : *soapValue*. Cette variable contient la valeur lexicale de l'élément (ici, la chaîne de caractère « 10 »). Ces accesseurs sont forcément « publics » puisque ces méthodes doivent être accessibles de l'extérieur de la classe pour contrôler ou modifier cette valeur (par exemple, la méthode « setSoapValue » modifie à la fois la valeur *soapValue* mais aussi la valeur contenue dans le nœud XML). Enfin, cette classe contient également deux méthodes virtuelles pures :

- convert_soapValue2cppValue : Effectue la conversion du type XML au type C++.
- convert_cppValue2soapValue : Effectue la conversion du type C++ au type XML.

La classe concrète *String* (cf. Figure 3-14) hérite bien sûr de la classe *PrimitiveType* et implémente de ce fait, les méthodes virtuelles. Chaque classe héritant de *PrimitiveType* contient un attribut *cppValue* correspondant à la valeur de l'élément en C++.

```
class String: public PrimitiveType {

private:
    string cppValue;


    // Facettes associé à ce type
    static class Int length_f;
    static class Int minLength_f;
    static class Int maxLength_f;
    static list<String> enumeration_f;
    static class Int whiteSpace_f;
    static class RegularExpression pattern_f;

public:
    // Constructeur
    String(DOMNode* n)  throw (XMLTypeException);
    String(string c);

    // Destructeur
    ~String();

    // Accesseurs
    string getCppValue();
    void setCppValue(string s);

    // Méthodes et Opérateurs associé à ce type
    Boolean& contains(String s);
    String& operator=(const String& s);
    Boolean& operator==(const String& s);
    …


protected:
    void convert_soapValue2cppValue();
    void convert_cppValue2soapValue();
};
```

Figure 3-14: La classe concrète – String

La complexité des méthodes virtuelles de conversion de type varie énormément suivant la discordance des types qui existe entre les champs *soapValue* et *cppValue* (pour le type String, les types *cppValue* et *soapValue* étant les mêmes, les méthodes de conversion, sont dans ce

cas une simple affectation. Pour d'autres types tels que les dates grégoriennes, cette conversion devient beaucoup plus complexe).

Chaque type concret possède également ses propres facettes et ses accesseurs ainsi que des méthodes spécifiques (par exemple : « *contains* » pour les types « string ») ou des opérateurs (par exemple, l'affectation : =) associés à ce type. Au travers la manipulation des types DeWeL, l'utilisateur ne peut accéder qu'à un sous ensemble de ces opérations vérifiées. Aucune opération DeWeL ne lui permet par exemple d'appeler le destructeur de l'objet, puisque la suppression d'un nœud est totalement interdite.

Comme on peut le voir sur la Figure 3-14, le type String contient un second constructeur ayant un paramètre du même type que l'attribut cppValue. Ce constructeur est utilisé dans la génération de code pour les variables temporaires propres aux connecteurs. Ces variables n'ont bien sûr pas de nœud XML attribué (node == NULL).

La TypeStructure contient également les nouveaux types spécifiques apparaissant dans le schéma XML du service. En effet, lorsque des schémas sont définis dans un Service Web, cela signifie que le prestataire du service utilise des types simples redéfinis (SimpleType) ou crée des structures complexes (ComplexType). Il est donc indispensable de pouvoir générer ces nouveaux types pour pouvoir contrôler par la suite, la manipulation de ces types dans les pré et post traitements. Ceci est détaillé dans les sections suivantes.

3.5.2.1 Génération des types simples

Les types simples définis dans les schémas ne sont en fait que des types primitifs restreints dans lesquels certaines valeurs contenues dans les facettes ont été préfixées. La première vérification à effectuer, à l'intérieur de la génération d'un type simple C++ à partir du nœud XML correspondant, est de s'assurer que la valeur récupérée à l'exécution est cohérente par rapport aux facettes prédéfinies.

De récents travaux [104] ont montré que des outils largement utilisés dans la communauté des Services Web tel que JAX-RPC 1.1 étaient fondamentalement défectueux pour traduire des données XML en objets natifs d'un langage de programmation usuel. Par exemple, un code postal représenté par un schéma XML peut être modélisé par une restriction sur le type « string » (cf. Figure 3-15).

```
<simpleType name= « UKPostCode »>
     <restriction base= « xsd :string »>
              <pattern value= « [A-Z]{2}\d \d[A-Z]{2} »/>
     </restriction>
</simpleType>
```

Figure 3-15: UKPostCode - Un exemple de type simple

Le résultat actuel de la conversion en utilisant JAX-RPC est une simple classe de type « String ». Toutes les informations de restriction sont perdues lors de la transformation du schéma XML en Java. D'autres erreurs sont également à déplorer, celles-ci sont répertoriées dans l'article [104].

Dans notre cas d'étude, nous sommes partis d'une connaissance approfondie des schémas XML pour générer des classes C++ qui soient au plus proche de l'expressivité qu'ils reflètent. Ainsi, l'élément XML « UKPostcode » génère la classe C++ présentée en Figure 3-16.

```
class UKPostcode: public String {

  private:
    // Facettes associé à ce type
    static class Int length_f;
    static class Int minLength_f;
    static class Int maxLength_f;
    static list<String> enumeration_f;
    static class Int whiteSpace_f;
    static class RegularExpression pattern_f;

  public:
    // Constructeur
    UKPostcode(DOMNode* n)  throw (XMLTypeException);
    UKPostcode(string c);
    // Opérateur
    Boolean& operator==(const UKPostCode& s);
    …
};

RegularExpression UKPostcode::pattern_f(« [A-Z]{2}\d \d[A-Z]{2} »);
```

Figure 3-16: Génération d'un type Simple

La dernière ligne de la Figure 3-16 prend en compte les informations restrictives contenues dans le schéma et instancie la facette appropriée. Ainsi, lors de la création d'une telle variable, le constructeur vérifie que l'élément lexical inséré en entrée vérifie bien les contraintes instaurées dans le schéma.

3.5.2.2 Génération des types complexes

La Figure 3-17 présente comment à partir d'un type complexe défini dans le Service Web de Google, le type « GoogleSearchResult » (cf. Figure 1-12), on peut générer la classe C++ correspondante.

Figure 3-17: Génération d'un type complexe

La classe générée contient, ici, plusieurs sous-eléments. Ces sous éléments sont des pointeurs vers des objets de type primitif ou complexe. A chaque fois que l'utilisateur utilise un de ces sous éléments à travers les types DeWeL, la génération de code produit les vérifications d'usage pour s'assurer que le chaînage pour accéder à ce sous élément n'est pas nul. Si c'est le cas, une exception SOAP est générée.

La génération des types simples et complexes fournit l'opérateur d'égalité (==). Cet opérateur est un élément important pour la mise en place de mécanismes de recouvrement à base de vote. Ces mécanismes sont automatiquement générés lors de la phase de compilation.

3.5.3 Compilation d'un programme DeWeL et Génération de code

Le processus de compilation a deux entrées: (i) le canevas du connecteur mis à jour par l'utilisateur avec la déclaration d'une stratégie de recouvrement et des pré-et-post actions de traitement, et (ii) la TypeStructure fournie par l'***Analyseur de Schéma***. Le processus de compilation vérifie la cohérence du code source de l'utilisateur avec les informations contenues dans la TypeStructure. La sortie de ce processus de compilation est un ensemble de fichiers C++ spécifiant les mécanismes du connecteur et les types spécifiques du service. Le ***Compilateur DeWeL*** que nous avons développé est en charge de cette étape. Cet outil génère également, à partir des pré-et-post traitements, le document WSDL associé au connecteur. Celui-ci s'appuie sur les éléments de base du document WSDL original du Service Web et rajoute des caractéristiques non-fonctionnelles. Un nouveau point d'accès du service est alors spécifié; il correspond au connecteur enregistré dans la plate-forme IWSD.

Cette phase de compilation consiste à générer les traitements définis par l'utilisateur. Le but est de traduire le programme DeWeL en un connecteur qui puisse être chargé et exécuté par la plate-forme. Pour expliquer le processus de génération de code et d'exécution du connecteur, nous allons nous servir de l'exemple de la Figure 3-18 présentant une opération de conversion de mesure pour un service de température (opération : *CelsiusToFarenheit*). Cette opération contient des pré et post traitements définis par l'utilisateur.

```
1- SpecificFaultToleranceConnector ITempConverterservice_Wrapped
2-       extend http://developerdays.com/cgi-bin/tempconverter.exe/wsdl/ITempConverter {

3-  int return_output    CelsiusToFarenheit(int temp) {
4-       Pre-Processing:
5-            if((temp < 0)&&(temp > 40)) return SOAPException("Error: Temperature out-of-bounds");
6-            return CelsiustoFarenheit;
7-       Post-Processing:
8-            if(return_output < temp) return SOAPException("Error: Invalid Temperature Result");
9-            return CelsiustoFarenheit;
10-  };
11- }
```

Figure 3-18: Programme DeWeL sur le service de la température

A partir de cet exemple, nous allons voir quelles fonctions sont générées par le compilateur DeWeL, comment elles sont chargées par la plate-forme et dans quel ordre elles sont exécutées.

3.5.3.1 Génération de la fonction « start_connector »

En premier lieu, le compilateur DeWeL génère la fonction « *start_connector* ». Cette fonction est le point d'accès aux différentes actions de tolérance aux fautes spécifiées par l'utilisateur dans le programme DeWeL du connecteur. La Figure 3-19 reprend l'exemple précédent et présente la fonction « start_connector » générée à partir du programme DeWeL.

Cette fonction prend en paramètre un objet de la classe « Processing » (cf. ligne 1 de la Figure 3-19). Cet objet contient, entre autres, la requête encore non traitée provenant du client. Il permet d'accéder également aux services internes de la plate-forme sous jacente (log, connexion à la base de données pour les diagnostics, mise en place de mécanismes de recouvrement, détection et signalement d'erreur) de façon sûre et contrôlée. L'accès direct à ces mécanismes serait compromettant pour la plate-forme. L'utilisateur aurait alors accès à de nombreuses ressources critiques.

Les assertions automatiquement générées dans la fonction « start_connector » consistent essentiellement à vérifier que le message reçu correspond bien à une opération définie dans le document WSDL (ligne 5-7 dans la traduction C). Si c'est le cas, on récupère les paramètres de la requête (ligne 10) pour pouvoir y effectuer les différentes vérifications dans la fonction associée (ligne 11) : la fonction « CelsiusToFarenheit0 ». Dans le cas contraire, une exception SOAP sera retournée. Cette fonction est indicée par un entier (ici 0) pour éviter la redéfinition d'opération de même nom. En effet, WSDL autorise la surcharge d'opérations alors que le C non.

Programme DeWeL:

```
1- SpecificFaultToleranceConnector ITempConverterservice_Wrapped
2-       extend http://developerdays.com/cgi-bin/tempconverter.exe/wsdl/ITempConverter {

3-  int return_output    CelsiusToFarenheit(int temp) {
4-       Pre-Processing:
5-            if((temp < 0)&&(temp > 40)) return SOAPException("Error: Temperature out-of-bounds");
6-            return CelsiustoFarenheit;
7-       Post-Processing:
8-            if(return_output < temp) return SOAPException("Error: Invalid Temperature Result");
9-            return CelsiustoFarenheit;
10-  };
11- }
```

Traduction C:

```
1- int start_connector(Processing* m)
2- {
3-   int result = -1;
4-   try{
5-     if(( m->getSoapRequest().getOperationName() == "CelsiustoFarenheit" )
6-                 &&(m->getSoapRequest().getElmt("temp") != NULL)
7-                 &&(m->getSoapRequest().getNumberOfParameters() == 1))
8-             {
9-                  Int* return_output;
10-                 Int* temp = new Int(soapRequest.getElmt("temp"));
11-                 return_output = CelsiusToFarenheit0(temp, m);
12-                 if( return_output != NULL ) result = 0;
13-                 else m->setSOAPException("No Valid Response\n");
14-            }
15-     else m->setSOAPException("No Valid Request\n");
16-   }
17-   catch (Exception& e) {
18-       m->setSOAPException(e.print());
19-       result = -1;
20-   }
21-   return result;
22- }
```

Figure 3-19: Le point d'accès du connecteur - la fonction "start_connector"

3.5.3.2 Génération du modèle d'exécution

Chaque fonction issue d'une opération WSDL (ici, « *CelsiusToFarenheit0* » présentée sur la Figure 3-20) contient cinq sous-fonctions définissant respectivement les assertions et actions de tolérance aux fautes insérées dans les traitements globaux (*genericProcessing*) et les traitements spécifiques du programme DeWeL (le *Pre-Processing*, le *Post-Processing*, la *CommunicationException* et le *ServiceException*). Le traitement « *RecoveryStrategy* » qui

permet seulement de déclarer le mode de recouvrement n'est pas représenté ici, car c'est lui qui met en œuvre le modèle d'exécution du connecteur. Lorsque aucun mode de recouvrement n'est défini, les fonctions, ci-dessus, sont appelées suivant le modèle d'exécution linéaire présenté dans la Figure 3-20.

Lorsque les traitements globaux et les prétraitements sont effectués (lignes 6 et 8), la méthode *getResponseToRequest* (ligne 10) est en charge d'envoyer la requête au service Web correspondant. La réponse est ensuite analysée (ligne 12) pour extraire le paramètre de sortie avant d'effectuer le post-traitement (ligne 14). Si une erreur se produit lors de l'envoie de la requête ou de l'analyse de la réponse, celle-ci est capturée par les différents mécanismes d'exception (ligne 16 à 33) pour y effectuer les traitements spécifiques « Communication-Exception » et « ServiceException » (lignes 20, 26 et 32) définis par l'utilisateur. Les lignes 34 à 38 permettent de capturer les erreurs provenant d'assertions défaillantes définies par l'utilisateur dans les pré et post traitements.

```
1- Int* CelsiusToFarenheit0(Int* temp, Processing* m)
2- {
3-   Int* return_output = NULL;
4-   try{
5-     // Exécution des traitements globaux
6-     genericProcessing(m);
7-     // Exécution du traitement spécifique: Pre-Proessing
8-     pre_processing_CelsiustoFarenheit0(temp, m);
9-     // Envoi de la requête
10-    SOAPParser* soapResponse = m->getResponseToRequest();
11-    // Analyse de la réponse
12-    return_output = new Int(soapRep.getElmt("return"));
13-    // Exécution du traitement spécifique: Post-Proessing
14-    post_processing_CelsiustoFarenheit0(temp, return_output, m);
15-  }
16-  catch(SOAPParseException& soapEx)
17-  {
18-        m->getCurrentAccesspoint().sendCommunicationExceptionToHealthMonitor();
19-        // Exécution du traitement spécifique: CommunicationException
20-        communicationException_CelsiustoFarenheit0(temp, m, new CommunicationException(-1,soapEx.getMessage()));
21-  }
22-  catch(CommunicationException& ce)
23-  {
24-        m->getCurrentAccesspoint().sendCommunicationExceptionToHealthMonitor();
25-        // Exécution du traitement spécifique: CommunicationException
26-        communicationException_CelsiustoFarenheit0(temp, m, &ce);
27-  }
28-  catch(ServiceException& se)
29-  {
30-        m->getCurrentAccesspoint().sendServiceExceptionToHealthMonitor();
31-        // Exécution du traitement spécifique: serviceException
32-        serviceException_CelsiustoFarenheit0(temp, m, &se);
33-  }
34-  catch(UserException& se)
35-  {
36-        m->getCurrentAccesspoint().sendUserExceptionToHealthMonitor();
37-        userException_CelsiustoFarenheit0(m, &se);
38-  }
39-  m->getCurrentAccesspoint().sendSuccessToHealthMonitor();
40-  return return_output;
41- }
```

Figure 3-20: La fonction "CelsiusToFarenheit0"

Les lignes 18, 24, 30, 36, 39 sont en charge d'envoyer des messages significatifs au moniteur de surveillance permettant de diagnostiquer le point d'accès courant et donc le connecteur lui-même. On peut voir ici un des points importants des connecteurs : être capable de monitorer les points d'accès impliqués dans une application orientée services même en l'absence de mécanisme de recouvrement et de pré et post traitement définis par l'utilisateur.

3.5.3.3 Génération des pré-et-post traitements

Programme DeWeL:

```
1- Post-Processing:
2-     foreach tns_Items it1 in body_output.Items {
3-         foreach tns_Item it2 in it1.Item {
4-             if(it2.ItemAttributes.Title.contains("drugs"))
5-             {
6-                 log("Invalid Response: " + ipTargetAddress + "  - " + date() + "\n" + it2);
7-                 raise SOAPException("Invalid Response");
8-             }
9-             if(it2.ItemAttributes.PriceRating > 15)
10-                it2.remove();
11-        }
12-        if(it1.size() > 5) it1.remove(5, it1.size());
13-    }
14-    return ItemSearch;
```

Traduction C:

```
1- void post_processing_ItemSearch0(tns_ItemSearch* body, tns_ItemSearchResponse* body_output, Processing* m)
2- {
3-  if(body_output.Items != NULL)  {
4-   // Ecriture du ForEach:
5-   list<tns_Items*>::iterator it1;
6-   for(it1= body_output.Items.begin(); it1 != body_output.Items.end(); ++it1)  {
7-    if((*it1).Item != NULL))  {
8-     // Ecriture du ForEach:
9-     list<tns_Item*>::iterator it2;
10-    for(it2= (*it1.Item())).begin(); it2 != (*it1).Item.end(); ++it2)  {
11-     if((*it2).ItemAttributes).node != NULL)&&((*it2).ItemAttributes.Title.node != NULL))  {
12-      if((*it2).ItemAttributes.Title.contains( String("drugs")))  {
13-       m->log("Invalid Response: " + m->getIpTargetAddress() + "  - "  + m->getDate() + "\n" + (*it2).toString());
14-       m->setUserException(true);
15-       m->raiseException( String("Invalid Response"));
16-      }
17-     }
18-     else {
19-       m->setUserException(true);
20-       throw PostProcessingException(-1,"Invalid Instruction");
21-     }
22-     if((*it2).ItemAttributes.node != NULL)&&((*it2).ItemAttributes.PriceRating.node != NULL))  {
23-      if((*it2).ItemAttributes.PriceRating ==  NonNegativeInteger(15))  {
24-       if((*it2).getMinOccurs() == Int(0))  {
25-        (*it2).autodelete();
26-       }
27-       else {
28-        m->setUserException(true);
29-        throw PostProcessingException(-1,"Impossible to delete this element: it2");
30-       };
31-      }
32-     }
33-     else {
34-       m->setUserException(true);
35-       throw PostProcessingException(-1,"Invalid Instruction");
36-     }
37-    } // Fin du For
38-   }
39-   else {
40-     m->setUserException(true);
41-     throw PostProcessingException(-1,"Invalid Instruction");
42-   }
43-   if((*it1).Item != NULL)  {
44-    if(sizeLIST((list<Type*>*)(*it1).Item) >  Int(5))  {
45-     removeLIST((list<Type*>*)(*it1).Item,  Int(5),sizeLIST((list<Type*>*)(*it1).Item));
46-    }
47-   }
48-   else {
49-     m->setUserException(true);
50-     throw PostProcessingException(-1,"Invalid Instruction");
51-   }
52-  } // Fin du For
53- }
54- else {
55-     m->setUserException(true);
56-     throw PostProcessingException(-1,"Invalid Instruction");
57- }
58- return;
59- }
```

Figure 3-21: Traduction du post-traitement

La Figure 3-21 représente la traduction en C du post traitement du programme DeWeL de Amazon montré en Figure 3-3. Le nombre de lignes de code C générées est, dans ce cas, environ quatre fois plus important que son équivalent en DeWeL. Les lignes en italique illustrent les vérifications effectuées automatiquement par DeWeL. Ces vérifications permettent de s'assurer que l'assertion programmée par l'utilisateur ne provoquera pas d'erreur à l'exécution.

3.5.4 Génération de la librairie dynamique

Cette étape est l'étape finale du processus de génération de code dans lequel un compilateur C++ conventionnel est employé pour produire le connecteur comme une librairie dynamique à partir des fichiers générés. En effet, à la fin de l'étape précédente, plusieurs fichiers ont pu être générés :

- ***Connector (.hpp et .cpp)*** : Ce sont les fichiers C++ contenant les actions du connecteur définies par l'utilisateur.
- ***SpecificType (.hpp et .cpp)*** : Ces fichiers C++ contiennent les types spécifiques générés à partir du schéma XML et manipulables dans le langage DeWeL au travers des types dit « construits ».
- ***WSDL Connector (.wsdl)*** : Ce fichier présente la description de l'interface du connecteur.

Les fichiers associés au connecteur et aux types spécifiques utilisés permettent de créer cette librairie. Celle-ci est stockée dans la plate-forme IWSD. Cette étape est effectuée par le compilateur standard GNU C/C++. C'est une étape de vérification supplémentaire puisqu'elle permet de s'assurer que la génération de code n'a pas produit d'erreur à la compilation. A l'exécution, le serveur d'exécution chargera cette librairie dynamique pour y effectuer les traitements spécifiés par l'utilisateur.

3.6 Récapitulatif

DeWeL a été conçu afin de permettre à un utilisateur de pouvoir concevoir un connecteur robuste. Il offre les abstractions et notations appropriées capables de réaliser des actions de tolérance aux fautes efficaces. Celles-ci sont réalisées en utilisant les facettes, variables et fonctionnalités prédéfinies dans DeWeL. A l'aide des différentes instructions, il permet de réaliser des assertions complexes. Cependant, DeWeL impose de nombreuses contraintes. Elles sont instanciées dans l'unique but de réaliser des vérifications statiques efficaces sur des propriétés critiques essentielles et représentent des restrictions reconnues au niveau des standards de développement de logiciels critiques qui sont appliqués dans des domaines tel que l'avionique ou le ferroviaire.

Le processus de génération de code permet également de mettre en œuvre des vérifications dynamiques importantes. Pour cela, une analyse rigoureuse des types spécifiques du service est effectuée. Cette analyse permet d'insérer et de manipuler dans le langage de nouveaux types définis à partir du schéma XML du service. Elle permet d'instaurer, entre autres, des opérateurs et des méthodes de base sur chacun de ces types. La création de l'opérateur d'égalité pour chacun d'eux est à l'origine des mécanismes de vote que peut configurer l'utilisateur.

La création d'une architecture orientée services est un processus complexe. Le chapitre 1 nous a permis de constater que celle-ci s'appuyait sur de nombreux composants sur étagères ayant

chacun une robustesse plus ou moins affirmée. Des défaillances peuvent ainsi survenir sur différentes couches logicielles. L'ajout de composants additionnels dans de telles architectures ne peut que faire augmenter ce taux de défaillance.

L'intérêt de DeWeL, ici, est de minimiser les sources d'erreur lors du développement des connecteurs. Ces connecteurs, chargés de détecter et d'effectuer le recouvrement de certaines erreurs liés à une application donnée, constituent, en fait, un point unique de défaillance qu'il faut traiter par des techniques de prévention de fautes (langage DeWeL) et de tolérance (mécanisme de réplication de IWSD).

DeWeL prend donc une place capitale dans une plate-forme sûre de fonctionnement capable de créer des connecteurs spécifiques de tolérance aux fautes. Il permet à l'utilisateur de programmer des assertions spécifiques en minimisant les risques de fautes du logiciel. L'utilisateur peut ainsi à sa guise paramétrer son connecteur et utiliser des fonctions prédéfinies du langage capables de lui fournir des informations opérationnelles capitales sur le service ciblé et ainsi adapter ses actions de tolérances aux fautes en fonction des données recueillies. Le connecteur créé par chaque utilisateur est ainsi exécuté sur le serveur d'exécution pour une application donnée. La sûreté de fonctionnement du serveur d'exécution repose sur des techniques classiques appliquées à l'implémentation de la plate-forme IWSD sous-jacente.

4 Le Connecteur en action

Ce chapitre détaille les différents mécanismes de tolérance aux fautes que peuvent réaliser les connecteurs. DeWeL fournit un support langage efficace pour programmer de tels composants. La plate-forme IWSD, quant à elle, fournit le cadre d'exécution de ces connecteurs spécifiques de tolérance aux fautes. Ceux-ci sont déclarés sous forme de contrats WSDL et mis en libre service dans un annuaire UDDI. L'objectif de ce chapitre consiste à détailler, dans un premier temps, le fonctionnement opérationnel d'un connecteur, c'est-à-dire, de présenter les différents mécanismes qu'il déclenche dans la plate-forme IWSD, et ensuite, de présenter le contrat WSDL du connecteur généré par le compilateur DeWeL.

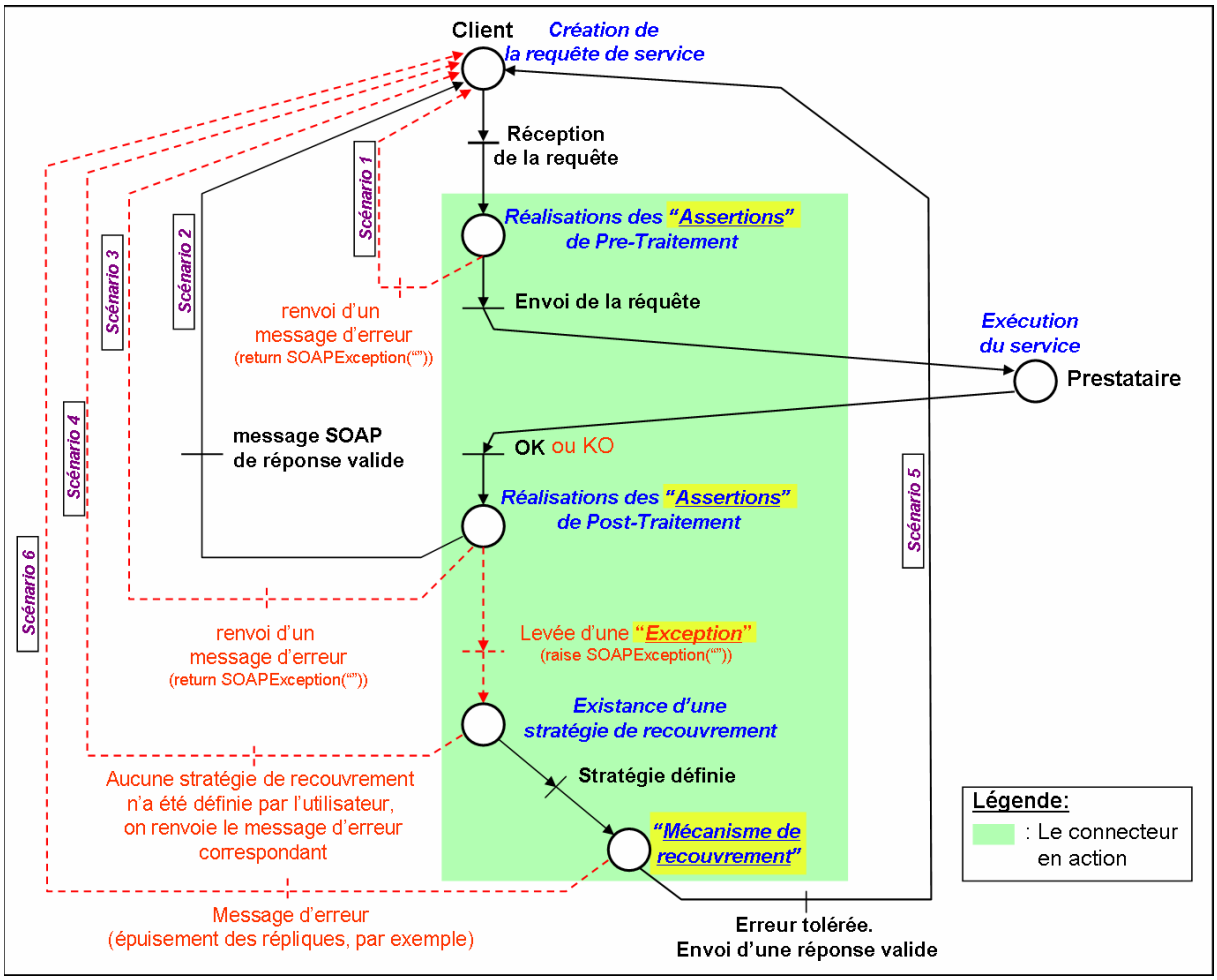

Figure 4-1: Le connecteur en action

La Figure 4-1 permet d'analyser, au niveau macroscopique, les différentes actions réalisées par le connecteur. Celles-ci sont composées d'assertions, de levées d'exception et de déclenchements de mécanismes de recouvrement. Selon les données reçues par le connecteur et les assertions définies par l'utilisateur, plusieurs cas de figures sont possibles :

- ***Scénario 1 :*** La requête reçue par le connecteur n'est pas validée par les assertions de pré-traitement de l'utilisateur. Un message d'erreur est, dans ce cas, retourné au client.

- ***Scénario 2 :*** La réponse du prestataire satisfait les assertions de post-traitement. Le message est retourné au client, ce qui correspond au cas nominal.
- ***Scénario 3 :*** Les assertions de post-traitement définies par l'utilisateur préconisent de retourner l'erreur au client. La stratégie de recouvrement, si elle est précisée, est, dans ce cas, annulée, car elle peut être considérée comme non appropriée.
- ***Scénario 4 :*** Les assertions de post-traitement définies par l'utilisateur lèvent une exception mais aucune stratégie de recouvrement n'a été précisée par l'utilisateur. Un message d'erreur correspondant à l'exception est retourné au client.
- ***Scénario 5 :*** Les assertions de post-traitement définies par l'utilisateur lèvent une exception qui déclenche la stratégie de recouvrement de l'utilisateur. Celle-ci permet de tolérer l'erreur et retourne un message SOAP valide au client.
- ***Scénario 6 :*** Les assertions de post-traitement définies par l'utilisateur lèvent une exception qui déclenche la stratégie de recouvrement de l'utilisateur. Celle-ci ne permet pas de tolérer l'erreur. Un message d'erreur est retourné au client.

Cette figure permet de présenter globalement le mode de fonctionnement du connecteur. Une présentation plus fine de chaque mécanisme (détections par assertions, levées d'exception et stratégies de recouvrement) est proposée dans la section suivante.

4.1 Les Mécanismes

Dans cette section, nous allons analyser les différents mécanismes actionnés par le connecteur à partir de la réception de la requête jusqu'au renvoi de la réponse par la plate-forme IWSD. Nous allons détailler ici les différents modèles d'exécution réalisés en fonction du mode de recouvrement sélectionné. Ces modèles d'exécution représentent la séquence d'actions effectuées par le connecteur pendant son activation (vérification, notification, envoi et réception de messages, sauvegarde ou restauration de l'état, …etc.).

4.1.1 Assertions

Le connecteur prévoit deux types d'assertions: explicites et implicites. Les assertions explicites sont définies par l'utilisateur à l'aide de DeWeL et sont spécifiques à chaque connecteur. Les assertions implicites sont automatiquement générées et exécutées par le connecteur.

Les ***assertions explicites*** correspondent à des mécanismes de détection spécifiques décrits par l'utilisateur dans un programme DeWeL. Ce type d'assertion peut être classé en deux catégories :

- ***les assertions globales :*** Ces assertions sont valides pour toutes les opérations du service. Elles sont définies dans les traitements globaux du programme DeWeL. Elles permettent essentiellement de faire des vérifications ou des redéfinitions sur les facettes des types.
- ***les assertions spécifiques :*** Ces assertions sont propres à chaque opération du service. Elles peuvent contrôler la validité d'une valeur donnée. Elles s'effectuent sur les paramètres d'entrée/sortie de l'opération. La violation de telles assertions signale une erreur au moyen d'exceptions ou déclenche les mécanismes appropriés de recouvrement si elles sont dans un post-traitement (voir la section 4.1.3).

Les ***assertions implicites*** sont faites par défaut quand un message (requête ou réponse) est examiné par le connecteur. Ces assertions permettent au client ou au fournisseur de se protéger contre les messages SOAP mal formés ou corrompus. La vérification est réalisée en traduisant les données reçues dans le message selon les types définis dans le schéma XML du document WSDL du service.

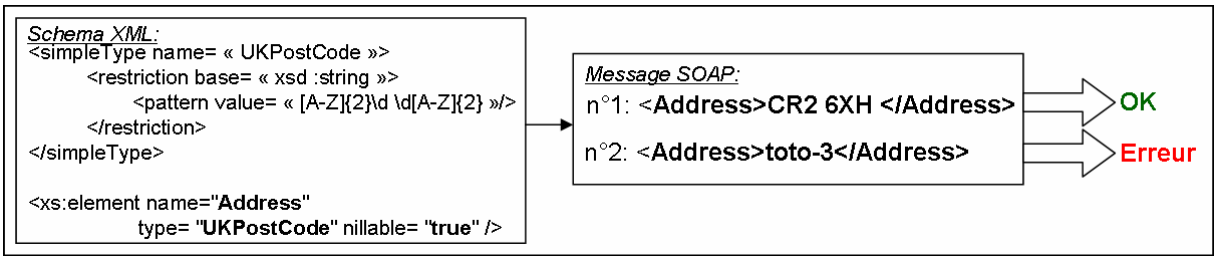

Figure 4-2: Assertions implicites

La Figure 4-2 reprend l'exemple du type « *UKPostCode* » défini dans le chapitre précédent et présente l'analyse de deux messages SOAP par la plate-forme. Le second cas lève une erreur implicite puisque l'analyse de l'élément est incorrecte car la valeur du paramètre ne respecte pas le format du type. On peut constater ici que la librairie JAX-RPC 1.1 est dans l'incapacité de détecter cette erreur implicite puisque cette dernière convertit l'adresse en une simple chaîne de caractères sans aucune restriction.

4.1.2 Exception

Les exceptions spécifiques de haut niveau sont fortement liées à l'application. Elles peuvent être également implicites ou explicites. Par exemple, une exception peut être retournée quand une requête SOAP ne peut pas être analysée par le parseur (exception implicite). Une assertion définie par un utilisateur dans les pré-et-post actions mène le plus souvent, dans le cas où elle est erronée, à une levée d'exception (exception explicite).

4.1.2.1 La Génération d'une exception

L'utilisateur peut définir et lever une exception quand une assertion est fausse. Cette levée d'exception est spécifiée par l'utilisateur dans DeWeL grâce à l'instruction « raise SOAPException(«Error_Message») ». Lorsque cela n'est pas précisé, l'exception retournée correspond à la version SOAP utilisée dans la requête:

- xmlns :soap-env= « http://www.w3.org/2003/05/soap-envelope » : Exception de type SOAP 1.2
- xmlns :soap-env= « http://schemas.xmlsoap.org/soap/envelope/ » : Exception de type SOAP 1.1

En effet, les plus grands changements entre les versions SOAP 1.1 et SOAP 1.2 sont réalisés au niveau du format des messages d'erreur échangés. Le connecteur est donc contraint de générer une exception qui soit compréhensible pour l'application cliente (exception de type SOAP 1.1 ou SOAP 1.2). Cette propriété confère une utilité supplémentaire au connecteur. Un prestataire peut, par exemple, utiliser un connecteur, pour traduire ses propres exceptions codées en SOAP 1.2 en SOAP 1.1 et ainsi permettre à des clients spécifiques n'ayant pas la dernière version du parseur de pouvoir utiliser son service.

4.1.2.2 La Capture d'une exception

Dans le connecteur, la capture d'une exception est implicite mais le code pour traiter l'exception peut être complexe et est donc défini par l'utilisateur. Un traitement très simple d'exception peut être la notification de celle-ci. L'utilisateur peut ainsi récupérer et traiter deux types d'exceptions: les exceptions de communication et les exceptions de service.

Une **exception de communication** est retournée si le service cible n'a pas répondu à la requête envoyée ou a répondu avec un message non SOAP. Ce type d'exception peut être provoqué de plusieurs façons: le canal de communication a été coupé (broken pipe), la fenêtre de temps spécifiée par l'utilisateur est dépassée (timeOut), ou une erreur dans le protocole de transport Internet utilisé est survenue. Cette dernière erreur est intimement liée au protocole Internet utilisé pour transporter les messages SOAP. Dans le cadre de notre expérimentation, seul le protocole HTTP a été pris en compte. Les types d'erreurs retournées par ce protocole sont listés dans la RFC 2616 [105]. Parmi elles, on peut citer les erreurs suivantes: Forbidden (403), Not Found (404), Internal Server Error (500), Bad Gateway (502), Service Unavailable (503), Gateway Time-out (504).

Les exceptions de communication sont les plus graves qui puissent survenir lors du traitement de la requête car pour certaines d'entre elles, on ne peut pas savoir si la requête a été traitée par le prestataire ou non. Dans le cas où une réplication passive serait activée, il s'avèrerait néfaste de passer à la réplique suivante. Sur ce genre de problème, la philosophie des Services Web préconise la mise en œuvre de transaction compensatoire en effectuant l'opération inverse pour annuler la transaction courante. Cette problématique est prise en charge par des protocoles de plus haut niveau (WS-Transaction, WS-Coordination) inclus dans BPEL4WS pour la mise en œuvre de processus métier. Si ce genre de problème survient sur des prestations avec états, deux possibilités sont envisageables pour le connecteur :

- Soit il effectue lui-même la transaction inverse avant de passer à la réplique secondaire.
- Soit il prévient le client ou le coordinateur du processus métier afin qu'il puisse effectuer les transactions compensatoires sur les services défaillants.

La variable « **ce** » de type **CommunicationException,** présenté dans les différents exemples de programme DeWeL, permet de manipuler cette exception. Elle contient plusieurs champs, dont le code de l'erreur et le message d'erreur s'il y en a. Lorsqu'on retourne une variable de ce type par l'instruction « return », une exception SOAP correspondant à l'erreur est automatiquement générée. S'il le souhaite, l'utilisateur a la possibilité de retourner une exception différente, qui peut par la suite être récupérée et traitée de manière différente par le client.

Une **exception de service** est retournée quand le service visé n'a pas pu réaliser la prestation et renvoie une erreur comme message SOAP. A travers, la variable DeWeL « **se** » de type **ServiceException**, l'utilisateur peut vérifier et contrôler l'exception et suivant le message reçu, retourner une exception plus spécifique et pertinente pour l'application cliente. Si une exception de ce type se produit vis-à-vis d'une réplique, l'envoi de l'exception est annulé et on passe à la réplique suivante.

Il existe également un autre type d'exception non-manipulable par l'utilisateur. Ce sont les **ProcessingException**. Ces exceptions peuvent se produire suite à divers problèmes : l'utilisateur n'a pas pu être identifié lors de l'utilisation d'un connecteur privé, le connecteur précisé en en-tête de la requête n'existe pas, la requête reçue ne vérifie pas le schéma XML

correspondant et provoque une erreur dans le parseur. Dans l'implémentation actuelle, ces différentes sortes d'erreurs ne sont pas remontées à l'utilisateur par le programme DeWeL. Elles sont directement traitées en interne et génèrent une exception SOAP correspondant au problème survenu.

4.1.3 Les stratégies de recouvrement

Fonctions DeWeL	Stratégie de recouvrement	Rôle du Connecteur vs. Stratégie de recouvrement	Rôle du fournisseur	Rôle du client
1) BasicReplication	Réplication Passive sans gestion d'état	Basculement sur une autre réplique	Gestion du recouvrement d'état	
2) StatefulReplication	Réplication Passive avec gestion d'état du Service Web en utilisant des fonctions de gestion d'état définies par le fournisseur.	Point de reprise en utilisant des fonctions de gestion d'état définies par l'utilisateur Basculement sur une autre réplique	Fournit les fonctions de gestion d'état	
3) LogBasedReplication	Réplication Passive avec gestion d'état en utilisant des mécanismes de session (log et répétition)	Enregistrement de toutes les requêtes durant la session. Re-exécution des requêtes enregistrées sur une autre réplique		Invocation du démarrage et de la fin de la session
4) ActiveReplication	Réplication Active sans vote	Envoi multicast des requêtes vers un ensemble de répliques. Sélection de la première réponse acceptable		
5) VotingReplication	Réplication Active avec vote	Envoi multicast des requêtes vers un ensemble de répliques. Effectue le vote ou la procédure de décision parmi les multiples réponses		

Figure 4-3: Les modes de recouvrement

Nous avons mis en œuvre deux types de techniques pour traiter, dans une certaine mesure, les fautes de conception du logiciel au niveau des Services Web :

1. éviter que la défaillance d'une tâche n'entraîne la défaillance de tout le système (approche de prévention).
2. assurer la continuité de service en utilisant des techniques de réplication diversifiée (approche de tolérance).

Dans le premier cas, on cherche à détecter au plus tôt une tâche erronée et à provoquer son arrêt afin d'éviter la propagation de l'erreur, ou des erreurs ; cette approche est souvent qualifiée de « fail-fast » [106]. La détection d'erreur est, ici, assurée par des assertions exécutables définies par l'utilisateur en DeWeL, portant sur les données traitées.

Le deuxième cas suppose que l'on dispose d'au moins un autre composant à même d'assurer la tâche conçue et réalisée séparément à partir de la même spécification : il s'agit de la diversification fonctionnelle. L'Internet offre de ce point de vue de nouvelles opportunités

pour la mise en œuvre de ces types de mécanismes notamment à travers l'utilisation de répliques équivalentes dans les Services Web abstraits (cf. Chapitre 2).

Le traitement spécifique de DeWeL intitulé « RecoveryStrategy » permet à l'utilisateur de choisir une stratégie appropriée de recouvrement pour un service ciblé. Il est clair que certains mécanismes mis en oeuvre ne peuvent pas être totalement fournis. Ces mécanismes sont donc partiellement intégrés, essentiellement en raison des problèmes liés à la gestion d'état du Service Web. La stratégie finale de tolérance aux fautes à employer est donc le résultat de la collaboration entre le client, les connecteurs s'exécutant sur la plate-forme IWSD, et le fournisseur si possible.

Les stratégies de recouvrement sont définies en utilisant les fonctions privées données dans la Figure 4-3. Pour chaque fonction DeWeL, cette table récapitule les principes des stratégies disponibles et le rôle de chaque participant (connecteur, fournisseur et client). Le clonage d'une réplique n'est pas considéré ici.

La mise en place de modes de recouvrement nécessite d'instaurer divers mécanismes internes dans la plate-forme qui puissent être configurables et paramétrables par l'utilisateur. Ainsi, l'activation d'un mode de recouvrement passe par la sélection et la configuration de ces différents mécanismes.

4.1.3.1 La fonction : BasicReplication

Ce type de recouvrement est très utile pour les opérations sans état. Pour cela, différentes implémentations de répliques identiques ou équivalentes peuvent être utilisées. La *BasicReplication* fournit, dans ce cas, les mécanismes de redirection sur les autres répliques, la détection d'erreurs étant faite au moyen des assertions et des captures d'exceptions. Son modèle d'exécution a déjà été présenté dans la Figure 2-11. Dans ce modèle, une liste de répliques est créée. On passe en revue la liste des répliques en envoyant la requête successivement sur chacune jusqu'à obtention d'une réponse valide ou jusqu'à l'épuisement de la liste des répliques. A chaque nouvelle requête, la liste des répliques disponibles est parcourue dans le même ordre. On suppose que les fautes sont alors transitoires.

Un rapport est envoyé au moniteur de surveillance pour mettre à jour les informations sur les différents points d'accès contactés.

Bien que ce type de recouvrement soit essentiellement employé pour des services sans état, il peut, sous certaines conditions, être également utilisé dans les opérations à état. Dans ce cas, le fournisseur est responsable de la gestion de celui-ci (voir la ligne 1 sur la Figure 4-3). Le temps de basculement d'une réplique à l'autre peut être défini par le fournisseur comme paramètre d'entrée de la fonction « BasicReplication ».

4.1.3.2 La fonction : StatefulReplication

Le *StatefulReplication* exécute le point de reprise en utilisant des fonctions de gestion d'état (*Save_State/Restore_State*) développées par le fournisseur. Ce genre d'approche a été employé dans les systèmes orientés objets tolérants aux fautes: une classe abstraite *StateManager* est fournie au prestataire qui est responsable de l'implémentation de deux méthodes virtuelles *Save_State* et *Restore_State* (voir. [107]). Dans notre contexte, ces opérations (*Save_State* et *Restore_State*) doivent être accessibles uniquement par la plate-forme IWSD pour être déclenchées au moment approprié. Le contrat WSDL du service de base peut être étendu avec la signature de ces dernières opérations de gestion d'état qui sont implémentées par le fournisseur. Clairement, de telles opérations exigent une authentification.

La plate-forme IWSD peut, à distance, contrôler l'état des répliques de Service Web. Un exemple de ce genre de contrat peut être consulté en annexe A.5.

Lorsqu'une erreur sur le primaire est détectée, celui-ci est rendu inactif le temps de l'activité du connecteur. La réplique suivante est déclarée primaire. Lors d'une prochaine exécution, l'ancien primaire devient une réplique secondaire.

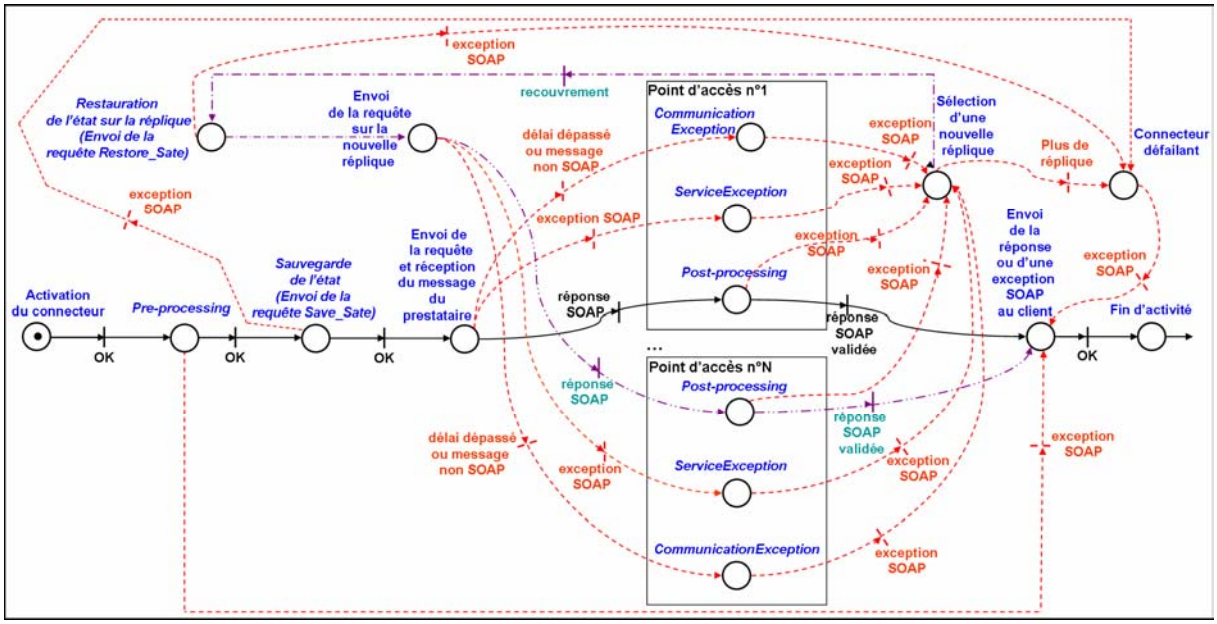

Figure 4-4: Le modèle d'exécution de la StatefulReplication

Le modèle d'exécution de la Figure 4-4 gère le flux d'instructions lorsque les prestations de services ont un état. Dans ce cas, lorsqu'on envoie une requête vers un prestataire plusieurs cas peuvent se produire :

- Si on récupère une réponse d'erreur SOAP (ServiceException), cela signifie que le fournisseur n'a pas pu fournir la prestation en question (destination inexistante sur cette compagnie). Dans ce cas, il est donc tout à fait possible de pouvoir rejouer la requête sur la réplique suivante en rechargeant l'état au préalable si nécessaire.
- Si une erreur de communication se produit, une analyse de cette erreur doit être effectuée pour s'assurer que la prestation a été faite ou non. Dans le cas où un message d'erreur est récupéré, par exemple, un message d'erreur du protocole de transport de type : Not Found (404), Forbidden (403), Service Unvailable (503), on se retrouve dans le cas précédent où la requête n'a pas pu être effectuée et donc, on a la possibilité de pouvoir rejouer directement la requête sur une autre réplique. Dans le cas où l'opération a été effectuée, les mécanismes existants tels que WS-AtomicTransaction [108] et WS-Coordination [109] peuvent être utilisés pour annuler l'opération faite sur la réplique avant d'exécuter la suivante.
- Si le message SOAP reçu est valide, on effectue le post-traitement. Deux alternatives peuvent survenir soit la réponse est validée par le post-traitement et retournée au client, soit la réponse génère une exception SOAP. Là encore, une transaction compensatoire doit être effectuée. Celle-ci peut être prise en charge par le connecteur ou le client.

Les modèles d'exécution que nous présentons ici, donnent les briques de base pour mettre en place les mécanismes de recouvrement complexes. Cependant, la plupart de ces modèles peuvent s'enrichir en gérant les transactions compensatoires avant de rejouer la requête sur une autre réplique suite à un post-traitement invalide.

A l'heure actuelle, lorsque le connecteur a besoin d'interroger une réplique autre que le primaire pour obtenir une réponse valide, celui-ci insère dans l'entête de la réponse fournie au client des informations concernant les points d'accès indisponibles mais également des informations sur la réplique à l'origine de la réponse. Ces informations peuvent être utilisées par le client pour défaire le traitement si nécessaire ou contrôler l'indisponibilité d'une réplique.

4.1.3.3 La fonction : LogBasedReplication

La solution *LogBasedReplication* se fonde sur des fonctions de gestion de session fournies par le connecteur et employées par le client (*start_session* et *end_session*). Pendant une session définie par le client (c'est-à-dire, encadrée par les opérations « *start_session* » et « *end_session* »), le connecteur enregistre toutes les opérations en cours. Quand la réplique primaire défaille, toutes les requêtes sauvées peuvent être renvoyées à une réplique de secours et être rejouées pour atteindre un état cohérent. Cette approche permet de défaire les opérations effectuées grâce à l'information enregistrée dans les logs. Les mécanismes WS-AtomicTransaction [108] et WS-Coordination [109] peuvent être employés dans ce but.

La mise en place d'un tel mécanisme de recouvrement impose au client de définir des sessions en utilisant les opérations « *start_session* » et « *end_session* » qui sont ajoutées au contrat WSDL du connecteur. L'en-tête de chaque requête SOAP contient, ainsi, l'identifiant de la session en cours.

Ce mécanisme suppose également que les répliques utilisées sont cohérentes au début de la mise en place du connecteur. Contrairement à la *StatefulReplication*, lorsque le primaire défaille, celui-ci est désactivé tant que la réplique n'a pas été réinitialisée. Ceci permet de conserver la cohérence des répliques. En effet, les requêtes ne peuvent être rejouées que sur des répliques maintenues dans leur état initial.

Enfin, ces mécanismes peuvent être couplés avec les opérations de gestion d'état vues précédemment. Ces dernières peuvent être employées afin de sauver périodiquement l'état du primaire toutes les *n* requêtes lorsqu'une session est très longue et initialiser ainsi une réplique à un état cohérent.

4.1.3.4 La fonction : ActiveReplication

Le modèle d'exécution de l'active réplication a été présenté dans la Figure 2-12. Celui-ci concerne les opérations sans état.

Dans ce modèle, un thread est créé pour chaque point d'accès. Chaque thread est en charge de traduire la requête pour la faire correspondre au point d'accès de la réplique avant de l'envoyer. Sur réception de la réponse, le thread insère le message correspondant dans une liste de résultats connus de la plateforme. Suivant le mode de recouvrement, le connecteur analyse toutes les réponses de cette liste ou ne renvoie que la première réponse qui satisfait le post-traitement.

Dans les deux solutions de réplication active (voir les lignes 4 et 5 de la Figure 4-3), l'objectif principal à assurer est de garantir la cohérence des répliques, lorsque les opérations ont un

état. Pour cela, le contrôle de l'ordonnancement total des requêtes peut être fait par le serveur d'exécution à l'aide d'algorithme prédéfinis [67] (voir Figure 4-5). La délivrance du message peut être assurée par des protocoles tels que WS-Reliability [68]. Cette solution impose que seul le connecteur soit en mesure de contacter les répliques du service, ce qui est une hypothèse très forte.

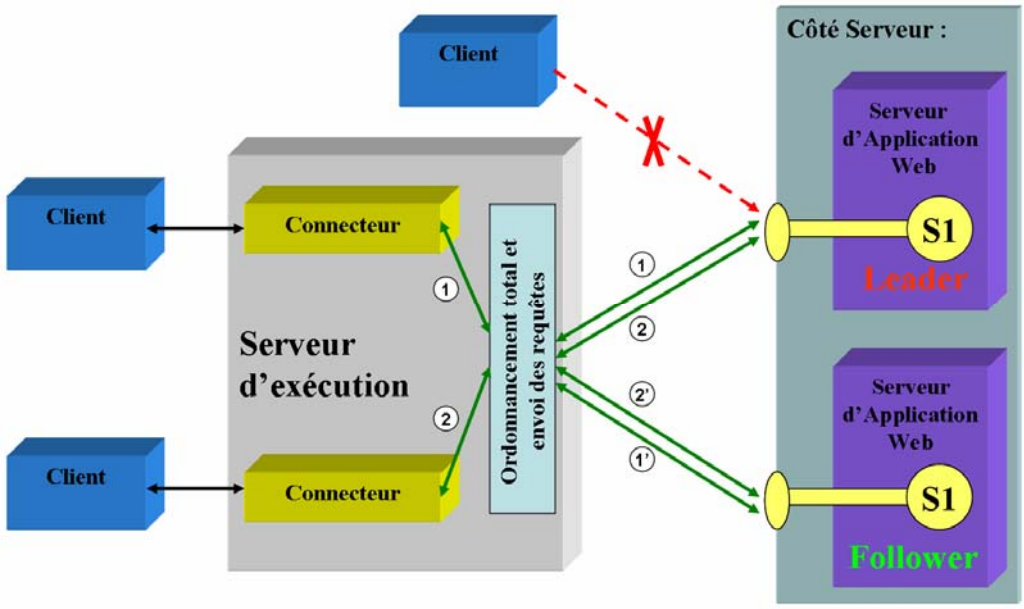

Figure 4-5: Ordonnancement total des requêtes

Le modèle d'exécution de la réplication active sans état diffère légèrement de la réplication active avec état. Dans cette dernière, lorsqu'une réplique défaille, celle-ci est supprimée de la liste des répliques. Ceci a pour but de préserver la cohérence des répliques utilisées. Cette particularité est également valable pour le « VotingReplication" présenté dans la section suivante.

4.1.3.5 La fonction : VotingReplication

Le modèle d'exécution de la fonction « VotingReplication » présenté sur la Figure 4-6 concerne essentiellement des services fournissant des prestations tels que des calculs scientifiques lourds, et critiques sur lesquels on souhaite tolérer des défaillances en valeur.

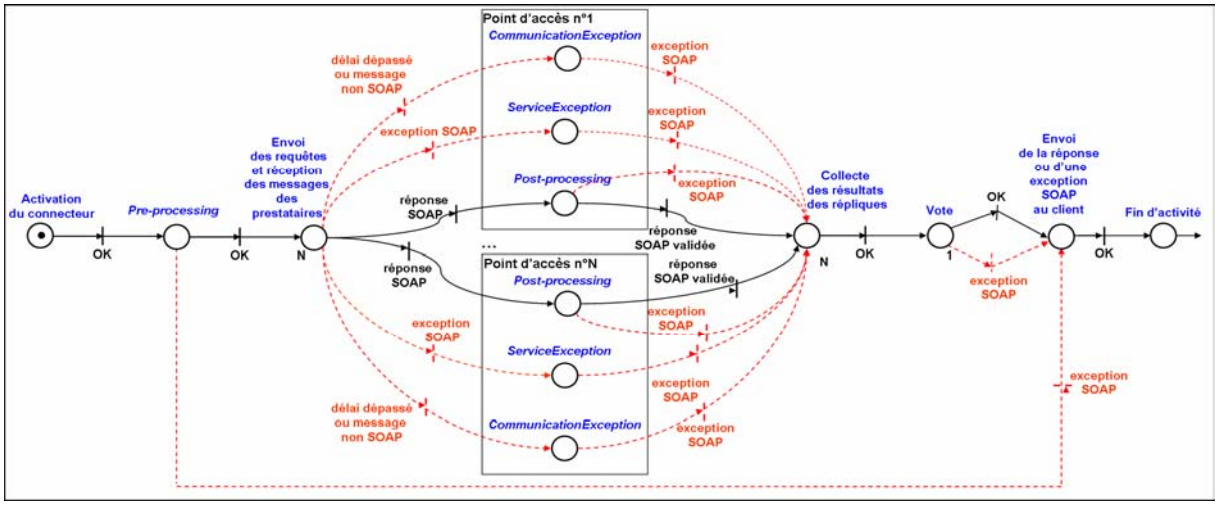

Figure 4-6: Le modèle d'exécution de la VotingReplication

Dans ce cas, on envoie la requête sur chaque réplique et on attend la fin de leur exécution. La phase suivante consiste à ne prendre que les requêtes post-traitées valides et les soumettre au voteur. Comme on a pu le voir dans le chapitre 3, une version de ce voteur peut-être automatiquement générée par DeWeL, spécifiquement, pour chaque opération de services. Comme les types primitifs de la plate-forme et les types complexes générés possèdent tous l'opérateur d'égalité (==), le voteur effectue l'égalité sur le paramètre de sortie retourné par chaque prestataire valide. Il a besoin d'au moins $2f+1$ requêtes post-traitées pour pouvoir tolérer f fautes. Si ce n'est pas le cas, le connecteur est défaillant et une exception SOAP est retournée au client.

A l'heure actuelle, seul le vote majoritaire est effectué. Cependant, différentes paramétrisations du vote peuvent être effectuées en DeWeL (la moyenne, la médiane, …etc) pour obtenir des voteurs plus adaptés aux besoins de l'utilisateur et de l'application.

4.2 L'interface du Connecteur

Nous venons de voir les différentes actions que pouvait mener le connecteur pendant son exécution, nous allons maintenant, analyser la manière dont il peut être décrit et référencé dans un contrat WSDL pour pouvoir être consulté, choisi et manipulé par un client.

Le document WSDL du connecteur permet à un client de générer un stub capable d'exploiter pleinement toutes les fonctionnalités qu'il peut proposer. Il contient toutes les opérations SOAP du contrat de service original. Pour chacune de ces opérations de base, une opération étendue est créée dans laquelle les paramètres supplémentaires définis par l'utilisateur dans DeWeL ainsi que des paramètres optionnels par défaut sont ajoutés (cf. Figure 4-9). Ces paramètres optionnels sont insérés pour authentifier un client en tant qu'utilisateur, pour configurer dynamiquement le serveur d'exécution (la fenêtre de temps pour les exceptions de communication, le nombre de tentatives de connexion à un Service Web, …etc) et pour ajouter des informations supplémentaires dans les réponses fournies au client (nombre de répliques contactées pour avoir la réponse, dernière répliques utilisées, …etc.). Le connecteur peut donc recevoir deux types de requêtes pour l'exécution d'une même opération : la requête originale définie par le prestataire et la requête étendue propre au connecteur définie par l'utilisateur de la plate-forme (cf. Figure 4-7). A l'exécution, le connecteur consomme ces paramètres additionnels pour fournir au prestataire une requête originale.

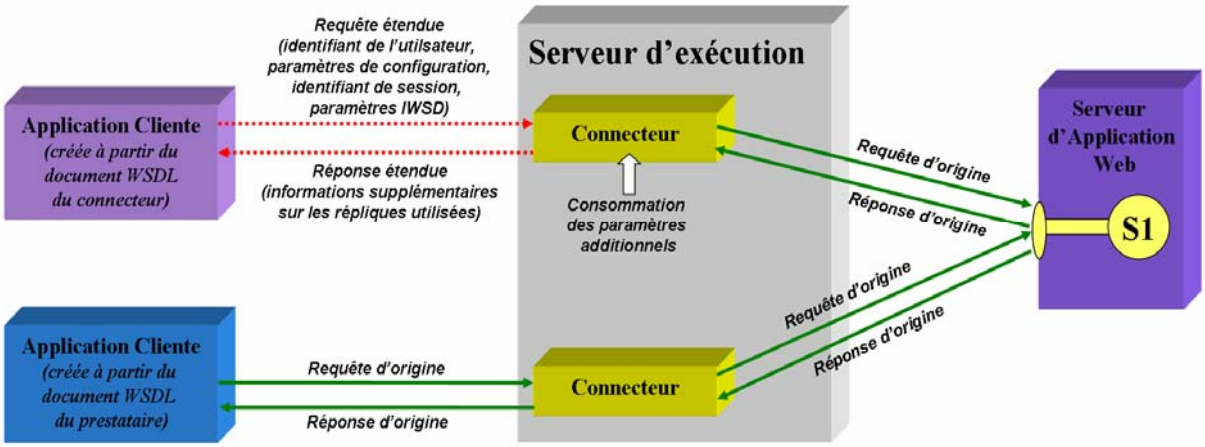

Figure 4-7: Appel d'opérations étendues

Le paramètre d'authentification inséré dans l'en-tête de chaque requête est indispensable pour qu'un client puisse être reconnu comme utilisateur de la plateforme afin qu'il puisse se servir

d'un de ces connecteurs privés. Pour les applications existantes, cette insertion d'identification dans la requête est prise en charge par le service d'écoute (cf. Chapitre 2). Des informations optionnelles sont également incorporées dans les entêtes des messages de sortie. Celles-ci insèrent des données concernant les répliques impliquées dans le traitement de la réponse fournie.

Des commentaires peuvent également être insérés dans le contrat du connecteur. Ils sont écrits sous forme de documentation (<documentation>) et, sont quant à eux, extraits à partir des commentaires écrits en DeWeL.

4.2.1 Processus de génération du contrat WSDL

La création du document WSDL d'un connecteur est effectuée à partir du document original et du canevas DeWeL. Ce processus réalisé par le ***service de génération d'interface des connecteurs*** du serveur de gestion nécessite plusieurs étapes (cf. Figure 4-8).

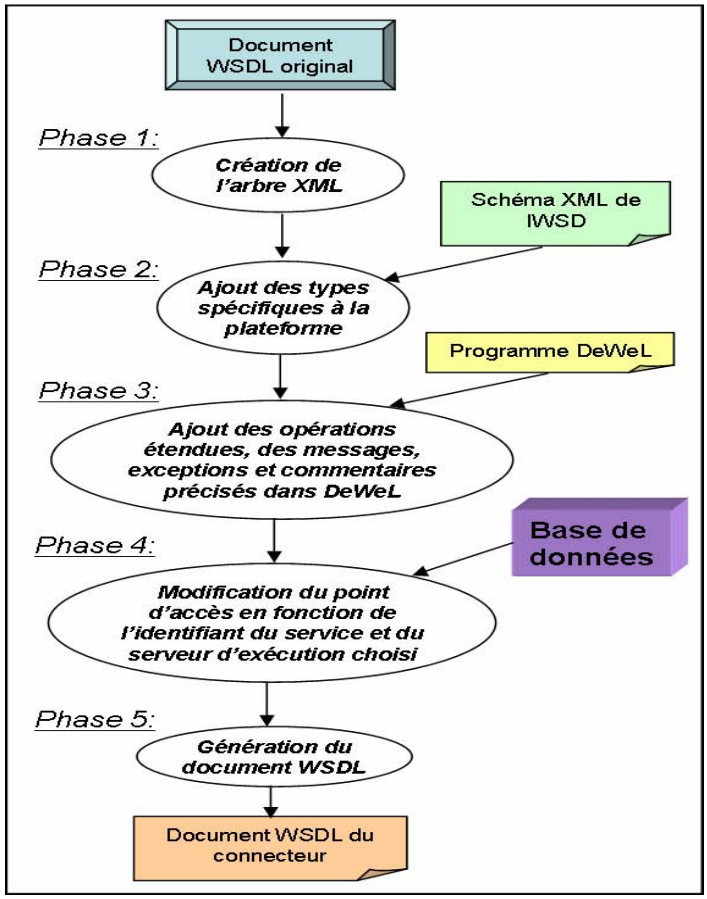

Figure 4-8: Processus de génération du contrat WSDL du connecteur

La première étape consiste à construire l'arbre XML à partir du document WSDL du service de base. L'étape suivante introduit un nouvel espace de nommage correspondant aux éléments requis et compris par la plate-forme IWSD pour pouvoir effectuer, entre autre, l'authentification du message et la configuration des mécanismes internes. Cette étape insère, en fait, dans l'arbre tous les types complexes spécifiques à la plate-forme. La troisième phase met en jeu le programme DeWeL et récupère pour chaque opération, les paramètres de configuration rajoutés par le développeur du programme et crée dans le contrat l'opération étendue en fonction de ces paramètres ainsi que du paramètre d'authentification et du paramètre de configuration (cf. Figure 4-9). La quatrième étape permet de modifier le point

d'accès du document original. Pour cela, elle interroge la base de données pour récupérer l'identifiant du connecteur et l'adresse du serveur d'exécution choisi par défaut. A la fin de ce processus, le contrat du connecteur est créé, il est référencé dans le serveur de gestion. Celui-ci joue le rôle d'annuaire UDDI pour connecteurs spécifiques de tolérance aux fautes.

En plus des opérations étendues, ce processus génère les opérations propres aux connecteurs. Ces opérations sont issues des fonctions publiques disponibles dans le langage DeWeL. Celles-ci permettent de connaître, par exemple, la disponibilité du connecteur et des services cibles utilisés. Ceci est un des atouts des connecteurs. En-dehors des actions de tolérance aux fautes que peut insérer l'utilisateur, ils permettent également d'obtenir des informations non-fonctionnelles et opérationnelles sur les Services Web impliqués dans l'application orientée service.

4.2.2 Exemples

La Figure 4-9 montre la description partielle du contrat WSDL du connecteur d'Amazon, en conformité avec le programme DeWeL donné dans la Figure 3-3.

Ce document reprend la description WSDL originale d'Amazon et ajoute des éléments non fonctionnels dans les cinq sections du document identifiées dans le chapitre 1 (cf. Figure 1-10) par les étiquettes suivantes:

- ***types***: Dans cette section, le schéma rajouté importe les types XML spécifiques à la plateforme pour effectuer les traitements non fonctionnels comme la gestion de l'authentification par exemple. D'autres schémas peuvent être rajoutés. Ces schémas correspondent à ceux importés dans le programme DeWeL et permettent d'insérer des paramètres d'entrée supplémentaires aux opérations (ligne 2).
- ***message***: Beaucoup de messages additionnels sont insérés dans le document du connecteur comme, par exemple, les messages d'exceptions spécifiques (ligne 9-11) et les messages d'entrée des opérations étendues (ligne 4-8) où sont insérés des paramètres additionnels comme DependabilityProperties (la ligne 6) et Authenticate (ligne 7). D'autres paramètres, non représentés ici, peuvent également être insérés dans cette partie tel que « idSession » qui représente l'identifiant de session attribué à un client lors de l'accès à sa session et les paramètres de configuration propre au connecteur spécifiés dans le programme DeWeL.
- ***portType***: Cette section définit toutes les opérations réalisables par le connecteur (ligne 14-25). Celle-ci reprend les opérations fonctionnelles de base du service et insère les paramètres de configuration propres au connecteur. Ces paramètres sont extraits à partir du programme DeWeL. Les opérations non-fonctionnelles issues des fonctions publiques de DeWeL sont définies par défaut pour chaque connecteur permettant de connaître, entre autres, sa disponibilité actuelle, le nombre de défaillances survenues, la date de dernière mise à jour, …etc. Les opérations de début et fin de session (« start_session » et « end_session ») sont également rajoutées.
- ***binding***: Les nouvelles opérations spécifiques au connecteur sont indiquées ici. Nous insérons le message d'authentification dans l'en-tête des requêtes SOAP (ligne 32) permettant aux utilisateurs d'accéder à un connecteur privé. Des informations supplémentaires sont rajoutées dans les réponses pour pouvoir indiquer les différentes répliques utilisées pour l'obtention de la réponse.

- ***service***: Dans cette section, le point d'accès du connecteur est précisé. Celui-ci est, bien sûr, différent du point d'accès original d'Amazon (ligne 41). La date de création (ligne 43); la date de la dernière mise à jour (ligne 44) ainsi que les répliques utilisées pour mettre en œuvre le recouvrement (ligne 45-48) sont également insérées.

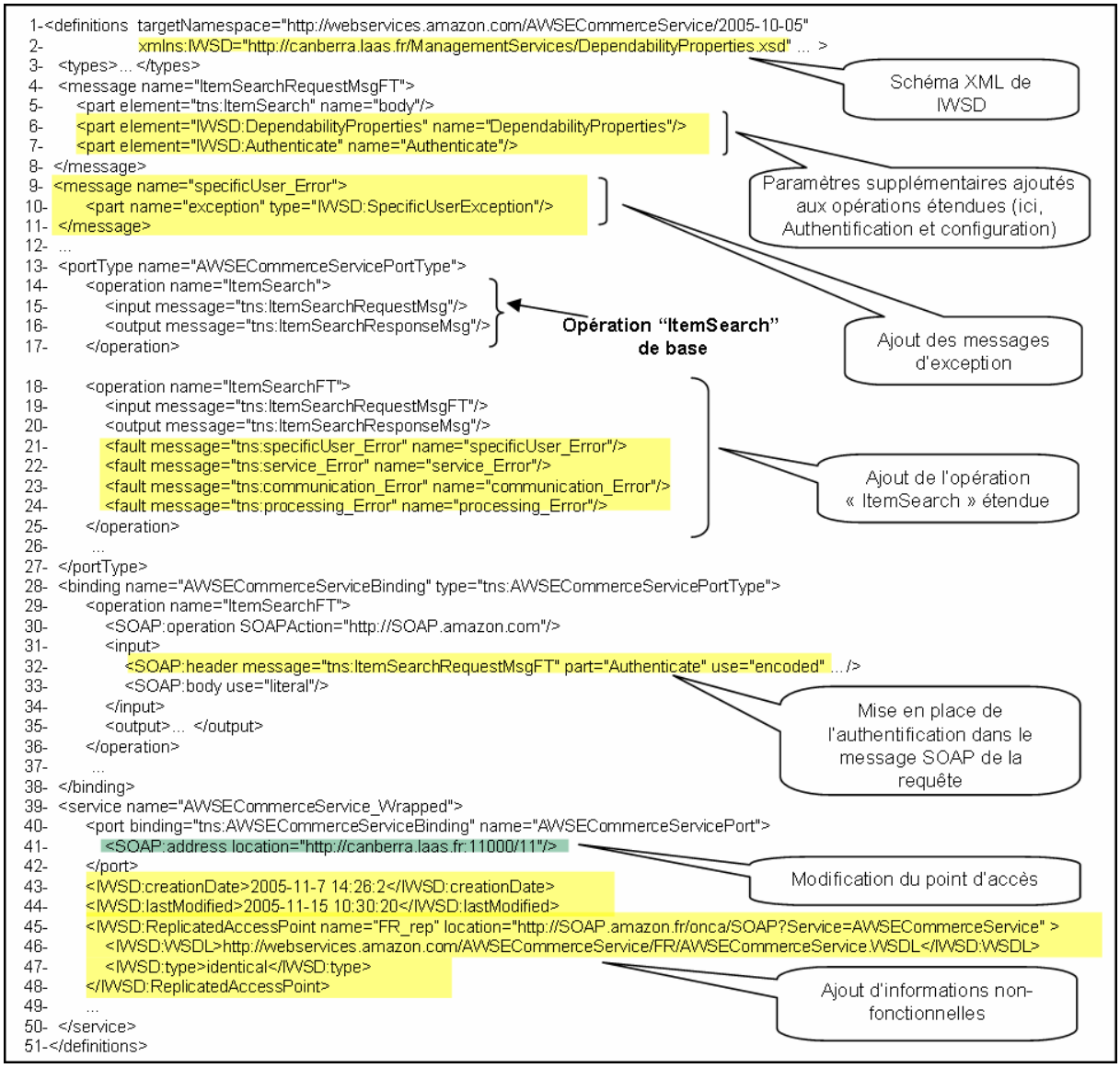

Figure 4-9: Le contrat WSDL du connecteur d'Amazon

Le document WSDL du connecteur associé à un Service Web peut être utilisé par n'importe quel client afin de créer un stub pour contacter le Service Web au travers du connecteur, si son créateur l'a déclaré comme "public".

4.3 Récapitulatif

Ce chapitre nous a permis de décrire les différents mécanismes de tolérance aux fautes dont peut être doté le connecteur. Ceux-ci peuvent être classifiés comme tels :

- ***Mécanismes de détection d'erreur :*** réalisés au travers des différents analyseurs internes (parseur SOAP et HTTP), des pré-et-post traitements mais également lors de la configuration des temps de dépassement pour la détection d'un gel de service.

- ***Mécanismes de signalement d'erreur :*** permettant grâce à la génération automatique de messages d'erreur SOAP de prévenir, de façon appropriée, le client en fonction de son application.
- ***Mécanismes de recouvrement d'erreur :*** permettant au moyen de répliques identiques ou équivalentes d'assurer la continuité de service en dépit de fautes, à partir de stratégies pré-définies.

Chaque mécanisme de recouvrement est associé à un modèle d'exécution correspondant. Ce modèle d'exécution représente en fait la séquence d'actions que devra effectuer le connecteur pendant son activation. Les modèles d'exécution définissent des stratégies de recouvrement. Celles-ci sont déclarées par l'utilisateur dans un programme DeWeL. Pour pouvoir spécifier et paramétrer ces stratégies, l'utilisateur doit tenir compte de données d'entrée importantes et d'hypothèses faites sur le service sur lequel s'appuie la stratégie. En effet, le modèle de défaillance du service (le crash, le gel ou la défaillance en valeur) ainsi que son type (avec ou sans état) sont autant de facteurs à prendre en compte pour définir la stratégie de recouvrement appropriée. La « BasicReplication » ne peut s'appliquer, par exemple, que sur des services sans état à silence sur défaillance. Dès que l'état du service doit être pris en compte, des stratégies comme la « StatefulReplication » ou la « LogBasedReplication » doivent être employées. Ces mécanismes imposent bien sûr une collaboration du prestataire et/ou du client. Les points de reprises peuvent être réalisés par le connecteur à l'aide des fonctions de gestion d'état fournies par le prestataire. La mise en place de sessions pour la « LogBasedReplication » peut être effectuée par le client à l'aide des fonctions « start_session » et « end_session » définies dans le contrat du connecteur. Les défaillances en valeur peuvent, également, être détectées et tolérées par des stratégies telle que la « VotingReplication ». D'autres mécanismes sont, bien sûr, à en prendre en compte comme la gestion de l'ordonnancement total pour la réplication active, la gestion des transactions, la notification de messages au moniteur de surveillance.

Lors de l'exécution d'un connecteur, le moniteur de surveillance collecte les différents problèmes survenus (types d'exceptions levées, nombre de répliques défaillantes, temps d'exécution, etc.). Ces informations sont accessibles via des opérations spécifiques au connecteur qui sont décrites dans son document WSDL automatiquement généré par le compilateur DeWeL. Ce document apporte ainsi des informations non-fonctionnelles et opérationnelles au contrat de base.

Les informations opérationnelles permettent aux utilisateurs de connaître la disponibilité, le taux et la fréquence des défaillances, le temps de réponse du service, le mécanisme de recouvrement ainsi que les répliques utilisées. Ces informations, fournies par le moniteur de surveillance, sont en fait nécessaires et capitales pour pouvoir réaliser une architecture orientée services semi-critique. A l'aide de celles-ci, un utilisateur peut adapter les actions de tolérances aux fautes à effectuer.

5 Résultats Expérimentaux et Analyses

L'objectif de ce chapitre est d'analyser au travers d'expériences l'efficacité du langage et de la plateforme. Pour cela, différents connecteurs ont été développés :

- des ***connecteurs de surveillance*** : ces connecteurs sont créés à partir d'un canevas où aucune action de tolérance aux fautes n'est précisée par l'utilisateur. Ils permettent seulement d'observer le comportement du service ciblé.
- des ***connecteurs de surveillance et de tolérance aux fautes*** : ces connecteurs sont créés à partir de canevas contenant des pré-et-post traitements de détection d'erreur ainsi que des stratégies de recouvrement décrites dans les chapitres précédents et définies par l'utilisateur.

Ces connecteurs vont permettre de pratiquer diverses expérimentations et d'obtenir plusieurs mesures validant le bon fonctionnement de la plate-forme.

Dans ce chapitre, nous présentons les résultats d'expériences que nous avons menées, les Services Web cibles sur lesquels nous avons travaillé et notre banc de test expérimental. Nous présentons ainsi à travers les résultats obtenus, une analyse sur les performances et la robustesse des connecteurs permettant ainsi d'améliorer la tolérance aux fautes des Services Web.

5.1 Cibles et contexte expérimental

La technologie des Services Web étant encore en voie de développement, très peu de Services Web critiques sont actuellement disponibles sur le Net. Pour réaliser des tests sur nos connecteurs, nous nous sommes donc servis de services populaires du Net. Cependant, nous avons également créé nos propres Services Web pour pouvoir injecter des fautes et vérifier l'efficacité de nos connecteurs.

Ainsi, différents Services Web ont été utilisés pour exhiber les caractéristiques spécifiques des connecteurs que nous avons utilisés tout au long de nos travaux. Parmi eux, on peut citer :

- ***Amazon :*** Célèbre site d'achat d'articles en ligne, Amazon est en fait un cas d'étude intéressant car il contient six répliques disséminées dans le monde. Grâce à ce service nous avons pu tester les différents modes de recouvrement sur l'opération de consultation.
- ***Les moteurs de recherches Google et MSNSearch :*** Ces deux services équivalents nous ont permis de créer un Service Web abstrait et de générer son connecteur associé afin de pouvoir tester différents modes de recouvrement sur des services d'origines différentes.

Les services spécifiques que nous avons développés pour nos expériences sont les suivants :

- ***Le service StockQuotes :*** ce service traditionnellement utilisé dans le e-business pour connaître le cours d'une action nous permettra de tester la robustesse des connecteurs en injectant des fautes aléatoires en valeur permettant ainsi de réaliser le recouvrement à base de vote.

- ***La calculatrice :*** ce service composite a été créé par agrégation de différents services unitaires telles que l'addition, la soustraction, la multiplication et la division. Ce service permet de tester l'ajout de plusieurs connecteurs dans une application orientée services.
- ***Le service de gestion de comptes bancaires :*** ce service avec état permet de gérer différents comptes clients sur le prestataire en créditant ou débitant leurs soldes. Ce service répliqué permet de mettre en place différents modes de recouvrement basés sur la réplication avec gestion d'état.

D'autres services Web du Net tels que FedEx ou BabelFish ont également été utilisés pour tester les performances des connecteurs mais également pour évaluer la sûreté de fonctionnement des services Web.

5.2 Banc de tests

Les architectures orientées services sont de facto des applications à grande échelle utilisant des ressources distribuées sur le Net. La mise en place de telles architectures devient donc très vite coûteuse car elle nécessite de nombreuses machines. Pour mettre en place nos tests, nous nous sommes servis d'un rack de 8 PC (PENTIUM III 133MHz, avec Linux Debian 2.4) sur lesquels nous avons installé le serveur de gestion, plusieurs serveurs d'exécution, différents clients et prestataires de services. Nous nous sommes également servis d'une machine externe au LAAS sur laquelle nous avons placé plusieurs clients, afin de tester les serveurs d'exécution lorsqu'ils sont isolés sur le Net.

Il est clair que des supports de tests tels que Grid5000 [110] ou PlanetLab [111] seront, à terme, indispensables pour tester et mettre à l'échelle de telles architectures. En effet, Grid 5000 fournira dès 2007, aux chercheurs français, une base de 5 000 postes interconnectés pour étudier et optimiser les performances des grilles de calcul tandis que PlanetLab est une plate-forme ouverte et récente pour le développement, le déploiement et l'accès à des services à échelle planétaire. Ce type de support sera donc indispensable dans l'avenir pour déployer et tester plusieurs serveurs d'exécution, exécuter différents connecteurs sur diverses applications critiques orientées services.

D'un point de vue implémentation, le serveur de gestion est une application Web installée sur un serveur Tomcat (Version 5.5.17). Elle a été réalisée en JSP/Servlet. Une partie en JNI a également dû être réalisée pour générer le canevas DeWeL afin de le rendre accessible par les utilisateurs au travers de l'interface Web. Le compilateur DeWeL a été réalisé en Flex et Bison. Le serveur d'exécution ainsi que les connecteurs ont été réalisés en C++. Nous nous sommes servis de librairies C++ réalisées au LAAS pour mettre en place certains mécanismes de recouvrement et de la librairie Xerces-C++ pour implémenter les différents parseurs SOAP, XML Schema et WSDL. Le projet correspond à environ 55 000 lignes de code.

L'évaluation de la plate-forme s'effectue à plusieurs niveaux. Tout d'abord, le langage DeWeL qui constitue le seul moyen pour chaque utilisateur de programmer un connecteur doit être validé. La plateforme doit ensuite être caractérisée. Enfin, la performance, la robustesse et l'efficacité des connecteurs doivent également être évaluées au travers de tests mettant en œuvre différents mécanismes de recouvrement.

5.3 Evaluation du langage

Pour évaluer notre langage, nous nous sommes concentrés sur deux points distincts : l'expressivité et la concision. Bien que de nombreuses restrictions aient été imposées, il est

important de vérifier de façon pragmatique les capacités du langage pour développer des connecteurs réalistes.

5.3.1 Expressivité

Evaluer l'expressivité en pratique d'un DSL exige de l'utiliser sur une large variété d'applications non-triviales. En effet, il est impossible de pouvoir quantifier ou de mesurer l'expressivité d'un langage. La seule façon d'y arriver est de le montrer par l'exemple.

Nous avons utilisé DeWeL pour empaqueter 170 Services Web (voir la Figure 5-1). Parmi eux, nous avons, entre autres, testé de nombreux connecteurs sur différents services capables de réaliser les mécanismes de recouvrement présentés dans le chapitre 4, et développé divers connecteurs capables de réaliser différentes assertions spécifiques comme, par exemple, un contrôle parental pour le service Web de Google. Quelques un de ces exemples ont d'ailleurs été présentés dans le chapitre 3.

En fait, l'expressivité de DeWeL reste assez analogue à celle d'un langage usuel tel que Java ou C++. Elle se résume à de la manipulation d'objets et de méthodes, l'utilisation de fonctions prédéfinies et d'opérateurs. Des restrictions sont simplement faites sur les instructions et certaines propriétés critiques telles que l'allocation dynamique de mémoire. Cela ne limite en rien les objectifs de DeWeL dont le but majeur est de pouvoir réaliser des assertions de pré-et-post traitements et des mécanismes de recouvrement.

5.3.2 Concision

Nom du Service Web	Canevas DeWeL	Types Spécifiques (*.hpp et *.cpp)	Connector (*.hpp et *.cpp)	Code total généré par DeWeL	Ratio
SQLDataSoap	46	18	833	851	18,50
AWSECommerceService	238	16979	4385	21364	89,76
DNSLookupService	34	512	577	1089	32,03
MapPoint	70	12935	1249	14184	202,63
MSNSearchService	22	1676	353	2029	92,23
WS4IsqlService	10	1225	101	1326	132,60
eBayWatcherService	22	0	354	354	16,09
BankValSOAP	22	0	354	354	16,09
SMS	70	927	1249	2176	31,09
…	…				
BNQuoteService	22	0	354	354	16,09
QueryIP	22	186	353	539	24,50
BabelFishService	22	0	358	358	16,27
pop	34	0	602	602	17,71
Fax	82	1008	1473	2481	30,26
SendEmailService	22	222	353	575	26,14
Airport	58	666	1025	1691	29,16
GoogleSearchService	46	555	845	1400	30,43
XMethodsQuery	70	543	1239	1782	25,46
Moyenne:	64,76	1093,88	1151,83	2245,71	31,56

Figure 5-1: Comparaison entre DeWeL et le langage C (en nombre de lignes de code)

Pour évaluer la concision, plus de 170 canevas DeWeL ont été produits à partir du document WSDL du service correspondant (cf. colonne 1 de la Figure 5-1). Ce tableau

représente les résultats partiels obtenus sur une vingtaine de services[15]. A partir de ces canevas, nous avons généré les fichiers créés lors du processus de la génération de code vu au chapitre 3 (cf. section 3.5.4) :

- Les types spécifiques rattachés à chaque service (cf. colonne 2 de la Figure 5-1),
- Le modèle d'exécution du connecteur (cf. colonne 3 de la Figure 5-1) avec les fonctions de pré et post traitements définies par l'utilisateur.

Les connecteurs sont développés à partir de canevas n'ayant aucune action de tolérance aux fautes définie par l'utilisateur. Leur modèle d'exécution est le modèle linéaire par défaut. Cependant, ces connecteurs offrent les fonctionnalités de base tels que le monitoring et l'accès à des informations non-fonctionnelles comme la disponibilité du service cible. Ce sont juste des connecteurs de surveillance. La colonne 4 de la Figure 5-1 représente la totalité du code généré par DeWeL à partir de ce type de canevas. Nous observons que les programmes DeWeL sont de l'ordre 30 fois plus petits que leur version équivalente en C dans les cas où les pré et post traitements sont vides (cf. dernière ligne de la colonne 5 de la Figure 5-1). Ceci permet de diminuer de façon drastique les fautes du logiciel à condition que la chaîne d'outils ait un haut niveau de qualité de service, ce qui est généralement le cas pour des langages dédiés comportant de fortes restrictions par rapport à des langages standards..

Par ailleurs, l'insertion de mécanismes de sûreté de fonctionnement, c'est-à-dire, des pré et post-traitements implémentés par un utilisateur, ne peut qu'augmenter ce rapport. Ceci peut être visualisé sur la Figure 3-21 où on obtient un rapport de 4,2 entre les actions écrites en DeWeL et celles traduites en C.

DeWeL est donc indispensable au bon fonctionnement de la plate-forme IWSD. Il permet de manipuler ses ressources critiques (manipulation de fichiers au travers des variables à mémoire, accès à la base de données, création de notations spécifiques) de façon sûre avec des primitives adaptées. Il permet d'écrire et de déployer un connecteur robuste sur la plate-forme pour n'importe quel Service Web ciblé.

5.3.3 Analyse et Prospective

Bien que l'expressivité et la concision soient des notions capitales, la validité du langage passe également par l'évaluation de la robustesse. Pour évaluer la robustesse d'un compilateur, la méthode la plus couramment utilisée est de générer des mutants. L'analyse des mutations est une technique de test unitaire fondée sur l'injection de fautes qui date de la fin des années soixante-dix [112, 113]. Elle a été utilisée avec succès pour tester des programmes écrits dans des langages tels que C, Fortran ou ADA [114, 115].

Pour un programme P, l'analyse de mutations produit un ensemble de programmes alternatifs dont chaque élément P_i est appelé un mutant de P. Un mutant est obtenu en modifiant une seule construction de P à la fois, suivant un ensemble de règles de mutations. Ces règles de mutations sont dérivées à partir d'études sur les erreurs généralement faites par les programmeurs lorsqu'ils traduisent une spécification en un programme [116].

Dans l'analyse de mutations traditionnelles, on veut évaluer la couverture d'un ensemble *t* de tests par rapport à un programme P. Le principe de l'analyse de mutations est que, si *t* couvre

[15] Le tableau complet est présenté en annexe A.2

convenablement P, alors au moins un des tests de *t* devrait pouvoir distinguer P d'un mutant P'. La proportion de mutants qui meurent (ceux qui sont détectés) pendant l'analyse de mutation indique à quel point P est couvert par *t*.

Pour évaluer la robustesse du langage, deux solutions sont envisageables dans le cadre de notre étude:

- Soit générer des mutants au niveau de l'interface WSDL utilisée comme point d'entrée par tous les générateurs de stubs de Services Web et de notre compilateur.
- Soit générer des mutants directement dans des programmes DeWeL et comparer les résultats obtenus en faisant des tests analogues dans le langage C.

Dans la première solution, le nombre de mutants est assez réduit puisque l'on ne peut agir essentiellement que sur l'interface: la signature des opérations du Service Web. Les mutations ne peuvent donc s'effectuer que sur des identificateurs (de type, de variable ou d'opération). D'autres mutations pourraient être réalisées en effectuant des inversions, des substitutions, des ajouts ou des suppressions d'éléments XML contenus dans le contrat WSDL. Les travaux présentés dans [117] prévoient neuf types de mutations dans ce but. Il est à noter que la plupart de ces mutations ne pourraient en aucun cas être détectées à la compilation mais permettraient de caractériser à l'exécution le comportement des Services Web en présence de fautes.

La seconde solution permet de tester de manière plus rigoureuse le comportement du compilateur DeWeL en présence de fautes. Dans ce but, une analyse minutieuse des différents mutants possibles doit être réalisée. Nous prévoyons plusieurs types de mutations possibles (sur les littéraux, les opérateurs, les identificateurs, …etc.). Cependant, comme DeWeL est à l'heure actuelle le seul langage pour déployer des connecteurs sur la plate-forme, il est difficile de pouvoir le comparer avec des langages traditionnels. Une solution serait de fournir à une équipe de développeur l'API des différents composants de la plate-forme et de leur faire implémenter la librairie dynamique et donc la fonction « start_connector » en fonction de l'interface fournie par le document WSDL du service à tester. Un retour sur cette expérience nous permettrait d'avoir une connaissance plus fine sur l'expressivité et la performance du langage.

Le langage étant le seul moyen d'écrire des connecteurs, nous nous sommes ici surtout concentré sur les performances de la plate-forme et le comportement des connecteurs en présence de Services Web défaillants puisqu'il est difficile de le comparer avec un autre langage. Nous pouvons tout de même signaler que nous avons effectué de nombreux tests en insérant des mutants dans les programmes DeWeL. Ces mutants ont tous été détectés à la compilation et nous ont permis de nous convaincre de la robustesse du langage. Par ailleurs, les restrictions importantes apportées au langage ont permis de limiter le nombre de vérifications imposées au compilateur DeWeL. Il en résulte au final un compilateur simple et fiable. Plus de 200 programmes DeWeL ont été compilés. Chacun a permis de voir le gain obtenu au niveau des lignes de codes générées. Cette particularité permet d'avoir un effet très positif sur le nombre de fautes logicielles.

5.4 Evaluation des performances de IWSD

Pour évaluer les performances de la plate-forme IWSD, nous avons réalisé trois jeux d'expériences. Dans un premier temps, nous avons comparé notre solution avec des intercepteurs classiques pouvant être réalisés à partir de composants existants. Nous avons ensuite étudié les performances obtenues sur des connecteurs de surveillance ciblant différents

types de service. Enfin, nous avons analysé l'impact des stratégies de recouvrement sur les performances du système en créant des connecteurs de tolérance aux fautes.

5.4.1 Comparaison avec des intercepteurs classiques

Nous avons réalisé plusieurs tests afin d'évaluer les performances de la plate-forme. Le premier d'entre eux fût de la comparer avec des solutions équivalentes. Pour cela, nous avons réalisé trois solutions différentes à base de médiateur (ou proxy). Ces médiateurs possèdent les caractéristiques de base de nos connecteurs, à savoir, intercepter les requêtes et les réponses entre un client et un prestataire. Nous avons réalisé nos tests à partir d'un Service Web simple faisant transiter environ 20 Ko de données. Les prestataires, les clients et les médiateurs sont installés sur la même machine pour éviter les perturbations du réseau. Ces solutions sont représentées dans la Figure 5-2 et détaillées ci-dessous.

- La première solution représente le cas classique où aucun médiateur n'est inséré. Le client contacte directement le prestataire.
- La deuxième solution a été réalisée en Java et s'appuie sur un serveur Tomcat et Axis pour sérialiser et désérialiser les requêtes SOAP,
- La troisième solution a été réalisée en exécutant un connecteur sur le serveur d'exécution. Il charge et active le connecteur capable de réaliser les mêmes fonctionnalités que le médiateur réalisé en Java (c'est-à-dire, sérialiser et désérialiser les requêtes SOAP), mais également de diagnostiquer le Service Web ciblé en transmettant des informations au moniteur de surveillance.

Les résultats obtenus sont présentés sur la Figure 5-2.

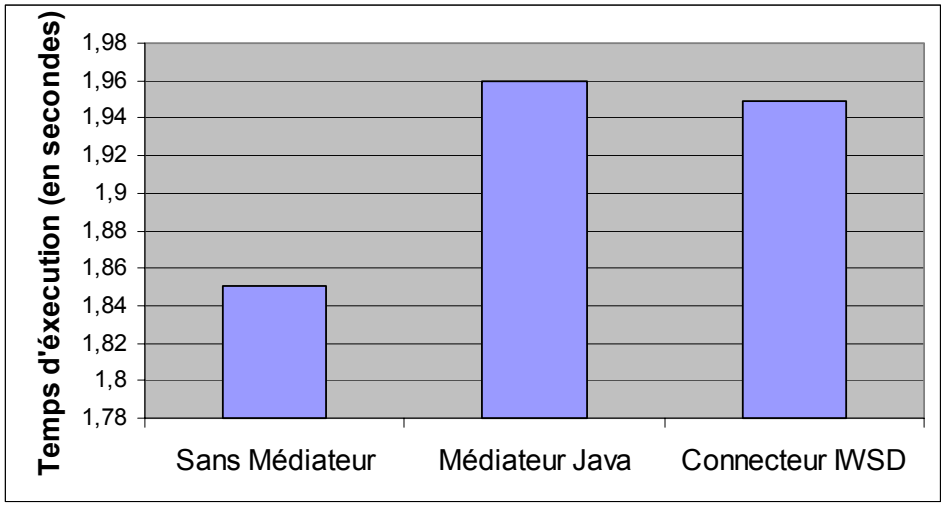

Figure 5-2: Comparaison des médiateurs

Au travers de ces tests, on peut souligner les faibles performances du médiateur Java qui insère un surcoût temporel de plus de 5,93%. Ceci peut s'expliquer par le nombre de couches logicielles à installer pour réaliser ce médiateur (la machine virtuelle JAVA, le serveur d'application Tomcat, et le moteur SOAP Axis). Lorsqu'on insère la plate-forme IWSD avec le chargement d'un connecteur spécifique capable de monitorer et d'insérer des mécanismes de tolérance aux fautes, le surcoût constaté est d'environ 5,36%. Ce résultat est bien sûr dépendant du service ciblé.

Il est cependant intéressant de remarquer que la performane de la solution proposée est du même ordre de grandeur que celles des solutions classiques n'offrant pas des mécanismes pour assurer la sûreté de fonctionnement

5.4.2 Performance des connecteurs de surveillance

Pour réaliser les expériences suivantes, nous nous sommes servis de Services Web sur étagères pris sur le Net. Différents tests ont été réalisés pour valider les diverses fonctionnalités et les outils d'IWSD, mais aussi pour effectuer des mesures de performance. En particulier, nous évaluons le surcoût d'exécution introduit par IWSD entre le client et le fournisseur.

Dans la première phase de l'expérience, nous avons installé des clients, le IWSD et des fournisseurs factices du Service Web d'Amazon sur la même machine (un PC, PENTIUM III 133MHz et Linux Debian 2.4) pour éviter les perturbations dues au temps de latence introduit par les réseaux de communication. La Figure 5-3 montre les résultats obtenus avec un Service Web factice d'Amazon renvoyant toujours une réponse prédéfinie correcte. La taille des messages SOAP transmis est dans ce cas de l'ordre de 100 Ko.

La courbe inférieure représente le temps de réponse standard entre le client et le fournisseur. La courbe supérieure représente le temps de réponse quand le IWSD est inséré. Le coût temporel supplémentaire est égal à 6.8% en moyenne, ce qui montre que l'insertion d'un intermédiaire entre client et prestataire n'est pas pénalisant, dans le contexte d'application à grande échelle à base de Service Web.

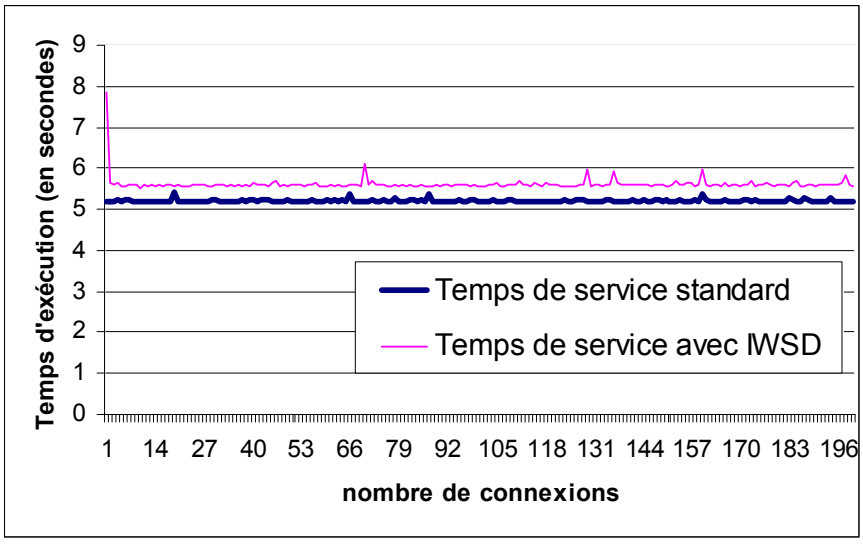

Figure 5-3: Temps d'exécution avec un prestataire factice d'Amazon

Dans la seconde phase de l'expérience, les clients et les serveurs d'exécution sont sur des machines différentes. Les clients se connectent à distance au vrai fournisseur du Service Web d'Amazon (voir la Figure 5-4).

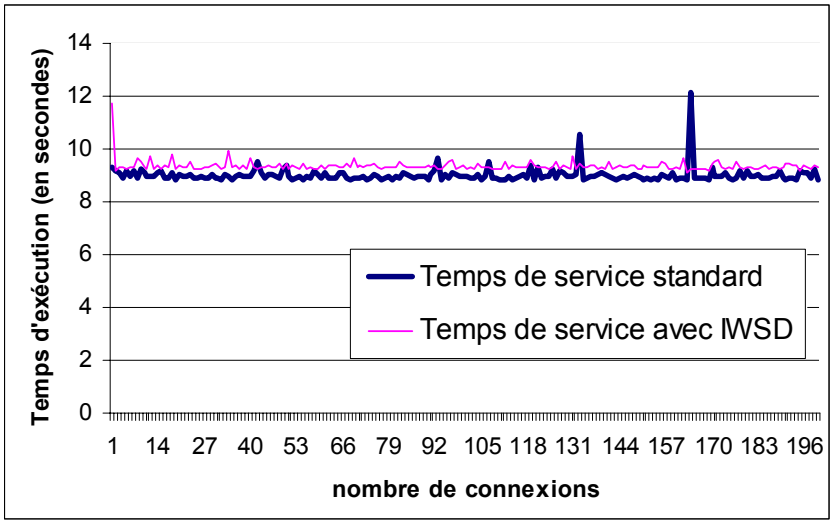

Figure 5-4: Temps d'exécution avec le prestataire d'origine d'Amazon

Le coût temporel relatif moyen diminue dans ce cas-ci à 4.36 %, parce que le temps de réponse est plus élevé en raison d'une période de latence plus grande, d'un temps de calcul de la prestation plus conséquent et/ou des perturbations réseaux plus importantes. Clairement l'impact des communications distantes réduit encore plus l'effet de l'IWSD sur la plate-forme globale.

La Figure 5-5 montre les résultats sur les temps d'exécution obtenus sur différents Services Web (Amazon, Fedex, Google, BabelFish, MSNSearch). Nous nous sommes basés sur des services de complexité variable (c'est-à-dire, la complexité des requêtes SOAP). En général, le temps écoulé dans une session IWSD est proportionnel à la complexité des requêtes SOAP. Plus les messages SOAP sont complexes, plus le temps d'analyse et le surcoût d'exécution sont grands.

	Taille des messages SOAP transmis (octet)	Temps de service standard (sec)	Sans Mécanismes de recouvrement		
			Temps de service standard avec IWSD (sec)	Temps d'exécution du connecteur (sec)	Surcoût d'exécution (%)
FedEx	58017	4,615	4,625	0,010	0,216
BabelFish	1623	3,377	3,468	0,091	2,694
MSN Search	7845	3,282	3,356	0,074	2,254
Amazon	108378	9,006	9,399	0,393	4,367
Google	7496	8,077	8,720	0,642	7,959

Figure 5-5: Expériences réalisés par plusieurs Services Web

5.4.3 Performance des connecteurs de tolérance aux fautes

La Figure 5-6 présente les actions de recouvrement réalisées grâce à plusieurs instances du Service Web d'Amazon. Nous avons utilisé les six points d'accès différents (soap.amazon.fr, soap.amazon.ca,… etc.) de ce Service Web pour mettre en œuvre les stratégies de réplication. Nous pouvons noter dans cette figure que l'ajout de traitements spécifiques de tolérance aux fautes augmente, bien sûr, sensiblement le temps d'exécution. Le surcoût observé pour la réplication active est le plus faible parce que l'IWSD renvoie, dans ce cas, la réponse du point d'accès le plus rapide.

	Avec Mécanismes de recouvrement					
	Basic Replication			Active Replication		
	Temps de service time avec IWSD (sec)	Temps d'exécution du connecteur (sec)	Surcoût d'exécution (%)	Temps de service avec IWSD (sec)	Temps d'exécution du connecteur (sec)	Surcoût d'exécution (%)
Amazon	9,417	0,411	4,568	9,390	0,384	4,094

Figure 5-6: Expériences sur Amazon avec recouvrement d'erreur

Pour réaliser les expériences suivantes, nous nous sommes servis de deux services distincts :

- Le service « StockQuotes » sans gestion d'état : Ce service n'a pas de mémoire des opérations effectuées.
- Le service de « comptes bancaires » avec gestion d'état : le résultat d'une opération dépend, ici, de l'état du service et donc des opérations précédemment effectuées.

Chacun de ces services possède trois répliques identiques. Nous avons créé des connecteurs implémentant les mécanismes de recouvrement appropriés.

Le premier jeu d'expériences a été réalisé sur le service « StockQuotes ». Les connecteurs créés pour ce service sont listés ci-dessous :

- Le connecteur ***sans recouvrement*** : Ce connecteur ne lance aucun mode de recouvrement. Aucun pré ou post traitement n'est défini comme dans les autres connecteurs ci-dessous. Seules les fonctions de diagnostic et de monitoring sont activées. C'est un connecteur de surveillance.
- Le connecteur avec le mode ***BasicReplication*** : Dans ce mode, le connecteur bascule sur une nouvelle réplique après une défaillance de la première.
- Le connecteur avec le mode ***ActiveReplication*** : le connecteur envoie la requête à toutes les répliques.
- Le connecteur avec le mode ***VotingReplication*** : le connecteur envoie la requête à toutes les répliques avant d'effectuer le vote.

Pour activer les différents modes de recouvrement sur ces connecteurs, des défaillances de services sont provoquées de façon aléatoire (les prestataires retournent un message d'erreur au lieu d'une réponse correcte). Sur les 2000 requêtes envoyées, environ 800 défaillances sont provoquées. Il est important de noter ici, que les services n'ayant pas de gestion d'état, le nombre de répliques utilisées reste constant pendant toutes les expériences. Elles ne sont jamais désactivées.

Nous avons comparé le temps d'exécution d'un connecteur n'ayant aucun mode de recouvrement avec les autres. Les durées représentées sur la Figure 5-7 correspondent au temps de passage dans le connecteur (le temps de session). La réplication avec vote a le surcoût le plus élevé (9,66 %). Ce surcoût est dû au fait que le connecteur est obligé d'attendre les réponses de toutes les répliques (au moins d'une majorité) avant de pouvoir effectuer le vote. La réplication passive introduit un surcoût d'exécution de 2,18 % tandis que

celui de la réplication active est de 1,39 %. Dans le dernier cas, le temps de réponse correspond à la première réplique qui transmet une réponse correcte.

En conclusion de cette expérience, on peut se rendre compte que la surcharge temporelle due aux connecteurs varient entre 65 et 72 millisecondes pour le service choisi comme cas d'étude, ce qui correspond à un surcoût temporel total variant entre 3,23 et 3,54% par rapport à une application sans connecteur.

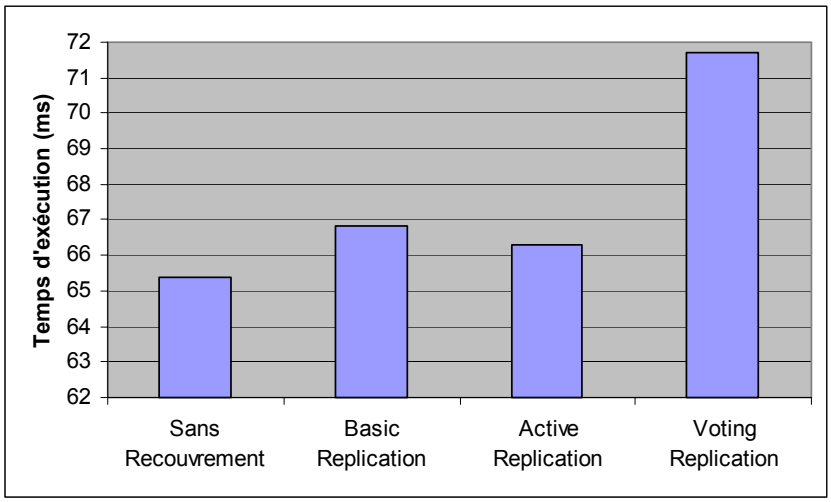

Figure 5-7: Comparaison des différents modes de recouvrement sans état

Le deuxième jeu d'expériences a été effectué sur le service à état de « comptes bancaires ». Ce service permet de créditer ou débiter de l'argent sur des comptes de différents clients.

Différents connecteurs ont été implémentés chacun mettant en oeuvre un mécanisme de recouvrement approprié pour les services avec gestion d'état. Ces connecteurs sont listés ci-dessous :

- Le connecteur avec le mode ***StatefulReplication***: Pour le mettre en oeuvre, nous avons construit les fonctions de gestion d'état s'agrégeant au contrat de base du prestataire. Les opérations « *Save_State* » et « *Restore_State* » ont été ainsi redéfinies par le prestataire. L'état du prestataire primaire est sauvegardé avant chaque envoi de requête. Sur défaillance du primaire, l'état du service est chargé sur le secondaire avant d'effectuer la prestation.
- Le connecteur avec le mode ***LogBasedReplication*** : ce mode nécessite de régénérer l'application cliente à partir du contrat du connecteur créé. En effet, celui-ci contient les opérations nécessaires par démarrer et arrêter une session. Dans ce mode, l'identifiant de la session est envoyé dans chaque requête d'opération bancaire. Ces requêtes sont sauvées et rejouées sur un serveur de secours sur défaillance du primaire.
- Le connecteur avec le mode ***ActiveReplication*** : ce connecteur envoie la requête reçue à toutes les répliques impliquées dans le recouvrement. L'ordonnancement total des messages est dans ce cas primordial. Contrairement à la réplication active pour les services sans gestion d'état que l'on a vu précédemment, lorsqu'une réplique défaille, celle-ci ne peut plus faire partie de la liste des répliques autorisées, car elle n'est plus cohérente avec les autres. Ce mode s'arrête donc dès qu'il ne reste plus aucune ressource disponible.

- Le connecteur avec le mode ***VotingReplication*** : ce connecteur envoie la requête reçue à toutes les répliques impliquées dans le recouvrement avant d'effectuer le vote. Ce mode a les mêmes contraintes que le mode précédent.

Les trois répliques utilisées dans ce test défaillent à des fréquences différentes :

- 1 défaillance sur 2 sur la première réplique
- 1 défaillance sur 3 sur la seconde réplique
- 1 défaillance sur 4 sur la troisième réplique

Ce test permet d'évaluer les connecteurs dans les pires cas, quand le taux de défaillance de chaque réplique est élevé. Les résultats obtenus sur les surcoûts temporels sont présentés dans la Figure 5-8. Ces temps d'exécution représentent le temps passé dans chaque connecteur.

Dans les tests présentés dans la Figure 5-8, nous comparons le surcoût d'exécution entre le connecteur sans recouvrement et les autres. A part pour le mode « StatefulReplication » qui a la capacité d'accéder à l'état d'une réplique et de la restaurer, les autres stratégies désactivent les répliques défaillantes. Ainsi, lorsque le nombre de répliques (n) ne permet plus de tolérer les erreurs ($n < 1$ pour le mode « ActiveReplication » et « LogBasedReplication », $n <= 2$ pour le mode « VotingReplication »), l'expérience est arrêtée. Le système est réinitialisé avant d'effectuer l'expérience suivante.

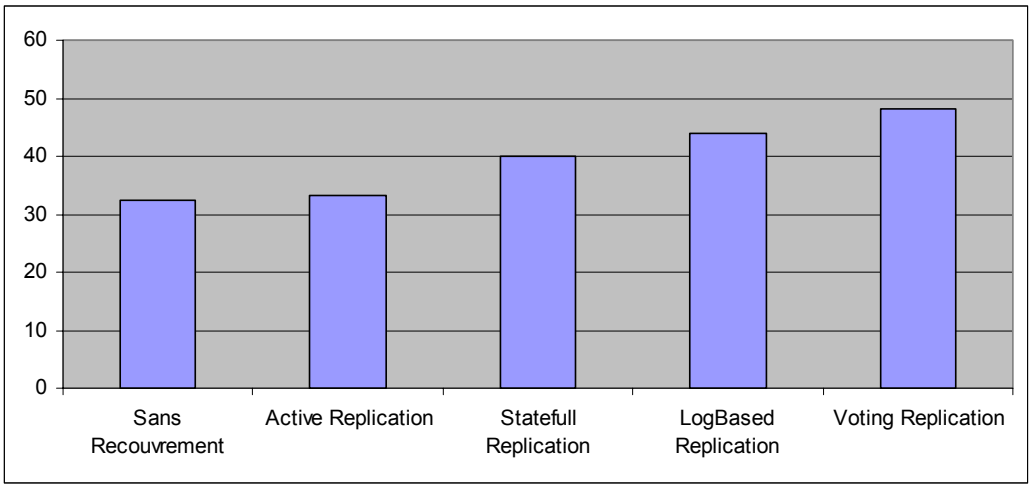

Figure 5-8: Comparaison des différents modes de recouvrements avec état

Parmi, ces cinq connecteurs, on peut se rendre compte que le connecteur avec le mode « ActiveReplication » reste le plus efficace du point de vue de la performance. En effet, la surcharge temporelle est comparable à celle du test précédent (2,4% au lieu de 1,39%). Dans l'expérience que nous avons menée le nombre de clients utilisant le connecteur étant égal à un, les problèmes d'ordonnancement total ne sont pas ici pris en compte.

L'augmentation du nombre de messages transmis pour sauvegarder et charger l'état sur la réplique de secours permet de comprendre le surcoût du connecteur du mode « StatefulReplication » qui est de 23%. Contrairement aux autres modes, celui-ci a la particularité de gérer l'état des répliques et a donc la possibilité de restaurer une réplique défaillante. Le nombre de répliques utilisables par le connecteur reste donc constant dans le temps.

Le connecteur avec le mode « LogBasedReplication » sauvegarde toutes les requêtes. Ce mode sollicite énormément les capacités du connecteur en le forçant à stocker un nombre considérable de données, et l'obligeant à rejouer toutes les requêtes enregistrées. Dans cette stratégie, lorsqu'une réplique défaille, elle n'est plus cohérente par rapport aux répliques initiales connues du connecteur, à moins que le prestataire ne la réinitialise. Elle est donc supprimée de la liste des répliques utilisables. Dès qu'il ne reste plus de répliques disponibles, l'expérience est stoppée avant de réinitialiser le système pour réaliser l'expérience suivante. Le surcoût temporel obtenu est alors de 35%.

Il faut bien se rendre compte ici que ce surcoût dépend énormément du nombre et de la fréquence de défaillances des répliques. En effet, l'expérience s'arrêtant dès l'épuisement de répliques, le connecteur ne stocke que peu de requêtes puisque les services défaillent assez rapidement. Avec un nombre de répliques plus grand et surtout un taux de défaillance plus faible, ce surcoût pourrait fortement augmenter. En fait, ce genre de recouvrement ne devrait être utilisé que sur des courtes sessions ne nécessitant que peu de requêtes à conserver, et des répliques avec un taux de défaillances réellement faible. Sur des sessions plus longues, il serait bon de pouvoir réaliser régulièrement des points de sauvegarde sur l'état du primaire permettant ainsi de supprimer les requêtes enregistrées au préalable et de pouvoir restaurer un état cohérent sur les répliques. Ceci nécessite bien sûr une implication plus importante du prestataire tout comme dans le mode « StatefulReplication ».

Enfin, le connecteur avec le mode « VotingReplication » a le même comportement que le connecteur de l' « ActiveReplication ». Dès que le nombre de répliques n'est plus suffisant pour satisfaire le vote, l'expérience est stoppée. Dans ce mode, le connecteur est contraint d'attendre les réponses de toutes les répliques avant de pouvoir réaliser le vote, ce qui explique que le surcoût d'exécution soit de 48%.

En conclusion de ces expériences, on peut se rendre compte que les temps d'exécution des connecteurs varient entre 32 et 48 millisecondes pour le service donné, ce qui correspond à un surcoût temporel total variant entre 4,28 et 4,95% par rapport à l'application sans connecteur.

En définitive, les résultats obtenus sur les performances ont permis de mettre en évidence deux éléments importants :

- La solution que nous proposons permet d'installer des connecteurs de surveillance capables d'observer le service cible et de fournir à l'utilisateur des informations non-fonctionnelles et opérationnelles essentielles en fournissant un surcoût d'exécution comparable à des intercepteurs classiques sans aucune fonctionnalité particulière.
- Le surcoût temporel des connecteurs implémentant des mécanismes de recouvrement varie fortement suivant le type de stratégies utilisé.

Cette stratégie peut d'ailleurs être définie en fonction des informations non-fonctionnelles et opérationnelles fournies par les connecteurs de surveillance. Il est important de voir ici que les chiffres obtenus sont fonction de différents facteurs : le type service ciblé (avec ou sans gestion de l'état), la taille des messages transitant par le connecteur, le nombre de répliques utilisées, la fréquence des défaillances, la taille de l'état à transférer, le nombre de clients utilisant simultanément le même connecteur, …etc).

Une étude de sensibilité sur ces différents paramètres permettrait de voir l'impact de chacun d'eux sur les performances du système et de pouvoir constater quels facteurs nuisent réellement aux performances des connecteurs. Un paramètre important est naturellement le type et la complexité de l'application.

5.5 Monitoring des Services Web

La plateforme IWSD peut être utilisée pour effectuer différentes mesures. Par exemple, la disponibilité de divers Services Web peut être évaluée grâce aux informations d'erreurs collectées par le moniteur de surveillance.

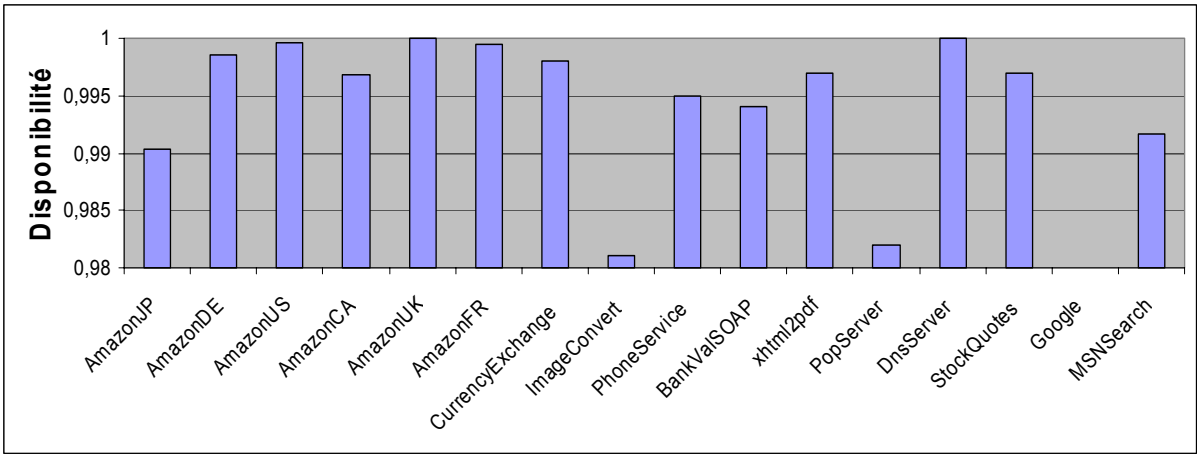

Figure 5-9: Disponibilité des Services Web

Les résultats obtenus sont donnés sur la Figure 5-9. Bien que le moniteur de surveillance fournisse d'autres types d'informations (comme le temps de session, les nombres de requêtes reçues, les nombres d'exceptions de communication ou de service survenues, etc.), nous rapportons ici seulement le rapport de disponibilité. Approximativement 2000 requêtes ont été envoyées à chaque service ciblé et nous avons observé que la disponibilité de ces services candidats pouvait considérablement changer durant les expériences. Par exemple, dans nos expériences, Google a exhibé le plus faible rapport de disponibilité (environ 82%) provoqué par 352 erreurs de communication sur 1913 requêtes envoyées. Parmi ces erreurs, 64 sont dues à une latence de temps de réponse trop grande et 288 dues à une indisponibilité du serveur (codes d'erreur : HTTP 502 Bad Gateway). Il faut bien comprendre que l'on ne parle pas du site Web de Google au demeurant très performant mais de son Service Web. Les besoins ne sont pas, ici, identiques. Une indisponibilité d'un site Web oblige l'utilisateur à se reconnecter ultérieurement. Une indisponibilité sur un Service Web peut bloquer tous les services en attente de sa réponse et rendre l'application orientée services défaillante.

Parmi les 6 répliques d'Amazon, seulement une (AmazonUK) a atteint un rapport de disponibilité de près de 100%. Les défaillances qui se sont produites étaient principalement dues aux erreurs de communication (communicationException) correspondant à une latence trop longue. Cette exception est déclenchée suite au dépassement de la valeur de temporisation définie par l'utilisateur. Par défaut, ce temps est fixé à 10 secondes. Ce paramètre permet de détecter un gel de service. Une valeur trop petite affectée à ce paramètre peut déclencher de fausses alarmes. Un temps trop long entraîne une latence de détection trop grande.

L'apport de telles informations est indispensable pour déterminer la sûreté de fonctionnement d'un Service Web et décider des assertions et mécanismes de recouvrement appropriés à mettre en œuvre dans les connecteurs associés.

5.6 Impact des mécanismes de recouvrement sur la disponibilité des Services Web

A partir des informations fournies par le moniteur de surveillance, nous avons créé un connecteur sélectionnant la réplication active avec différentes répliques d'Amazon.

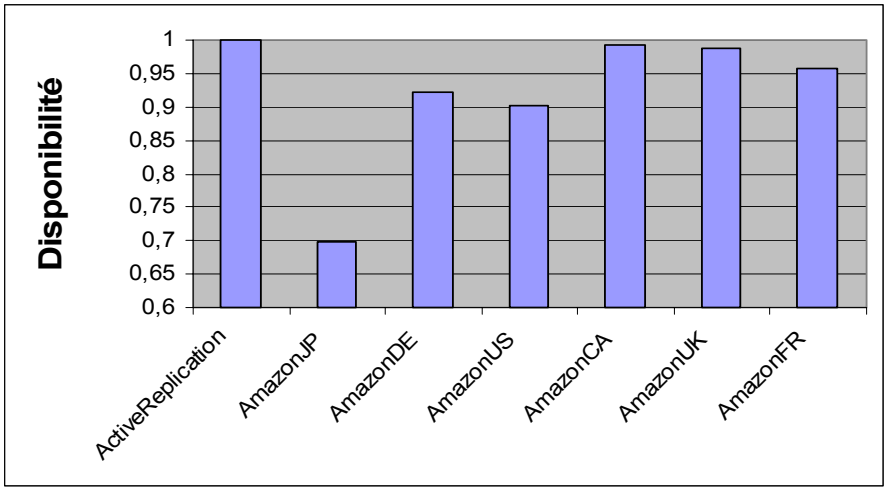

Figure 5-10: Réplication Active avec Amazon

La Figure 5-10 montre les résultats de disponibilité obtenus en utilisant cette stratégie de recouvrement. Comme espéré, le rapport de disponibilité est de 100%, bien que la disponibilité de quelques répliques puisse être très basse (par exemple, environ 70% pour AmazonJP). Pendant une expérience avec un connecteur utilisant la réplication passive, nous avons observé qu'une erreur transitoire a causé la redirection de la requête sur quatre répliques avant de renvoyer une réponse valide!!!

On peut constater ici qu'un des objectifs essentiels des connecteurs est atteint : améliorer la disponibilité des Services Web.

5.7 Utilisation des mécanismes de recouvrement sur des services équivalents

Nous avons également exécuté des expériences avec des services équivalents, à travers la notion de requête abstraite. Comme Google a montré un rapport de disponibilité faible dans les expériences précédemment effectuées, nous avons développé un connecteur capable de traduire les requêtes sur les Services Web de Google et MSN. Ces services sont semblables mais non identiques (différents documents WSDL). Les résultats de l'expérience utilisant la réplication active sont présentés sur la Figure 5-11. Le Service Web générique ciblé a un rapport de disponibilité égal à 100% pour ce connecteur. Ce niveau de disponibilité est atteint grâce à la disponibilité intrinsèque de MSN comparée à celle du Service Web de Google.

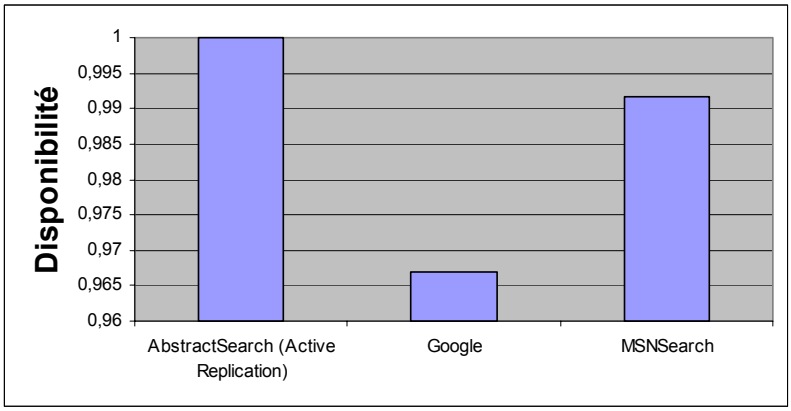

Figure 5-11: Réplication Active avec un service abstrait

Les expériences avec la stratégie de réplication passive de base montre que 6% des requêtes ont été redirigées sur le serveur MSN à cause d'une indisponibilité du serveur Google considéré comme primaire, ceci étant naturellement totalement transparent pour le client.

Nous avons pu créer ici un service abstrait de « moteur de recherches » capable d'interroger différents services opérationnels équivalents et concurrents. Cette opportunité offerte par les connecteurs s'appuie sur cette particularité intéressante du Net : la redondance inhérente des ressources qui s'y trouvent !

5.8 Cas d'étude sur une application orientée services

La section qui suit permet de mettre en évidence l'utilité des connecteurs dans une application orientée services. Nous avons créé, ici, une application assez simple dans laquelle nous avons injecté des fautes issues du modèle présenté dans le chapitre 1. Ce cas d'école va nous permettre de voir comment, à l'aide des connecteurs, on peut détecter ces fautes et les tolérer.

5.8.1 Objectif et Scénario

L'objectif de cette section est de montrer l'utilité des connecteurs lorsqu'ils sont insérés dans une application orientée services. Pour cela, nous nous sommes servis du Service Web composite de la calculatrice. Ce service tout à fait élémentaire ne sert que dans le but de tester différents connecteurs en présence de fautes. Il a été créé afin de pouvoir maîtriser dans sa globalité tous les nœuds utilisés pour faire fonctionner cette application orientée services.

Il analyse une chaîne de caractères reçue en paramètre d'entrée représentant le calcul à effectuer et appelle les services unitaires agrégés dans sa composition tels que l'addition, la soustraction ou la division ou d'autres services composites comme la multiplication qui réalise son calcul en interrogeant régulièrement le service de l'addition (cf. Figure 5-12).

Nous avons demandé au service de la calculatrice de nous fournir le résultat d'un calcul simple faisant intervenir tous les opérateurs :

5+4*2-6/3 =11.

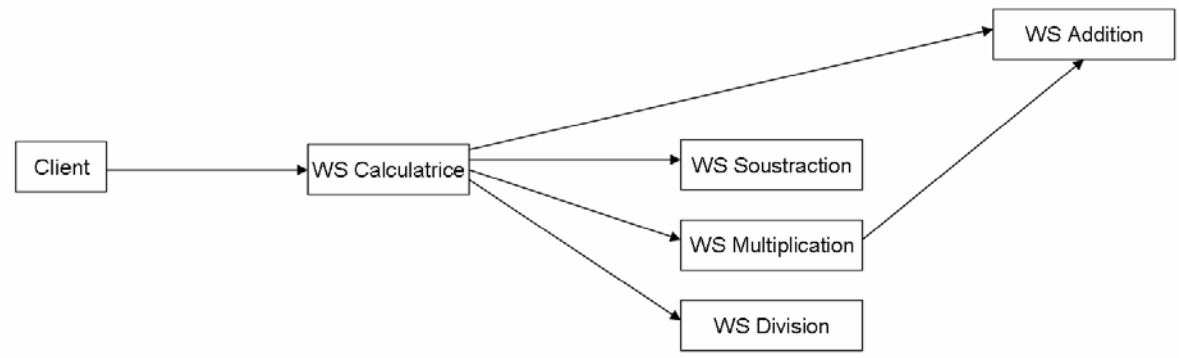

Figure 5-12: Application orientée services

Ce service fonctionne de manière séquentielle en envoyant d'abord une requête au service de la multiplication puis à celui de la division puis aux opérations moins prioritaires.

5.8.2 Injection de fautes

Nous avons intentionnellement injecté des fautes de manières aléatoires sur ces différents services unitaires (cf. Figure 5-13). Ainsi, nous provoquons des défaillances logicielles classiques qui pourraient survenir dans une exécution standard :

- ***Défaillances en valeur :*** elles sont produites sur le service qui effectue à certains moments, une opération incorrecte. Ce type d'erreur peut survenir sur un service malveillant ou mal conçu. Etant donné qu'aucune information non fonctionnelle n'est fournie dans le contrat de service, ce problème se produit lorsqu'un client a une trop grande confiance en un prestataire. Ici, le service défaillant est celui de l'addition qui effectue par moment une soustraction au lieu d'une addition. Ces défaillances impactent les services de la multiplication et de la calculatrice ainsi que de tous les clients s'appuyant sur un de ces deux prestataires.
- ***Défaillances du service :*** Ces défaillances correspondent à une exception levée de manière aléatoire et qui est retournée au client. Ce type de problème se produit fréquemment suite à l'activation d'une erreur. Le service de la soustraction est la cible de ce type de problème dans nos expériences.
- ***Gel du service :*** le gel du service est provoqué de façon aléatoire sur un service qui ne retourne, à certains moments, aucune réponse dans la fenêtre de temps autorisé. Le service de la division est la cible de ce type de problème dans nos expériences.

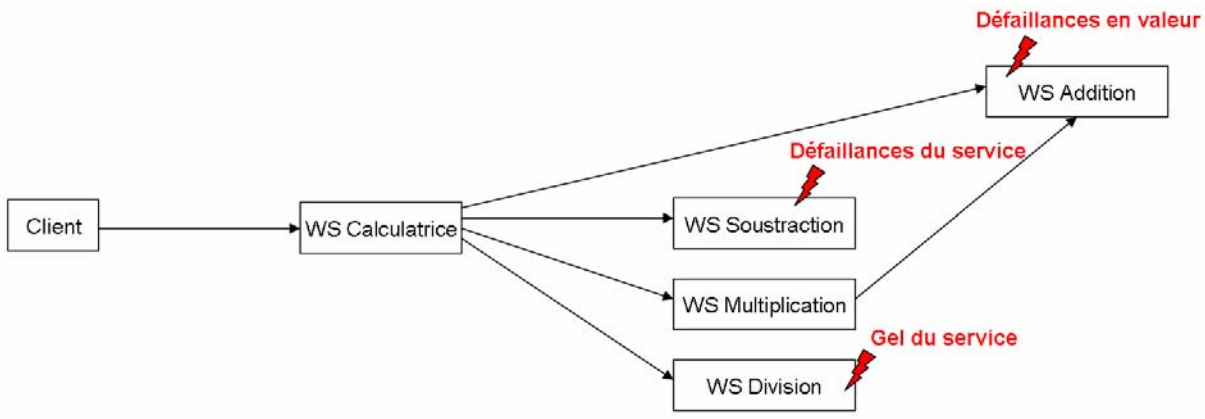

Figure 5-13: Le Service Web composite de la calculatrice

Nous avons testé cette architecture orientée services sur plus de 1000 requêtes. Les résultats obtenus sont présentés sur le tableau ci-dessous :

	Réponses retournées	Valeur Correcte	Défaillance en valeur	Défaillance du service	Gel du service	Nombre de requêtes envoyées
Sans Connecteur	808	511	297	86	106	1000

Ces expériences nous permettent de constater que nous obtenons une faible disponibilité du service (0,808) en comptant les erreurs en valeurs produites (0,511 dans le cas contraire). Cela signifie que le client ne reçoit qu'une réponse valide sur deux provenant de la calculatrice en supposant qu'il ait pu déceler les erreurs en valeur.

Dans cet exemple, ces informations sont recueillies par un client qui a préféré s'assurer que le service qu'il souhaitait utiliser était assez sûr avant de l'intégrer dans une application plus critique. La qualité du service de calculatrice sur le plan de la disponibilité étant tout à fait inacceptable, son prestataire est dans l'obligation d'améliorer son service. Le problème, c'est que les informations recueillies ne lui permettent pas de savoir où peuvent se trouver les erreurs. En effet, celles-ci peuvent se situer soit :

- dans son propre service (WS Calculatrice), dans l'implémentation de la composition des services.
- dans un ou plusieurs services unitaires agrégés dans la composition de son service.

5.8.3 Mise en place des connecteurs de surveillance

Nous avons voulu effectuer une autre série de tests en insérant cette fois-ci des connecteurs de surveillances sur chaque liaison comme illustré sur la Figure 5-14. Ces connecteurs sont générés à partir de programmes DeWeL dans lesquels aucun mécanisme de recouvrement ni de pré ou post traitements n'est défini par l'utilisateur. Leur seule fonction est donc de monitorer et de diagnostiquer le service qu'ils ciblent. Ces connecteurs de surveillance sont listés ci-dessous :

- Le connecteur ***C1*** observe le comportement du service de la calculatrice.
- Le connecteur ***C2*** observe le comportement du service d'addition.
- Le connecteur ***C3*** observe le comportement du service de la soustraction.
- Le connecteur ***C4*** observe le comportement du service de la multiplication.
- Le connecteur ***C5*** observe le comportement du service de la division.

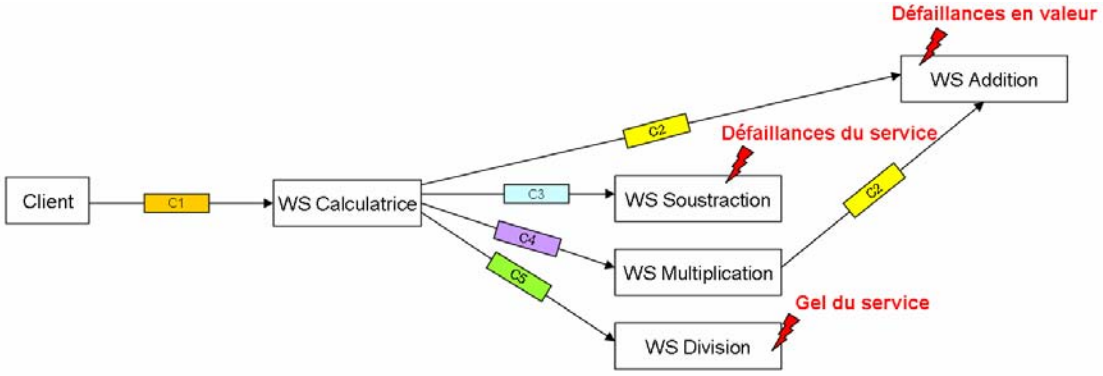

Figure 5-14: Utilisation des connecteurs sur le service composite

Nous avons testé cette configuration en envoyant 1008 requêtes. Les résultats obtenus sont présentés ci-dessous :

	Réponses retournées	Valeur Correcte	Défaillance en valeur	Défaillance du service	Gel du service	Nombre de requêtes envoyées
Client	794	529	265	214	0	1008
C1	794			214	0	1008
C2	4914			0	0	4914
C3	832			111	0	943
C4	1008			0	0	1008
C5	905			0	103	1008

Un des principaux intérêts des connecteurs, dans ce cas, est de pouvoir détecter les services incorrects. Ainsi, les connecteurs C3 et C5 ont permis d'identifier respectivement que les services de la soustraction et de la division commettent fréquemment des erreurs entraînant une indisponibilité du service. L'insertion de post-traitements dans les connecteurs et surtout dans celui de l'addition aurait permis de détecter les défaillances en valeur produites par celui-ci en vérifiant par exemple que lorsque deux nombres positifs sont envoyés en entrée, le résultat est supérieur au deux.

Nous venons de voir ici la première utilité des connecteurs : permettre à un prestataire ou un client de détecter quels sont les services défaillants dans son application orientée services.

5.8.4 Mise en place des connecteurs de surveillance et de tolérance aux fautes

La phase précédente nous ayant permis de détecter les services défaillants (l'addition, la soustraction et la division), nous avons inséré dans les connecteurs déjà mis en place (ici, C2, C3 et C5) un mécanisme de recouvrement approprié pour pouvoir tolérer les défaillances observées.

Dans notre dernière série de tests exposée sur la Figure 5-15, nous avons voulu tester ces différents mécanismes de recouvrement et ainsi analyser l'efficacité des connecteurs de surveillance et de tolérance aux fautes mis en place :

- Le connecteur ***C2*** active le mode « VotingReplication » pour tolérer les erreurs en valeur sur le service de l'addition. Celui-ci utilise deux répliques supplémentaires pour pouvoir effectuer le vote. Contrairement, à la première réplique, aucune faute n'est injectée dans les deux autres. Ceci vaut également pour les autres répliques insérées dans les différents mécanismes de recouvrement mis en place dans les connecteurs présentés ci-dessous.
- Le connecteur ***C3*** active le mode « BasicReplication » pour tolérer les levées d'exception du service de la soustraction. Il utilise une réplique de secours pour pouvoir réaliser ce recouvrement.
- Le connecteur ***C5*** active le mode « ActiveReplication » pour tolérer les gels du service de la division. Il utilise une réplique secondaire pour pouvoir réaliser ce recouvrement.

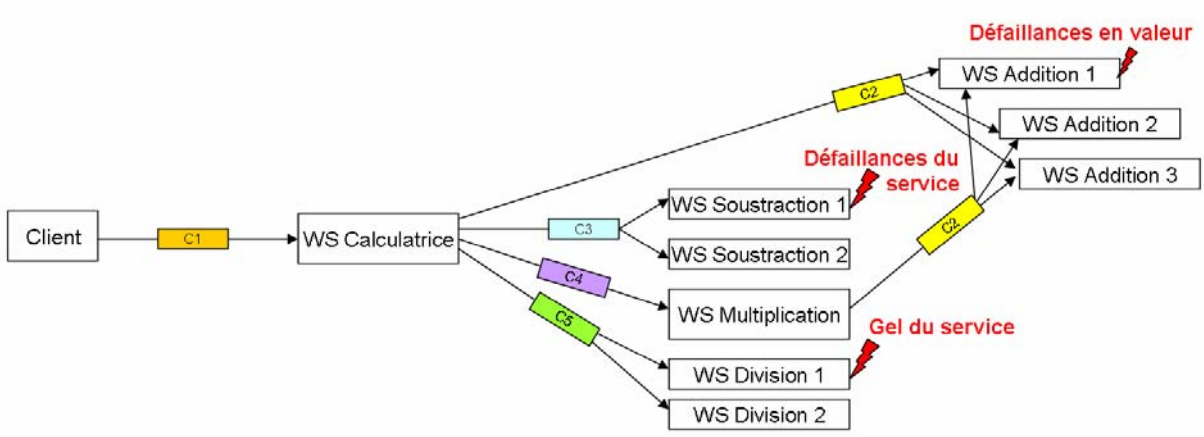

Figure 5-15: Utilisation des connecteurs avec recouvrement sur le service composite

Nous avons testé cette nouvelle configuration en envoyant 1019 requêtes. Le tableau ci-dessous représente les résultats obtenus :

	Valeur correcte	Recouvrement en valeur	Recouvrement des défaillances de service	Recouvrement des gels de service	Nombre de requêtes envoyées
Client	1019/1019				1019
C2 (VotingReplication)		554/554	0	0	5095
C3 (BasicReplication)		0	139/139	0	1019
C5 (ActiveReplication)		0	0	106/106	1019

Au final, toutes les fautes injectées ont été détectées et tolérées. 799 recouvrements ont été nécessaires pour permettre à cette architecture orientée services d'avoir une disponibilité de 100% sans aucune erreur en valeur.

Dans ce cas d'école, nous avons voulu nous mettre dans la peau d'un prestataire désireux de fournir un service sûr de fonctionnement à ses clients. La mise en place d'une composition de services est un processus complexe, nécessitant de prendre en compte de nombreux paramètres pour réaliser la séquence d'opérations cohérentes entre chaque service agrégé. Ce processus est d'autant plus complexe lorsque le prestataire ne possède aucune caractéristique non-fonctionnelle et opérationnelle sur les services qu'il utilise. L'ajout de connecteurs dans la composition de service permet au prestataire de créer des zones d'observation et de confinement d'erreurs dans lesquelles le prestataire peut adapter la sûreté de fonctionnement des services unitaires qu'il utilise en fonction de la criticité globale de son application.

5.9 Récapitulatif

Pour évaluer la robustesse d'une plate-forme, la technique la plus couramment utilisée est l'injection de fautes. Pour ce faire, une étude détaillée du modèle de fautes et des modes de défaillance en résultant doit être effectuée. Elle a été présentée dans le chapitre 1.

Dans nos travaux, nous nous sommes essentiellement concentrés sur des fautes provenant des prestataires afin de pouvoir vérifier l'efficacité de nos mécanismes de recouvrement. Les résultats obtenus lors des différentes expériences effectuées dans ce chapitre ont permis de justifier l'intérêt d'insérer dans la communication client-prestataire un connecteur capable de surveiller et contrôler l'activité d'un Service Web au sein d'une application donnée.

Nous avons pu voir dans un premier temps les capacités de DeWeL en tant que générateur de code. Nous avons constaté que le code généré était 30 fois plus petit que son équivalent en C. De plus, en raison des importantes restrictions imposées au langage, le compilateur ne comportant au final qu'un nombre restreint de vérification à effectuer par rapport à un compilateur standard, devient assez simple à implémenter. Ces deux facteurs permettent d'avoir un effet très positif sur le taux d'erreur des connecteurs générés. Cependant, bien que de nombreux tests et vérifications aient pu être effectués, une étude sur la robustesse du langage doit être étendue.

Les expériences menées sur la performance de la plate-forme IWSD ont permis de mettre en évidence que cette solution était comparable à des intercepteurs classiques possibles (par exemple, un serveur d'application Tomcat et le moteur SOAP Axis) en fournissant toutefois des fonctionnalités avancées d'observation et de diagnostic sur les services ciblés.

Les autres expériences réalisées pour évaluer les performances des connecteurs ont permis de constater que le surcoût d'exécution engendré par les stratégies de recouvrement selon le type des services (avec ou sans état) variait entre 3% et 5%. Bien sûr ces résultats ont été obtenus sur une application faisant transiter des messages de taille fixe. Une étude de sensibilité sur différents paramètres (tailles des messages, nombre de répliques utilisées, fréquences de défaillance des répliques, etc.) permettrait de voir de façon plus fine quels facteurs impactent réellement sur le surcoût d'exécution, au-delà de l'application elle-même.

Différents connecteurs de surveillance ont été mis en place pour observer des services pris sur le Net. Ces connecteurs ont permis d'identifier quels services bénéficieraient le plus d'actions de tolérance aux fautes pour augmenter leur disponibilité. Nous avons ainsi réalisé des connecteurs capables de vérifier, entre autres, une notion essentielle présentée dans ce manuscrit, à savoir, un connecteur pour un service abstrait de moteur de recherche. Ce connecteur est capable d'interroger deux services équivalents ayant pourtant des contrats de services différents. Ce connecteur a permis d'augmenter considérablement la disponibilité de ce type de service en recouvrant les erreurs récurrentes du service Web de Google.

Le cas d'école exposé en fin de cette section nous a permis de nous mettre dans la peau d'un développeur d'architecture orientée services et de voir l'intérêt que peuvent apporter les connecteurs dans la mise en place de processus complexes tel que la réalisation d'une composition de services. Ces derniers permettent de créer des zones d'observation et de confinement d'erreurs permettant d'améliorer nettement la sûreté globale de l'application.

En définitive, ces tests ont permis d'apprécier les trois caractéristiques principales apportées par les connecteurs : le diagnostic, la détection et le recouvrement d'erreur.

6 Conclusion et Perspectives

Dans nos travaux de thèse, nous nous sommes particulièrement intéressés à la mise en œuvre de la tolérance aux fautes dans les systèmes distribués à base de Services Web de façon transparente et adaptable. Nous avons montré la nécessité de disposer de composants de type « connecteurs » capable d'améliorer la robustesse d'application semi-critique élaborée à partir de services disponibles sur le Web. Les applications développées à partir de services existants reposent sur des standards comme SOAP, WSDL et UDDI qui constituent une infrastructure de base permettant la mise en oeuvre d'architectures orientées services. Ces derniers sont sans exigence poussée de fiabilité, de sécurité ou de gestion transactionnelle. Cette base est donc insuffisante pour la création de processus métiers critiques qui est, à terme, la cible des technologies de Services Web. En effet, il faut bien se rendre compte ici, que l'on parle d'application planétaire reposant totalement sur des ressources Web qui peuvent être à tout moment déplacées, défaillantes ou supprimées. Les problèmes de disponibilité et d'intégrité se retrouvent donc au premier plan dans ce type d'architecture.

De plus, lorsque nous disons qu'un document WSDL décrit un service, nous faisons en fait un raccourci : en réalité, un document WSDL décrit seulement l'interface d'un service. Ce que le service fait, au-delà des échanges de messages, et la façon dont il le fait, ne sont pas formalisés dans le document WSDL. En fait, deux éléments essentiels manquent à l'appel :

- la description des fonctions du service;
- la description des caractéristiques opérationnelles (d'une occurrence) du service.

En l'état actuel des choses, aucune technologie de description formelle des fonctions et des caractéristiques opérationnelles d'un service n'est disponible. Il reste deux moyens classiques pour l'utilisateur du service :

- soit une documentation détaillée du développement est accessible ;
- soit le code source d'une des implémentations du service est disponible.

Les limites de la première approche sont bien connues. Dans la pratique, la lecture de la documentation s'accompagne souvent de sessions de formation et d'échanges informels entre les clients, les concepteurs et les experts. Cela est viable au cas par cas, à l'intérieur de l'organisation et dans les partenariats bien établis, mais le problème de l'accès au bon niveau d'information pour le plus grand nombre d'utilisateurs reste ouvert.

La deuxième approche est viable pour les services internes à l'organisation, mais elle n'est pas envisageable lorsqu'il s'agit d'utiliser un service externe. Cette interdiction objective va cependant forcer le changement d'habitudes bien ancrées et favoriser la pratique de la réutilisation du logiciel en mode « boîte transparente ». L'utilisation croissante de composants en source libre va clairement dans ce sens. Cependant, force est de constater que ces deux solutions ne passent pas à l'échelle.

En outre, il faut éviter de tomber dans le piège qui consiste à considérer que la distinction entre contrat et implémentation du service correspond à la distinction traditionnelle entre analyse fonctionnelle et implémentation, entre le métier et la technique. Le paradoxe que représente l'implémentation d'un service au moyen d'autres services dont on ne connaît pas

l'implémentation n'est maîtrisable sur grande échelle que si les services utilisés sont dotés de spécifications opérationnelles, en plus des spécifications fonctionnelles et des spécifications d'interface (ces dernières étant les seules spécifications formelles à la date d'aujourd'hui). Ces spécifications opérationnelles caractérisent le comportement d'un service en exécution.

Une partie des caractéristiques opérationnelles du service, comme les niveaux de fiabilité de l'échange, de sécurité et de gestion des transactions, est exprimée par les formalismes développés dans le cadre de ces technologies d'infrastructure. Il reste que l'expression des autres caractéristiques, comme la performance, la disponibilité, etc., n'est pas formalisée aujourd'hui. Cela peut aller, par exemple, de l'engagement de disponibilité d'un service à des propriétés toutes simples, mais dont l'expression est très utile, comme le délai maximal d'attente (timeout) que l'on s'autorise dans une interaction. Les informations fournies par le moniteur de surveillance de la plate-forme IWSD sur les différents services ciblés par les connecteurs peuvent aider dans ce but.

Ainsi, dans ce mémoire, nous avons analysé les systèmes à base de Services Web, en s'appuyant sur un modèle de fautes adapté et les modes de défaillances qui en résultent. A partir de cette étude, nous avons conçu la plate-forme IWSD capable d'améliorer la détection et de mettre en place les mécanismes de recouvrement appropriés. Nous avons développé des techniques visant à empêcher la propagation d'erreur entre le prestataire et le client et ainsi créé une zone de confinement d'erreur séparée de l'application. Cette zone de confinement d'erreur est contrôlée par un langage « durci » nommé DeWeL pour la tolérance aux fautes et implémenté sous forme de connecteurs. Dans une telle architecture, l'arrêt inopiné, le gel de service ou la défaillance en valeur sont détectés par le code applicatif défini et inséré par un utilisateur, lui permettant d'exécuter des actions de recouvrement d'erreurs pour tolérer la défaillance d'un Service Web COTS.

L'un des principaux apports de la thèse est la notion de « connecteur » spécialisable par l'utilisation du point de vue de la tolérance aux fautes. En effet, les connecteurs viennent s'intégrer au cœur des architectures orientées services en se fixant de manière non-intrusive sur chaque liaison client-prestataire. Véritables médiateurs, ils permettent de fournir aux clients des zones d'observation et de confinement d'erreur spécifiques à chaque Service Web ciblé et à chaque contexte d'application donné en fonction de leurs exigences en matière de sûreté de fonctionnement. Ils permettent ainsi d'augmenter la confiance d'un client envers un prestataire non-fiable ou supposé fiable.

Pour améliorer la sûreté des applications orientées services, les connecteurs créés doivent être, idéalement, exempts de fautes. La description du connecteur réalisé par l'utilisateur doit donc comporter les abstractions et notations appropriées capables de lui permettre d'effectuer des actions de tolérance aux fautes spécifiques à son application (assertions exécutables, détection et mise en place de stratégie de recouvrement d'erreur) sans injecter de fautes dans le connecteur. De nombreuses restrictions reconnues, y compris au niveau de standard de développement de logiciels critiques ont été appliquées dans le langage DeWeL, pour garantir la sûreté du connecteur créé sans toutefois compromettre son expressivité.

La plate-forme IWSD fournit le support d'exécution et de gestion des connecteurs. Le serveur de gestion permet à un client, un prestataire ou un tiers de s'enregistrer comme utilisateur afin qu'il puisse créer ses propres connecteurs. Le serveur d'exécution peut être considéré comme une machine virtuelle en charge d'activer le connecteur approprié sur réception de la requête cliente. Il est implémenté en mode duplex pour assurer la sûreté de fonctionnement de l'architecture. Le moniteur de surveillance permet de faire un diagnostic précis des connecteurs créés et fournit également des informations opérationnelles sur les points d'accès

(ou services) utilisés. Ces informations permettent de connaître par exemple la disponibilité courante, la fréquence des défaillances ou le temps de réponse moyen des répliques de service utilisées. Elles sont capitales pour un utilisateur souhaitant adapter au mieux la stratégie de recouvrement dans un connecteur.

Ces stratégies de recouvrement sont simplement déclarées et paramétrées dans un programme DeWeL. Elles sont précisées par l'utilisateur en fonction d'hypothèses faites sur le service (service à silence sur défaillance, service à défaillance en valeur, déterministe ou non, …etc), du type de service (avec ou sans état), du nombre de répliques et de leurs taux de défaillance, du type et du contexte de l'application ciblée. Les différentes fonctions définies en DeWeL permettent de mettre en œuvre trois modes de recouvrement : la réplication passive (*BasicReplication, StatefulReplication* et *LogBasedReplication*), la réplication active sans vote (*ActiveReplication*) et la réplication active avec vote (*VotingReplication*) en fonction des hypothèses précédentes. Parmi elles, certaines nécessitent une collaboration du prestataire et/ou du client. Ainsi, des fonctions de gestions d'état doivent être fournies par le prestataire au connecteur pour qu'il réalise la *StatefulReplication*. La mise en place d'une session dans le mode « LogBasedReplication » doit être instaurée par le client en utilisant les fonctions « start_session » et « end_session » précisées dans le contrat du connecteur. De même, des notions comme la garantie de livraison ou l'ordonnancement total des messages doivent être prises en compte pour assurer les mécanismes à base de réplication active (*ActiveReplication* et *VotingReplication*).

Pour assurer ces mécanismes de recouvrement, le connecteur doit bien sûr disposer de plusieurs répliques. Là aussi, les travaux de cette thèse apportent une notion intéressante avec la création de Services Web Abstraits. Ce type de service n'a pas de réalité fonctionnelle mais est capable de contacter différents services concrets équivalents sur le Net et de profiter ainsi de la redondance inhérente des ressources fournies sur ce support. Ainsi, les connecteurs peuvent tirer avantage de la diversification de différentes répliques identiques fournies le plus souvent par le même prestataire (comme les six répliques fournies par le prestataire d'Amazon) ou de services concurrents équivalents.

Enfin, l'ensemble des mécanismes de tolérance aux fautes définis par l'utilisateur peut être consigné dans le contrat WSDL du connecteur. Ce contrat permet ainsi d'ajouter une vue non-fonctionnelle et opérationnelle au contrat de base et ainsi d'augmenter la confiance d'un client.

Nos travaux ouvrent ainsi plusieurs champs d'investigation dans un domaine essentiel, celui des architectures orientées services critiques.

Tout d'abord, la validation et la vérification de DeWeL et de ses outils permettraient d'avoir des mesures plus fines sur la robustesse et l'expressivité du langage. Des évolutions pourraient être également apportées au langage (prise en compte de la gestion de la sécurité et des transactions).

La standardisation des fonctions de gestion d'état pour les services qui le nécessitent est également une voie possible d'exploration. Celle-ci permettrait de pouvoir récupérer plus facilement et de manière plus fine l'état d'un client particulier et ainsi d'améliorer nettement les performances des mécanismes de recouvrement mis en jeux. Le traitement de la gestion de l'état du prestataire, la mise en place des points d'arrêt, la sauvegarde des requêtes, la reconstitution des répliques sont autant de points qui pourraient être spécifiés et négociés à travers un contrat de service avant d'être réalisés dans un connecteur.

Le passage à l'échelle reste également un enjeu majeur sur ce type d'architecture. Celle-ci peut être atteinte à l'aide d'un nombre important de machines et donc par l'utilisation de plates-formes tel que Grid 5000 ou planetLab. Un réel cas d'étude créé à partir d'une application semi-critique à l'échelle planétaire permettrait de valider de façon plus rigoureuse les résultats expérimentaux. De part la jeunesse des Services Web, de tels cas d'étude ne sont pas encore disponibles facilement. Une solution possible serait d'en réaliser une. Par exemple, on peut imaginer une application de type e-rescue à l'aide de différents Services Web équivalents comme MapPoint et Mappy capables de donner un itinéraire précis suite à un appel d'urgence. Ce type d'application permettrait de réaliser des tests plus poussés en se servant de grilles de calcul.

Enfin, profiter de la diversification naturelle des ressources du Web semble être une des voies les plus prometteuses. L'étude que nous avons menée sur la création de Services Web abstraits permet de créer un connecteur capable d'interroger des services mettant en oeuvre des spécifications fonctionnelles équivalentes bien qu'ayant des interfaces différentes. Il semble tout à fait opportun d'automatiser ce processus en insérant de la sémantique au contrat de service. Le seul programme d'envergure dans ce domaine est celui mené par l'activité Semantic Web du W3C, avec la formalisation d'un langage de description d'« ontologies » (OWL Web Ontology Language) pour caractériser les ressources accessibles sur le Web. La définition d'un langage qui permet de rédiger des « ontologies », propres aux différents secteurs économiques, compréhensibles par programme, n'est qu'un premier pas vers l'automatisation de la création de Services Web abstraits.

A.1 Le Langage DeWeL

Ce chapitre peut être vu comme le manuel de référence du langage DeWeL. Il liste les constructions du langage, donne leur syntaxe précise et une sémantique informelle.

Notation

La syntaxe du langage DeWeL est donnée sous la forme d'une notation BNF. Les symboles terminaux sont écrits de la façon suivante : **terminal**. Les symboles *non-terminaux* sont mis en *italic*. La barre verticale | souligne une alternative dans la règle. Les parenthèses (…) représentent un groupe. Les parenthèses avec un signe étoilé $(\ldots)^*$ précisent que l'occurrence de ce groupe peut être de zéro, un ou plusieurs. Les parenthèses avec un signe plus $(\ldots)^+$ montrent qu'il peut y avoir une ou plusieurs occurrences du groupe sélectionné. Les parenthèses avec un point d'interrogation $(\ldots)^?$ signifient que le groupe est optionnel.

Disponibilité

La distribution complète du langage DeWeL (incluant le compilateur) et de la plate-forme IWSD est librement disponible sur le Net à l'adresse suivante:
http://www.laas.fr/~nsalatge/prototypes/IWSD/IWSD.tar.gz.

A.1.1 Introduction

DeWeL est un langage dédié permettant de spécifier des propriétés non-fonctionnelles à l'interface du Service Web. Concrètement un Service Web peut être décrit par un ensemble d'opérations. Le point d'entrée à la spécification DeWeL est la définition du contrat WSDL.

A.1.2 Convention lexicale

Les Blancs

Les caractères suivants sont considérés comme des blancs : espace, retour chariot et tabulation. Les blancs séparent les identifiants, les littéraux ou mots clés adjacents.

Les Commentaires

Les commentaires sont identiques à ceux du C. Ils commencent avec les caractères /* et finissent avec les caractères */. Le style de commentaire C++ peut aussi être utilisé ; tous les caractères écrits depuis les caractères // jusqu'à la fin de la ligne sont considérés comme une partie d'un commentaire. Les commentaires DeWeL sont insérés par la suite dans le contrat WSDL du connecteur.

Les identifiants

Les identifiants sont une séquence de lettres, chiffres et _ (le caractère underscore) commençant par une lettre. Une lettre peut être majuscule ou minuscule établie à partir du code ASCII. L'implémentation courante ne place pas de limite sur le nombre de caractères d'un identifiant.

<identifier> ::= <letter> (<letter> | <digit> | _)*
<letter> ::= A..Z | a..z

Les littéraux
Les littéraux dans DeWeL peuvent être de plusieurs sortes comme décrit ci-dessous :

<literal> ::= <integer literal> | <floating-point literal> | <boolean literal> |
<string literal> | <uri literal> | <gregorian date literal> |
<duration literal> | <binary literal>

Les littéraux entiers
Un entier est une séquence d'un ou plusieurs chiffres, optionnellement précédée par le signe moins. Dans l'implémentation actuelle du langage DeWeL, seuls les entiers de base 10 sont considérés:

<integer literal> ::= 0 | (<sign>)$^{?}$ <non zero digit> (<digits>)$^{?}$
<digits> ::= <digit> | <digits> <digit>
<digit> ::= 0 | <non zero digit>
<non zero digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<sign> ::= + | -

Les littéraux flottants
Un nombre flottant est représenté par une mantisse suivie de la lettre E (e est autorisé aussi) suivie d'un chiffre représentant un exposant. La mantisse doit être un nombre décimal; l'exposant doit être un nombre entier de type entier. Le E et l'exposant sont optionnels.

<floating-point literal> ::= (<sign>)$^{?}$ <digits> . (<digits>)$^{?}$ (<exponent part>)$^{?}$
(<float type suffix>)$^{?}$ <digits> (<exponent part>)$^{?}$ (<float type suffix>)$^{?}$
<exponent part> ::= <exponent indicator> <signed integer>
<exponent indicator> ::= e | E
<signed integer> ::= (<sign>)$^{?}$ <digits>
<float type suffix> ::= f | F | d | D

Les littéraux booléens
Le type *boolean* n'a que deux valeurs possibles, représentées par les littéraux **true** ou **false**.

<boolean literal> ::= **true** | **false**

Les littéraux chaînes de caractères
<string literal> ::= " <string characters>?"
<string characters> ::= <string character> | <string characters> <string character>
<string character> ::= <input character> except " and \
<single character> ::= <input character> except ' and \
<input character> correspond à tous les caractères UNICODE existant.

Les littéraux pour les URI
Pour des raisons de clarté, les règles lexicales de <uri literal> ne sont pas fournies dans ce manuscrit puisqu'elles correspondent rigoureusement aux spécifications IETF (RFC 2396 et RFC 2732).

Les littéraux pour les dates grégoriennes
<gregorian date literal> ::= <string literal>

Dans l'implémentation courante, les dates grégoriennes comprises par le langage sont écrites sous forme de chaînes de caractères comprises dans un champ lexical bien précis (cf. tableau des types).

Les littéraux pour la mesure du temps
<duration literal> ::= <string literal>

Dans l'implémentation courante, les mesures d'intervalle de temps sont écrites sous forme de chaînes de caractères comprises dans un champ lexical bien précis (cf. tableau des types, annexe A.1.3.4.1).

Les littéraux pour les nombres binaires
<binary literal> ::= <string literal>

Dans l'implémentation courante, les mesures d'intervalles de temps sont écrites sous forme de chaînes de caractères comprises dans un champ lexical bien précis (cf. tableau des types, annexe A.1.3.4.1).

Les Opérateurs
Les tokens suivants sont les opérateurs DeWeL :

```
=   +   -   *   /   %
==  !=  <   >   >=  <=
&&  ||  !
```

On peut remarquer qu'une séquence de caractères, telle que != ou >= est lue comme un unique token.

Les mots clés
Les identifiants ci-dessous sont les mots clés réservés du langage :

SpecificFaultToleranceConnector	**if**
extend	**else**
memory	**switch**
RecoveryStategy	**break**
Pre-Processing	**case**
Post-Processing	**foreach**
CommunicationException	**in**
ServiceException	**optional**
import	**with**
return	**raise**

Les ambiguïtés
Les ambiguïtés lexicales sont résolues selon la règle « longest match » : Quand une séquence de caractères peut être décomposée en tokens (ou unité lexicale) de plusieurs façons, la décomposition résultante est celle avec le premier token le plus long.

A.1.3 Spécification de DeWeL

La Figure 6-1 décrit la syntaxe BNF de la spécification du connecteur. Une spécification d'un connecteur de service est introduite par le mot clé « **SpecificFaultTolerantConnector** ». L'identifiant qui lui succède est le nom du connecteur défini. Cette spécification doit obligatoirement hériter d'un contrat WSDL de base à l'aide du mot « **extend** ». Ainsi, la compilation du script permettra de vérifier si les instructions spécifiées par l'utilisateur ne sont pas contradictoires ou interdites pour le contrat de base. A l'intérieur de cette spécification, deux types de traitements peuvent être effectués : genericProcessing et wrappedOperationProcessing

A.1.3.1 Les traitements spécifiques (<processing>)

```
<connector> ::= ( <import schema> )*  <wrapped service>

<import schema> ::= #import <string-literal>

<wrapped service> ::= SpecificFaultTolerantConnector <identifier> extend <uri-literal>  { <processing> }
<processing> ::= <genericProcessing>  (<wrappedOperationProcessing>)*
<genericProcessing> ::= (<functionInvocation>)* (<typeRestriction>)* (<assignement>*
<wrappedOperationProcessing> ::=
            <outputParameter> <operationName> ( <inputParameters> ) { <specificProcessing> }
                (extend with <inputParameters>)?;

<specificProcessing> ::= <recoveryStrategy> <preProcessing>  <postProcessing>
                                        <communicationException> <serviceException>
<recoveryStrategy> ::= RecoveryStrategy : (<functionInvocation>)?
<preProcessing> ::= Pre-Processing : <statements>
<postProcessing> ::= Post-Processing : <statements>
<communicationException> ::= CommunicationException <identifier> : <statements>
<serviceException> ::= ServiceException <identifier> : <statements>

<operationName> ::= <identifier>
<inputParameters> ::= <parameter> (, <parameter> )?
<outputParameter> ::= <parameter> | void
<parameter> ::= <type> <identifier>
```

Figure 6-1: Syntaxe BNF de la spécification du connecteur

Les GenericProcessing :
Les genericProcessing correspondent aux traitements globaux qui seront appliqués quelle que soit l'opération appelée. Les instructions que peut effectuer l'utilisateur sont de deux sortes : « functionInvocation » et « typeRestriction », celles-ci seront détaillées par la suite.

Les wrappedOperationProcessing:
Les wrappedOperationProcessing sont les traitements spécifiques spécifiés par l'utilisateur pour chaque opération du service. Ainsi, le point d'entrée à un wrappedOperationProcessing est la signature de l'opération du connecteur. Celle-ci contient, bien évidemment, des

paramètres d'entrée et de sortie qui pourront être analysés, vérifiés ou modifiés dans la plupart des 6 sous-traitements:

- ***RecoveryStrategy***: cette zone permet à l'utilisateur de spécifier le mécanisme de recouvrement ainsi que le nombre et les répliques qu'il souhaite utiliser.
- ***Pre-Processing***: cette section est l'endroit où l'utilisateur peut effectuer des vérifications et modifications sur la requête reçue.
- ***Post-Processing***: cette section est l'endroit où l'utilisateur peut effectuer des vérifications et modifications sur la réponse correspondante.
- ***CommunicationException***: l'utilisateur peut effectuer des traitements spécifiques dans le cas où il ne peut pas joindre le service cible.
- ***ServiceException***: l'utilisateur peut effectuer des traitements spécifiques dans le cas où le service répond par un message d'erreur.

A.1.3.2 Les instructions

```
<statements> ::= (<declaration-statement>)* (<statement>)*

<statement> ::= <standard-statement>
                | <optional-statement>

<standard-statement> ::= ;
        | { <statements> }
        | <assignment> ;
        | <functionInvocation> ;
        | <methodInvocation> ;
        | <typeRestriction> ;
        | <if-statement> ;
        | <switch-statement>
        | <foreach-statement>
        | <return-statement>
        | <raise-statement>

<declaration-statement> ::= <basic-declaration> | <memory-declaration>
<basic-declaration> ::= <declaration>
<memory-declaration> ::= memory::<declaration>
<declaration>::=<type> <identifier> = expression ;

<optional-statement> ::= optional <standard-statement>

<if-statement> ::= if  <expression> <statement> ( else <statement> )?
<switch-statement> ::= switch <expression> { <cases> }
<foreach-statement> ::= foreach <type> <identifier> in <array-list> { <statements> }
<return-statement> ::= return (<functionInvocation> | <operationName>) ;
<raise-statement> ::= raise <functionInvocation> ;

<cases> ::= ( case <expression> : <statements> break ; )+ (default : <statements>)? ;
<assignment> ::= <left hand side> = <expression>
<left hand side> ::= <identifier> | <variableInvocation>
<functionInvocation> ::= <identifier> ( <argument list> )
<methodInvocation> ::= <objet> . <methodName> ( <argument list> )
<methodName> ::= <identifier>
<variableInvocation> ::= <objet> . <identifier>
<argument list> ::= <expression> | <argument list> , <expression>

<typeRestriction> ::= <type> . <facette> = <expression>
<facette> ::= <identifier>
<array-list> ::= <objet>
```

Figure 6-2: Syntaxe BNF des instructions

Les instructions en DeWeL sont multiples et se rapproche de celle du C++ :

L'invocation de fonction: <functionInvocation>
L'utilisateur peut s'il le souhaite appeler des méthodes prédéfinies par le langage. Dans l'implémentation courante, ces fonctions sont les suivantes:

- ***log*** : cette fonction permet de stocker dans un fichier les informations souhaitées par l'utilisateur tel que la requête par exemple.
- ***setTimeout***: cette fonction permet de régler le temps d'attente entre l'envoi de la requête et la réception de la réponse.
- ***getCreationDate*** : permet de récupérer la date de création du connecteur.
- ***getLastUpdate*** : permet de récupérer la dernière date de mise à jour du connecteur.
- ***getDate*** : permet de récupérer la date courante.
- ***getResponseTime*** : permet de fournir à l'utilisateur le temps de réponse du service (cette fonction retourne -1 si elle est appelée ailleurs que dans un traitement autre que Post-Processing ou ServiceException).
- ***getAverageResponseTime*** : donne le temps de réponse moyen du service (renvoi -1 si l'opération n'a jamais atteint un Post-Processing ou un Service-Exception).
- ***getCommunicationExceptionNumber*** : donne le nombre total de Communication-Exception survenu depuis le début du cycle de vie du connecteur.
- ***getServiceExceptionNumber*** : donne le nombre total de ServiceException survenu depuis le début du cycle de vie du connecteur.
- ***getAccessPointInfo*** : fournit des informations sur le point d'accès concret qui a produit la réponse.
- ***SOAPException*** : génère une exception spécifique de type SOAP.
- ***BasicReplication*** : sélectionne la réplication passive sans état comme mode de recouvrement.
- ***StatefulReplication*** : sélectionne la réplication passive avec gestion d'état comme mode de recouvrement. Le prestataire doit, dans ce cas, fournir les fonctions de gestion d'état.
- ***LogBasedReplication*** : sélectionne la réplication passive avec gestion d'état comme mode de recouvrement. Le client doit, dans ce cas, créer une session à l'aide des opérations « start_session » et « end_session » fournies dans le contrat du connecteur.
- ***ActiveReplication*** : sélectionne la réplication active sans vote comme mode de recou-vrement.
- ***VotingReplication*** : sélectionne la réplication active avec vote comme mode de recou-vrement.

D'autres fonctions peuvent bien sûr être incorporées dans le langage. Ces fonctions peuvent être insérées sous forme de librairies spécialisées.

L'appel de méthode: <methodInvocation>
L'utilisateur peut également appeler des méthodes prédéfinies sur des variables de types primitifs. Les méthodes utilisables dans l'implémentation actuelle sont précisées dans les tableaux des types (cf. annexe A.1.3.4.1).

L'appel de variable: <variableInvocation>
Lorsque l'utilisateur manipule des variables de type complexe (complexType), il peut être amené à contrôler le contenu des variables internes.

La restriction de type: <typeRestriction>
Tout comme pour les schémas, les types primitifs de DeWeL possèdent des facettes. Ce sont des attributs internes qui peuvent être redéfinis pour restreindre le champ de valeur des types utilisés dans la requête.

L'instruction <if-statement>: if <expression> <statement1> (else <statement2>) [?]
Cette expression permet de définir conditionnellement un chemin d'exécution. L'accès au chemin d'exécution défini dans <statement1> (et respectivement <statement2>) est possible au temps *t* si et seulement si l'expression <expression> est évaluée à vrai (**true**) et respectivement à faux (**false**) dans l'environnement présent au temps *t*. On peut remarquer que le traitement « **else** » peut être omis.

L'instruction <switch-statement>: switch <expression> {}
Le « **switch** » permet d'effectuer un chemin d'exécution contenu dans les cases en fonction de l'égalité de l'expression contenue dans le « **switch** » et le premier « **case** » correspondant.

L'instruction <return-statement>: return (<operationName> | <functionInocation>)
Cette instruction permet de sortir d'un traitement spécifique en envoyant soit au client soit à un prestataire un message SOAP (requête, réponse ou exception). Elle permet également de retourner un message d'erreur au client. La stratégie de recouvrement, si elle est précisée, est dans ce cas, annulée car elle peut être considérée comme non appropriée.

L'instruction <raise-statement>: raise <functionInocation>
Cette instruction permet de sortir d'un traitement spécifique en levant une exception et déclenche ainsi le mécanisme de recouvrement défini par l'utilisateur. S'il n'est pas précisé, un message d'erreur est retourné au client.

L'instruction <foreach-statement>: foreach <type> <identifier> in <array-list> { statements}
Le foreach est une instruction permettant au programmeur d'effectuer un parcours de tableau de façon sûre. En effet, le langage DeWeL interdit l'indexation de tableau. L'utilisateur ne peut atteindre un élément spécifique du tableau qu'au moyen de cette instruction.

A.1.3.3 Les expressions

Les expressions de DeWeL sont définies comme précisé dans la syntaxe de la BNF de la figure 6-3. Une expression DeWeL qui peut être complètement évaluée à la compilation est référée comme une expression statique. Autrement, cela réfère à une expression dynamique. Les identifiants dans le test d'expression conditionnelle peuvent seulement se référer à des variables internes ou des paramètres d'entée/sortie de l'opération.

```
<expression> ::= <orexpression> | <assignement>
<orexpression> ::= <andexpression>  | <orexpression> || <andexpression>
<andexpression> ::= <eqexpression>  | <andexpression> && <eqexpression>
<eqexpression> ::= <relexpression>
                    | <eqexpression> == <relexpression>
                    | <eqexpression> != relexpression
<relexpression> ::= <addexpression>
                    | <relexpression> < <addexpression>
                    | <relexpression> > <addexpression>
                    | <relexpression> <= <addexpression>
                    | <relexpression> >= <addexpression>
<addexpression> ::= <multexpression>
                     | <addexpression> + <multexpression>
                     | <addexpression> - <multexpression>
<multexpression> ::= <unaryexpression>
                    | <multexpression> * <unaryexpression>
                    | <multexpression> / <unaryexpression>
                    | <multexpression> % <unaryexpression>
<unaryexpression> ::= <primary>  | ! <unaryexpression>
<primary> ::= <functionInvocation> | <literal> | ( <expression> )  | <objet>
<objet> ::= <identifier> | <methodInvocation> | <variableInvocation>
```

Figure 6-3: Syntaxe BNF des expressions

La figure 6-4 montre les priorités et associativités relatives aux opérateurs. Les opérateurs sont ordonnés de la priorité la plus haute à la plus basse.

Opérateurs	Associativité
!	right to left
* / %	left to right
+ -	left to right
< <= > >=	left to right
== !=	left to right
&&	left to right
||	left to right

Figure 6-4: Priorité et Associativité des opérateurs pour les expressions

A.1.3.4 Les types de DeWeL

Les types de DeWeL sont multiples. Chacun de ces types correspond à un type de base du schéma XML.

```
<type> ::= <primitive type> | <reference type>
<primitive type> ::= <numeric type> | <string type> | <boolean type> | <gregorian date type>
                                    | <duration type> | <binary type> | <uri type> | <xml type>
<reference type> ::= <complex or simple type> | <array type>
<complex or simple type> ::= <complex type> | <simple type>
<complex type> ::= <type name>
<simple type> ::= <type name>
<array type> ::= <type> [ ]
<type name> ::= <identifier>

Les types numériques:
<numeric type> ::=  <integral type> | <floating-point type>
<integral type> ::=  integer | nonPositiveInteger | negativeInteger | long | int | short | byte |
                     nonNegativeInteger | unsignedLong | unsignedInt | unsignedShort |
                     unsignedByte | positiveInteger
<floating-point type> ::= decimal | float | double

Les types chaînes de caractères:
<string type> ::=  string | normalizedString | token | language | Name | NCName | QName

Le type booléen
<boolean type> ::= boolean

Les types : Les dates grégoriennes:
<gregorian date type> ::= gYear | gMonthDay | gDay | gMonth | gYearMonth

Les types pour la mesure du temps:
<duration type> ::= duration | dateTime | time | date

Les types binaires:
<binary type> ::= base64Binary | hexBinary

Le type pour les URI:
<uri type> ::= anyURI

Les types hérités de XML 1.0:
<xml type> ::= NOTATION | NMTOKEN | NMTOKENS | ID | IDREF |
               IDREFS | ENTITY | ENTITIES
```

Figure 6-5: Syntaxe BNF pour les types de DeWeL

Les facettes de DeWeL :

Tout comme pour les schémas XML, le tableau suivant donne la liste des facettes associées à chaque type de base. Les types sont regroupés par série ayant le même lot de facettes. Les cellules barrées indiquent les zones où aucune facette n'est autorisée. Les relations facette-type sont reprises dans la description détaillée de chaque type DeWeL.

<table>
<tr><th rowspan="2">Type de base</th><th colspan="4">Facette autorisée</th></tr>
<tr><th>Facette du type de base</th><th colspan="3">Facette commune à plusieurs catégories de type de base</th></tr>
<tr><td>boolean</td><td></td><td rowspan="31"></td><td rowspan="31">enumeration</td><td rowspan="31">whiteSpace
pattern</td></tr>
<tr><td>anyURI</td><td rowspan="17">length
minLength
maxLength</td></tr>
<tr><td>base64Binary</td></tr>
<tr><td>ENTITY</td></tr>
<tr><td>ENTITIES</td></tr>
<tr><td>hexBinary</td></tr>
<tr><td>ID</td></tr>
<tr><td>IDREF</td></tr>
<tr><td>IDREFS</td></tr>
<tr><td>language</td></tr>
<tr><td>Name</td></tr>
<tr><td>NCName</td></tr>
<tr><td>NMTOKEN</td></tr>
<tr><td>NMTOKENS</td></tr>
<tr><td>normalizedString</td></tr>
<tr><td>NOTATION</td></tr>
<tr><td>QName</td></tr>
<tr><td>string</td></tr>
<tr><td>token</td></tr>
<tr><td>byte</td><td rowspan="14">totalDigits
fractionDigits</td><td rowspan="14">maxInclusive
maxExclusive
minInclusive
minExclusive</td></tr>
<tr><td>decimal</td></tr>
<tr><td>int</td></tr>
<tr><td>integer</td></tr>
<tr><td>long</td></tr>
<tr><td>negativeInteger</td></tr>
<tr><td>nonPositiveInteger</td></tr>
<tr><td>nonNegativeInteger</td></tr>
<tr><td>positiveInteger</td></tr>
<tr><td>short</td></tr>
<tr><td>unsigedByte</td></tr>
<tr><td>unsignedInt</td></tr>
<tr><td>unsignedLong</td></tr>
<tr><td>unsignedShort</td></tr>
<tr><td>date</td><td rowspan="14"></td></tr>
<tr><td>dateTime</td></tr>
<tr><td>double</td></tr>
<tr><td>duration</td></tr>
<tr><td>float</td></tr>
<tr><td>gDay</td></tr>
<tr><td>gMonth</td></tr>
<tr><td>gMonthDay</td></tr>
<tr><td>gYear</td></tr>
<tr><td>gYearMonth</td></tr>
<tr><td>time</td></tr>
</table>

Les facettes: minInclusive, minExclusive, maxInclusive, maxExclusive
Le rôle de ces facettes est de définir les bornes d'un espace de valeurs, les valeurs minimales et maximales autorisées étant exclues ou incluses en fonction du choix de la facette.

La facette: whiteSpace
Le rôle de cette facette est de spécifier le traitement appliqué aux séparateurs blancs se trouvant dans les données au moment de leur transfert de l'espace lexical à l'espace de valeurs. La facette permet de définir trois modes de traitement des blancs: preserve, replace et collapse.

La facette: pattern
Tous les types sans exception sont concernés par cette facette. La valeur autorisée pour pattern est une expression régulière permettant de définir un motif lexical. Le rôle de cette facette est d'imposer au contenu des éléments ou des attributs une forme lexicale précise. C'est d'ailleurs la seule facette à agir directement sur l'espace lexical.

La facette: enumeration
Tous les types sont concernés par cette facette, à la seule exception du type boolean. Les valeurs autorisées dans l'énumération sont toutes les valeurs autorisées par le type de base.

Les facettes: length, maxLength, minLength
Le rôle de la facette length est de spécifier la longueur de la donnée exprimée en nombre d'octets occupés en mémoire. Celui des facettes maxLength, minLength est de spécifier les longueurs maximales et minimales des données, l'unité de compte étant toujours l'octet. La valeur indiquée pour length est toujours un nombre entier positif (0 est autorisé), et cette facette ne peut être utilisée en même temps que minLength ou maxLength. La valeur de length est obligatoirement égale à celle du type ancêtre déjà porteur de cette facette, si tel est le cas.

La facette: totalDigits
Le rôle de cette facette est de spécifier le nombre maximal de chiffres composant un nombre. Les blancs de tête et de queue, après la décimale, sont insignifiants dans ce décompte.

La facette: fractionDigits
Cette facette propre au type décimal permet de spécifier la précision du nombre décimal. Sa valeur est obligatoirement un nombre supérieur ou égal à zéro et ne peut en aucun cas être supérieure à celle de la facette totalDigits.

Les correspondances de types de DeWeL:
DeWeL est un langage intermédiaire qui fait la correspondance entre les types des schémas XML complexes et les types primitifs du C++ facilitant ainsi, l'évaluation et la manipulation des types XML.
Le catalogue que nous présentons dans cette annexe reprend, regroupés par familles de types, tous les types définis dans le langage DeWeL.

Chaque type est présenté par une fiche d'identité non-exhaustive, qui contient, entre autres:

- L'origine du type, primitif ou dérivé, avec le chemin de dérivation depuis le type originel le cas échéant.

- La définition générale du type.
- La définition précise des espaces lexicaux du type ainsi que son espace de valeur
- Le type équivalent dans le schéma XML.
- La classe C++ correspondante.
- Les méthodes autorisées par DeWeL.

Sont également données pour chaque type quelques explications complémentaires, ainsi que la liste des facettes et l'éventuelle mention d'une relation d'ordre permettant le tri de valeurs. Enfin, nous donnons la liste officielle des types dérivés.

A.1.3.4.1 Les types numériques

Le Type **decimal**

Définition	Générale :	decimal est le type le plus général des nombres décimaux.
	Espace lexical :	Sa représentation lexicale est une chaîne de caractères de longueur finie comprenant éventuellement un point décimal (la virgule décimale n'est pas autorisée). Il est autorisé de mettre un signe + ou – devant le nombre. Les zéros de tête ou de queue ne sont pas interdits.
	Espace de valeurs :	L'espace de valeurs dépend de l'éventuelle limite imposée par la facette qui limite le nombre de chiffres *m* avant la virgule (*m* = *t-f*, où t=totalDigits et f=fractionDigits).
Règle d'ordre	Les nombres décimaux peuvent être comparés deux à deux sans ambiguïté	
Facettes	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd :decimal	
Classe C++	Decimal	
Méthodes permises	A définir	
Opérateurs permis	= + - * / % < <= > >= == !=	
Type dérivé	Integer	

Le Type **integer**

Définition	Générale :	integer est un nombre décimal dont la longueur décimale est définie à 0
	Espace lexical :	La représentation lexicale est celle d'une suite de chiffres précédée d'un signe + ou – optionnel. Par défaut, le nombre est positif.
	Espace de valeurs :	L'espace de valeurs des nombres entiers n'est pas limité, sauf dans le cas où un nombre maximum de chiffre t est imposé par la facette totalDigits. Dans ce cas, les bornes de l'espace de valeurs sont -10^t et 10^t.
Facettes	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd :integer	
Classe C++	Decimal	
Méthodes permises	A définir	
Opérateurs permis	= + - * / % < <= > >= == !=	
Type dérivé	nonPositiveInteger, long, nonNegativeInteger	

Le Type **nonPositiveInteger**

Définition	Générale :	Les entiers de ce type sont les entiers négatifs, 0 compris
	Espace lexical :	Identique à celui de integer en ne conservant que les nombres négatifs précédés du signe – et 0.

	Espace de valeurs :	Même règle que pour integer en ne conservant que les entiers négatifs. Dans le cas où totalDigits est spécifié, l'espace de valeurs est celui des nombres compris entre -10^t et 0.
Facettes	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd : negativeInteger	
Classe C++	NegativeInteger	
Méthodes permises	A définir	
Opérateurs permis	= + - * / % < <= > >= == !=	
Type dérivé	negativeInteger	

Le Type **negativeInteger**

Définition	Générale :	Les entiers de ce type sont strictement négatifs
	Espace lexical :	Identique à celui de integer en ne conservant que les nombres négatifs précédés du signe -. Le 0 sous toutes ses formes est exclu (0, 00, 00000, -0, etc.).
	Espace de valeurs :	Même règle que pour integer en ne conservant que les entiers négatifs. Dans le cas où totalDigits est spécifié, l'espace de valeurs est celui des nombres compris entre -10^t et 0.
Facettes	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd :negativeInterger	
Classe C++	NegativeInteger	
Méthodes permises	A définir	
Opérateurs permis	= + - * / % < <= > >= == !=	
Type dérivé	Pas de type dérivé	

Le Type **long**

Définition	Générale :	Les entiers de ce type sont les entiers qui peuvent être codés de manière binaire sur 8 octets, soit une valeur maximale de 2^{63} en base deux (1 bit est conservé pour le signe).
	Espace lexical :	Identique à celui des integers en ne conservant que les nombres compris entre -2^{63} et 2^{63}
	Espace de valeurs :	Les entiers de ce type ont une valeur comprise entre -9223372036854775808 inclus et 9223372036854775807
Facettes	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd :long	
Classe C++	Long	
Méthodes permises	A définir	
Opérateurs permis	= + - * / % < <= > >= == !=	
Type dérivé	Int	

Le Type **int**

Définition	Générale :	Le type int correspond aux entiers codés sur 4 octets
	Espace lexical :	Identique à celui de integer en ne conservant que les nombres compris entre -2^{31} et 2^{31}.
	Espace de valeurs :	Les valeurs sont comprises entre -2147483648 inclus et 2147483647 inclus.
Facettes	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd:int	
Classe C++	Int	
Méthodes permises	A définir	

Opérateurs permis	= + - * / % < <= > >= == !=
Type dérivé	Short

Le Type **short**

Définition	Générale :	Le type int correspond aux entiers codés sur 2 octets
	Espace lexical :	Identique à celui de integer en ne conservant que les nombres compris entre -2^{15} et 2^{15}
	Espace de valeurs :	Les valeurs sont comprises entre -32768 inclus et 32767 inclus
Facettes	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd :short	
Classe C++	Short	
Méthodes permises	A définir	
Opérateurs permis	= + - * / % < <= > >= == !=	
Type dérivé	Byte	

Le Type **byte**

Définition	Générale :	Le type byte correspond aux entiers codés sur 2 octets
	Espace lexical :	Identique à celui de integer en ne conservant que les nombres compris entre -2^7 et 2^7
	Espace de valeurs :	Les valeurs sont comprises entre -128 inclus et 127 inclus
Facettes	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd :byte	
Classe C++	Byte	
Méthodes permises	A définir	
Opérateurs permis	= + - * / % < <= > >= == !=	
Type dérivé	Pas de typé dérivé	

Le Type **nonNegativeInteger**

Définition	Générale :	Les entiers de ce type sont les entiers positifs, 0 compris.
	Espace lexical :	Identique à celui integer en ne conservant que les nombres positifs précédés du signe + (optionnel) et le 0.
	Espace de valeurs :	Même règle que pour integer en ne conservant que les entiers positifs ou nuls. Dans le cas où totalDigits est spécifié, l'espace de valeurs est celui des nombres compris entre 0 inclus et 10^t exclu.
Facettes	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd: nonNegativeInteger	
Classe C++	NonNegativeInteger	
Méthodes permises	A définir	
Opérateurs permis	= + - * / % < <= > >= == !=	
Type dérivé	unsignedLong, positiveInteger	

Le Type **unsignedLong**

Définition	Générale :	Les entiers de ce type sont les entiers positifs qui peuvent être codés de manière binaire sur 8 octets, soit une valeur maximale de 2^{64} en base deux.
	Espace lexical :	Identique à celui de integer en ne conservant que les nombres compris entre 0 et 2^{64}-1.

	Espace de valeurs :	La valeur d'un entier de ce type doit être supérieure ou égale à 0 et inférieure ou égale à 18446744073709551615.
Facettes	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd: unsignedLong	
Classe C++	UnsignedLong	
Méthodes permises	A définir	
Opérateurs permis	= + - * / % < <= > >= == !=	
Type dérivé	unsignedInt	

Le Type **unsignedInt**

Définition	Générale :	Les entiers de ce type sont les entiers positifs qui peuvent être codés de manière binaire sur 2 octets, soit une valeur minimale de 216 en base deux.
	Espace lexical :	Identique à celui de integer en ne conservant que les nombres compris entre 0 et 2^{32}-1
	Espace de valeurs :	La valeur d'un entier de ce type doit être supérieure ou égale à 0 t inférieure ou égale à 4294967295
Facettes	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd :unsignedInt	
Classe C++	UnsignedInt	
Méthodes permises	A définir	
Opérateurs permis	= + - * / % < <= > >= == !=	
Type dérivé	unsignedShort	

Le Type **unsignedShort**

Définition	Générale :	Les entiers de ce type sont les entiers positifs qui peuvent être codés de manière binaire sur 2 octets, soit une valeur maximale de 216 en base deux.
	Espace lexical :	Identique à celui de integer en ne conservant que les nombres compris entre 0 et 2^{16}-1.
	Espace de valeurs :	La valeur d'un entier de ce type doit être supérieure ou égale à 0 et inférieure ou égale à 65535
Facettes	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd :unsignedShort	
Classe C++	UnsignedShort	
Méthodes permises	A définir	
Opérateurs permis	= + - * / % < <= > >= == !=	
Type dérivé	unsignedByte	

Le Type **unsignedByte**

Définition	Générale :	Les entiers de ce type sont les entiers positifs qui peuvent être codés de manière binaire sur 1 octet, soit une valeur maximale de 2^8 n base deux.
	Espace lexical :	Identique à celui de integer en ne conservant que les nombres compris entre 0 et 2^8-1.
	Espace de valeurs :	La valeur d'un entier de ce type doit être supérieure ou égale à 0 et inférieure ou égale à 255.
Facettes	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd :unsignedByte	
Classe C++	UnsignedByte	
Méthodes permises	A définir	

Opérateurs permis	= + - * / % < <= > >= == !=
Type dérivé	Pas de type dérivé

Le Type **positiveInteger**

Définition	Générale :	Les entiers de ce type sont les entiers strictement positifs.
	Espace lexical :	Identique à celui de integer en ne conservant que les nombres positifs précédés du signe + (optionnel).
	Espace de valeurs :	Même règle que pour integer en ne conservant que les entiers positifs. Dans le cas où totalDigits est spécifié et si t représente cette valeur, l'espace de valeurs est celui des nombres compris entre 0 exclu et 10^t exclu.
Facettes	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd :positivInteger	
Classe C++	PositiveInteger	
Méthodes permises	A définir	
Opérateurs permis	= + - * / % < <= > >= == !=	
Type dérivé	Pas de type dérivé	

Le Type **float**

Définition	Générale :	Le type float correspond aux nombres flottants simple précision sur 32 bits
	Espace lexical :	Un nombre flottant est représenté par une mantisse suivie de la lettre E (e est autorisé aussi) suivie d'un chiffre représentant un exposant. La mantisse doit être un nombre décimal de type decimal ; l'exposant doit être un nombre entier de type integer. Le E et l'exposant sont optionnels.
	Espace de valeurs :	L'espace de valeurs est fait de nombres multiples de puissances de 2 qui s'écrivent sous la forme *m* * 2e où *m* est un nombre entier dont la valeur absolue doit être inférieure à 2^{24}, et où -149<=e <= 104. Si le E ou le e sont omis, l'exposant pris en compte par défaut est 0.
Facettes	pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd :float	
Classe C++	Float	
Méthodes permises	A définir	
Opérateurs permis	= + - * / % < <= > >= == !=	
Type dérivé	Pas de type dérivé	

Le Type **double**

Définition	Générale :	Le type double correspond aux nombres flottants double précision sur 32 bits
	Espace lexical :	Un nombre flottant est représenté par une mantisse suivie de la lettre E (e est autorisé aussi) suivie d'un chiffre représentant un exposant.
	Espace de valeurs :	L'espace de valeurs est fait de nombres multiples de puissances de 2 qui s'écrivent sous la forme *m* * 2e où *m* est un nombre entier dont la valeur absolue doit être inférieure à 2^{24}, et où -149<=e <= 104. Si le E ou le e sont omis, l'exposant pris en compte par défaut est 0.
Facettes	pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive	

Schéma XML	xsd:double
Classe C++	Double
Méthodes permises	A définir
Opérateurs permis	= + - * / % < <= > >= == !=
Type dérivé	Pas de type dérivé

A.1.3.4.2 Les types chaînes de caractères

Le Type **string**

Définition	Générale :	Le type string est celui des chaînes de caractères.
	Espace lexical :	L'espace lexical est celui défini par la règle de production Char de XML 1.0.
	Espace de valeurs :	-
Facettes	lenght, minLenght, maxLenght, pattern, enumeration, whiteSpace.	
Schéma XML	xsd:string	
Classe C++	String	
Méthodes permises	Contains	
Opérateurs permis	+ =	
Type dérivé	normalizedString	

Le Type **normalizedString**

Définition	Générale :	Ce type représente les chaînes de caractères dont les séparateurs blancs sont normalisés dans l'espace de valeurs.
	Espace lexical :	Les caractères retour chariot (#xD) et tabulation (#x9) sont interdits.
	Espace de valeurs :	Dans l'espace de valeur, la chaîne de caractères ne contient plus les caractères retour chariot (#xD), nouvelle ligne (#xA) et tabulation (#x9), qui sont remplacés par des blancs (#x20).
Facettes	lenght, minLenght, pattern, enumeration, whiteSpace.	
Schéma XML	xsd:normalizedString	
Classe C++	NormalizedString	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Token	

Le Type **token**

Définition	Générale :	Ce type représente une chaîne de caractères constituée d'unités lexicales.
	Espace lexical :	L'espace lexical est celui des chaînes de caractères qui ne contiennent ni caractère nouvelle ligne (#xA), ni tabulation (#x9), ni blanc de tête ou de queue (#x20), ni séquence interne de deux blancs ou plus (#x20).
	Espace de valeurs :	Les contraintes qui portent sur l'espace de valeurs sont identiques à celles de l'espace lexical.
Facettes	lenght, minLenght, maxLenght, pattern, enumeration, whiteSpace	
Schéma XML	xsd:token	
Classe C++	Token	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	language, name, NMTOKEN	

Le Type **language**

Définition	Générale :	Le type language est destiné aux identifiants de langues naturelles tels que définis par l'IETF (RFC 1766).
	Espace lexical :	Les formes lexicales autorisées sont celles des identifiants de langues définis.
	Espace de valeurs :	L'espace de valeurs est égal à l'espace lexical.
Facettes	lenght, minLenght, maxLenght, pattern, enumeration, whiteSpace	
Schéma XML	xsd:language	
Classe C++	Language	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Il n'y a pas de type dérivé.	

Le Type **Name**

Définition	Générale :	Le type Name correspond au type Nom de XML 1.0.
	Espace lexical :	La forme d'un nom est définie par la règle de production : Name::= (Letter | '_' | ':' |) (NameChar)* NameChar::= Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar | Extender Un type Name commence obligatoirement par une lettre, un tiret ou deux-points.
	Espace de valeurs :	Identique à l'espace lexical
Facettes	lenght, minLenght, maxLenght, pattern, enumeration, whiteSpace	
Schéma XML	xsd:name	
Classe C++	Name	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	NCName	

Le Type **NCName**

Définition	Générale :	Le type NCName est celui des non-colonized Name, dans laquelle le mot colonized fait référence au caractère deux-points qui se dit *colon* en anglais.
	Espace lexical :	La forme d'un NCName est définie par la règle de production : NCName::=(Letter | '_') (NCNameChar)* NCNameChar::= Letter | Digit | '.' | '-' | '_' | CombinigChar | Extender Il s'agit de la même définition que pour un Name dans laquelle les deux-points ont été reliés.
	Espace de valeurs :	-
Facettes	lenght, minLenght, maxLenght, pattern, enumeration, whiteSpace	
Schéma XML	xsd:NCName	
Classe C++	NCName	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

Le Type **QName**

Définition	Générale :	QName est le type des noms qualifiés, définis par le tuple {nom d'espace de noms, partie locale}
	Espace lexical :	La forme lexicale d'un QName est définie par la règle de production : QName::= (Prefix ' :') ? LocalPart

		Prefix::= NCName LocalPart::= NCName
	Espace de valeurs :	Dans l'espace de valeurs, le préfixe est remplacé par l'URI de l'espace de noms associé au préfixe.
Facettes	lenght, minLenght, maxLenght, pattern, enumeration, whiteSpace	
Schéma XML	xsd:Qname	
Classe C++	Qname	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

A.1.3.4.3 Le type booléen

Le Type **boolean**

Définition	Générale :	Le type boolean représente les valeurs booléennes.
	Espace lexical :	Les formes littérales autorisées sont {true, false,0, 1}.
	Espace de valeurs :	L'espace des valeurs est limité aux deux seules valeurs true et false.
Facettes	pattern, whiteSpace	
Schéma XML	xsd:boolean	
Classe C++	Boolean	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Il n'y a pas de type dérivé.	

A.1.3.4.4 Les dates grégoriennes

Le Type **gYear**

Définition	Générale :	Ce type représente une année grégorienne.
	Espace lexical :	La représentation lexicale est indépendante de la durée réelle de l'année. La forme lexicale est CCYY. Les quatre chiffres sont obligatoires et la valeur peut être suivie d'un indicateur de décalage horaire. Pour les années avant 0001, le signe – est autorisé. Pour les années dont le nombre représentatif est supérieur à 9999, des chiffres peuvent être ajoutés. Par exemple, 250 000 ans avant J.-C. s'écrit -250000.
	Espace de valeurs :	-
Facettes	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd:gYear	
Classe C++	GYear	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

Le Type **gMonthDay**

Définition	Générale :	Ce type représente un jour dans le mois, indépendamment de l'année. Par exemple, le 23 mai.
	Espace lexical :	La forme lexicale est : --MM-DD, complétée éventuellement d'un indicateur de décalage horaire. Aucun signe + ou – n'est autorisé avant.
	Espace de valeurs :	-

Facettes	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
Schéma XML	xsd:gMonthDay
Classe C++	GMonthDay
Méthodes permises	A définir
Opérateurs permis	=
Type dérivé	Pas de type dérivé

Le Type **gDay**

Définition	Générale :	ce type représente un jour sans précision de mois, le 5 par exemple. Il s'agit d'une notion de répétition mensuelle d'un jour précis.
	Espace lexical :	La forme lexicale de ce type est --- DD, complétée optionnellement d'une indication de décalage horaire.
	Espace de valeurs :	-
Facettes	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd:gDay	
Classe C++	GDay	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

Le Type **gMonth**

Définition	Générale :	Ce type représente un mois sans indication d'année. Il convient donc pour donner des répétitions annuelles d'un mois précis.
	Espace lexical :	La représentation lexicale est –MM--, suivie d'une indication optionnelle de décalage horaire.
	Espace de valeurs :	-
Facettes	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd:gMonth	
Classe C++	GMonth	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

Le Type **gYearMonth**

Définition	Générale :	Ce type représente un mois d'une année précise.
	Espace lexical :	La représentation lexicale est CCYY-MM, éventuellement suivie d'une indication de décalage horaire. Les quatre chiffres de l'année sont obligatoires. La forme peut être précédée d'un signe - et le nombre représentant l'année peut avoir plus de 4 chiffres.
	Espace de valeurs :	-
Facettes	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd:gYearMonth	
Classe C++	GYearMonth	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

A.1.3.4.5 Les types pour la mesure du temps

Le Type **duration**

Définition	Générale :	Le type duration permet de spécifier des durées.
	Espace lexical :	La représentation lexicale est celle définie dans l'ISO 8601 sous la forme PnYn MnDTnHnMnS, où nY est le nombre d'années, nM le nombre de mois, nD le nombre de jours, T le séparateur usuel entre les jours et les heures, nH le nombre d'heures, nM le nombre de minutes et nS le nombre de secondes et optionnellement de fractions de secondes. Cette forme lexicale peut-être précédée du signe-. Exemple : une durée d'un an, deux mois, trois jours, dix heures et trente minutes s'écrit P1Y2M3DT10H30M. une simple durée portant sur des heures, 15 heures par exemple, s'écrit P15H.
	Espace de valeurs :	L'espace de valeurs est formé de tuples contenant l'année grégorienne, le mois, le jour, l'heure, la minute et la seconde.
Facettes	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd:duration	
Classe C++	Duration	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

Le Type **dateTime**

Définition	Générale :	Ce type représente une heure précise par rapport à une date.
	Espace lexical :	La seule représentation lexicale autorisée est un sous-cas de l'ISO 8601, il s'agit de la forme CCYY-MM-DDThh:mm:ss dans laquelle : • CC est le siècle. • YY l'année. • MM le mois. • DD le jour. • T est le séparateur entre la date et l'heure. • Hh,mm,ss représentent les heures, les minutes et les secondes. Il est possible d'ajouter une valeur décimale représentant des fractions de seconde. Cela constitue la seule partie optionnelle de l'espace lexical de ce type. Une telle date peut être précédée du signe- pour indiquer une valeur négative. Les quatre chiffres des années sont obligatoires, et l'année 875 s'écrit 0875. il est possible de mettre plus de quatre chiffres pour les années supérieures à 9999. l'année 0000 est interdite. Tous les autres champs doivent obligatoirement être composés de deux chiffres. Le principe des zéros de tête est retenu pour les valeurs inférieures à 10. Par exemple, 9 s'écrit 09. Cette représentation peut être immédiatement suivie d'un

		Z pour indiquer qu'il s'agit du temps universel ou d'un signe + ou – pour indiquer qu'il s'agit d'un décalage horaire en décalage avec le TU (cas des pays de l'Est).
	Espace de valeurs :	-
Facettes	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd:dateTime	
Classe C++	DateTime	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

Le Type **time**

Définition	Générale :	Le type time représente une heure de la journée indépendante du jour.
	Espace lexical :	L'espace lexical est de la forme hh:mm:ss.sss, à laquelle on peut ajouter un indicateur de décalage horaire.
	Espace de valeurs :	
Facettes	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd:time	
Classe C++	Time	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

Le Type **date**

Définition	Générale :	Date représente une date du calendrier grégorien telle que définie par l'ISO 8601. Un jour est une entité pleine et entière, quel que soit le nombre réel d'heures de la journée.
	Espace lexical :	La représentation lexicale est CCYY-MM-DD, complétée d'une indication optionnelle de décalage horaire. La forme peut être précédée du signe – pour les années avant l'année 1 (la première année du calendrier grégorien).
	Espace de valeurs :	-
Facettes	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive	
Schéma XML	xsd:date	
Classe C++	Date	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

A.1.3.4.6 Les types binaires

Le Type **base64Binary**

Définition	Générale :	Ce type est celui des données binaires codées en base 64.
	Espace lexical :	-
	Espace de valeurs :	L'espace de valeurs est constitué de séquence d'octets binaires. L'algorithme de codage est défini par l'IETF (RFC 2045).
Facettes	lenght, minLenght, maxLenght, pattern, enumeration, whiteSpace	
Schéma XML	xsd:base64Binary	

Classe C++	Base64Binary
Méthodes permises	A définir
Opérateurs permis	=
Type dérivé	Pas de type dérivé

Le Type **hexBinary**

Définition	Générale :	Ce type représente les données binaires représentées en clair sous la forme de leur code hexadécimal.
	Espace lexical :	La représentation lexicale est une conversion de chaque octet binaire en un tuple représentant le code hexadécimal de l'octet. Sont autorisés les chiffres de 0 à 9 et les lettres [a-f] et [A-F]. Par exemple, 0FB7 est la représentation en clair de l'entier 4023, qui tient sur deux octets quand il est codé en binaire sous la forme 0000111110110111. «OF »= 00001111 et « B73=10110111.
	Espace de valeurs :	L'espace de valeurs est une forme binaire, par exemple 0000111110110111.
Facettes	lenght, minLenght, maxLenght, pattern, enumeration, whiteSpace	
Schéma XML	xsd:hexBinary	
Classe C++	HexBinary	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

A.1.3.4.7 Les types URI

Le Type **anyURI**

Définition	Générale :	Le type anyURI représente un URI.
	Espace lexical :	L'espace lexical prend la forme d'une chaîne de caractères finie, qui doit être conforme aux spécifications IETF (RFC 2396 et 2732). L'URI peut être un simple nom ou une adresse complète de ressource. Un URI peut être absolu ou relatif et être complété d'un identifiant de fragment.
	Espace de valeurs :	-
Facettes	lenght, minLenght, maxLenght, pattern, enumeration, whiteSpace	
Schéma XML	xsd:anyURY	
Classe C++	AnyURI	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

A.1.3.4.8 Les types hérités de XML 1.0

Le Type **NOTATION**

Définition	Générale :	Ce type est une spécification d'unité lexicale de type QNames, dont la valeur doit être définie par une déclaration de NOTATION.
	Espace lexical :	Identique à celui de QName
	Espace de valeurs :	Identique à celui de QName
Facettes	lenght, minLenght, maxLenght, pattern, enumeration, whiteSpace	
Schéma XML	xsd:NOTATION	
Classe C++	NOTATION	

Méthodes permises	A définir
Opérateurs permis	=
Type dérivé	Pas de type dérivé

Le Type **NMTOKEN**

Définition	Générale :	Le type NMTOKEN est le même que celui de XML 1.0.
	Espace lexical :	La forme lexicale est celle d'une unité lexicale formée de caractères de noms – Nmtoken ::= (Letter | Digit | '-' | '.')+ -- Un NMTOKEN est une suite contiguë de caractères autorisés. Aucun séparateur blanc n'est permis (ni blanc, ni tabulation, ni retour chariot).
	Espace de valeurs :	Identique à l'espace lexical
Facettes	lenght, minLenght, maxLenght, pattern, enumeration, whiteSpace	
Schéma XML	xsd:NMTOKEN	
Classe C++	NMTOKEN	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	NMTOKENS	

Le Type **NMTOKENS**

Définition	Générale :	Le type NMTOKENS est le même que celui de XML 1.0.
	Espace lexical :	La forme lexicale est celle d'une série de NMTOKEN séparés les uns des autres par un blanc.
	Espace de valeurs :	Identique à la forme lexicale
Facettes	lenght, minLenght, maxLenght, pattern, enumeration, whiteSpace	
Schéma XML	xsd:NMTOKENS	
Classe C++	NMTOKENS	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

Le Type **ID**

Définition	Générale :	Le type ID représente des valeurs de type NCName dont la particularité est d'être uniques dans tout l'ensemble d'information.
	Espace lexical :	Identique à celui de NCName
	Espace de valeurs :	Identique à celui de NCName
Facettes	lenght, minLenght, maxLenght, pattern, enumeration, whiteSpace	
Schéma XML	xsd:ID	
Classe C++	ID	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

Le Type **IDREF**

Définition	Générale :	Le type IDREF représente des valeurs de type NCName dont la particularité est d'être égales à des valeurs de type ID.
	Espace lexical :	Identique à celui de NCName
	Espace de valeurs :	Identique à celui de NCName
Facettes	lenght, minLenght, maxLenght, pattern, enumeration, whiteSpace	
Schéma XML	xsd:IDREF	
Classe C++	IDREF	
Méthodes permises	A définir	

Opérateurs permis	=
Type dérivé	Pas de type dérivé

Le Type **IDREFS**

Définition	Générale :	Le type IDREFS représente des valeurs de type NCName dont la particularité est d'être égales à des valeurs de type ID.
	Espace lexical :	Identique à celui de NCName
	Espace de valeurs :	Identique à celui de NCName
Facettes	lenght, minLenght, maxLenght, pattern, enumeration, whiteSpace	
Schéma XML	xsd:IDREFS	
Classe C++	IDREFS	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

Le Type **ENTITY**

Définition	Générale :	Le type ENTITY est celui qui spécifie qu'une chaîne de caractères de type NCName doit être égale à une entité connue de l'ensemble d'information.
	Espace lexical :	Identique à celui de NCName
	Espace de valeurs :	Identique à celui de NCName
Facettes	lenght, minLenght, maxLenght, pattern, enumeration, whiteSpace	
Schéma XML	xsd:ENTITY	
Classe C++	ENTITY	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

Le Type **ENTITIES**

Définition	Générale :	Le type ENTITIES est la spécification d'une liste de noms d'entités de type ENTITY séparés les uns des autres par un blanc.
	Espace lexical :	Identique à celui de NCName
	Espace de valeurs :	Identique à celui de NCName
Facettes	Length, minLenght, maxLenght, pattern, enumeration, whiteSpace	
Schéma XML	xsd:ENTITIES	
Classe C++	ENTITIES	
Méthodes permises	A définir	
Opérateurs permis	=	
Type dérivé	Pas de type dérivé	

A.2 Comparaison entre DeWeL et le langage C

Nom du Service Web	Canevas DeWeL	Types spécifiques	Connecteur	Code généré par DeWeL	Ratio
DOTSAddressValidate	106	2258	1921	4179	39,42
DotnetDailyFact	22	174	353	527	23,95
DDBJ	94	0	1718	1718	18,28
GetCustomNews	22	180	353	533	24,23
SportingGoodsFinder	22	180	353	533	24,23
GetEntry	658	0	12258	12258	18,63
SQLDataSoap	46	18	833	851	18,50
AWSECommerceService	238	16979	4385	21364	89,76
AmazonBox	406	5364	7521	12885	31,74
ZipCode	94	1241	1697	2938	31,26
BarCodesService	22	99	365	464	21,09
ZipCodes	118	1574	2134	3708	31,42
ISOcode2shortformatSOAP	22	0	354	354	16,09
CATrafficService	22	0	354	354	16,09
DNSLookupService	34	512	577	1089	32,03
AmazonQuery	34	348	593	941	27,68
UPSTracking	34	1130	577	1707	50,21
CommonService	70	12935	1249	14184	202,63
MSNSearchService	22	1676	353	2029	92,23
PersonLookup	22	398	353	751	34,14
InstantMessageAlert	58	738	1025	1763	30,40
WebSearchWS	22	180	353	533	24,23
AddressLookup	166	2732	3041	5773	34,78
RandomGoogleSearchService	22	105	353	458	20,82
XmlDailyFact	22	174	353	527	23,95
FindPeopleFree	46	617	801	1418	30,83
avernetService	22	0	320	320	14,55
DOTSFastWeather	106	2364	1921	4285	40,42
HashClass	34	360	577	937	27,56
CurrencyExchangeService	22	0	358	358	16,27
DOTSGeoCoder	82	1592	1473	3065	37,38
SRS	58	0	1034	1034	17,83
YourHost	46	823	801	1624	35,30
Calendar	22	0	354	354	16,09
WhoIS	58	678	1025	1703	29,36
XigniteSimulation	94	2825	1697	4522	48,11
CodeGenerator	22	180	353	533	24,23
DailyDilbert	34	330	577	907	26,68
DOTSFastTax	22	444	353	797	36,23
FreeDBService	70	1371	1249	2620	37,43
YahooUserPingService	22	0	354	354	16,09
Guitars	46	698	801	1499	32,59
eSynapsSerach	22	186	353	539	24,50

ICQ	94	1164	1697	2861	30,44
GlobalAddressVerification	70	2425	1249	3674	52,49
Blast	70	0	1290	1290	18,43
Zip2Geo	22	345	353	698	31,73
DOTSGeoPinPoint	22	414	353	767	34,86
WeatherFetcher	34	519	577	1096	32,24
IgetNumbersservice	22	0	366	366	16,64
Phone	262	3585	4833	8418	32,13
NameLookup	142	1919	2593	4512	31,77
UKAddressVerification	46	1521	801	2322	50,48
TxSearch	70	0	1274	1274	18,20
Puki	22	180	353	533	24,23
BNPrice	22	180	353	533	24,23
Dispenser	118	1875	2145	4020	34,07
FaxService	22	0	374	374	17,00
USHolidayDates	298	3906	5505	9411	31,58
Phonebook	22	410	353	763	34,68
BZip2	34	342	577	919	27,03
DOTSPhoneAppend	46	1034	801	1835	39,89
WS4IsqlService	10	1225	101	1326	132,60
XigniteSecurity	202	5028	3713	8741	43,27
XMethodsFilesystemService	94	0	1746	1746	18,57
TAP	58	672	1025	1697	29,26
NumPager	58	672	1025	1697	29,26
XigniteOptions	58	1906	1025	2931	50,53
WhoIsService	22	180	353	533	24,23
unitext	22	186	353	539	24,50
Bible	22	0	354	354	16,09
XigniteRealTime	286	7972	5281	13253	46,34
CompanyInfoService	22	231	361	592	26,91
eBayWatcherService	22	0	354	354	16,09
IRomanservice	34	0	578	578	17,00
convert	22	0	362	362	16,45
HyperlinkExtractor	22	293	353	646	29,36
TWSFissionDotNet	34	573	581	1154	33,94
bork	22	18	353	371	16,86
Horoscope	22	374	353	727	33,05
PopulationWS	46	694	801	1495	32,50
DOTSFastQuote	70	1370	1249	2619	37,41
LocalTime	22	180	353	533	24,23
XigniteEdgar	226	7190	4161	11351	50,23
FedRoutingDirectoryService	82	1832	1473	3305	40,30
WeblogsSubscriber	22	108	323	431	19,59
BankValSOAP	22	0	354	354	16,09
dConverterSOAP	22	18	361	379	17,23
ScreenScraper	22	180	353	533	24,23
DOTSGeoCash	58	1049	1025	2074	35,76
DOTSYellowPages	46	1087	801	1888	41,04
XigniteSurvey	106	2549	1921	4470	42,17
XreOnline	46	528	801	1329	28,89

CupScores	34	614	577	1191	35,03
ErrorMailer	58	702	1025	1727	29,78
ISOcodesSOAP	22	218	353	571	25,95
SMS	70	927	1249	2176	31,09
DOTSEmailValidate	58	954	1025	1979	34,12
IP2Geo	34	477	577	1054	31,00
DOTSPackageTracking	166	2502	3041	5543	33,39
XigniteNews	130	2989	2369	5358	41,22
eSynapsMonitor	34	552	577	1129	33,21
MSProxy	22	174	353	527	23,95
ITempConverterservice	34	0	578	578	17,00
MailLocate	34	285	613	898	26,41
SmsService	46	0	821	821	17,85
CountCheatService	34	461	577	1038	30,53
GeoPlaces	118	1436	2161	3597	30,48
AddressFinder	58	903	1040	1943	33,50
BusinessNews	58	1102	1025	2127	36,67
finnwordsService	22	0	354	354	16,09
imstatus	22	18	353	371	16,86
ServiceSMS	46	546	801	1347	29,28
GMChart	34	3312	577	3889	114,38
BNQuoteService	22	0	354	354	16,09
dic2	58	860	1025	1885	32,50
XigniteInsider	202	9061	3713	12774	63,24
FlashBarChartService	22	99	397	496	22,55
HistoricalStockQuotes	46	853	801	1654	35,96
DOTSUPC	22	390	353	743	33,77
DnB	118	2583	2145	4728	40,07
dns	22	0	354	354	16,09
engtoarabic	22	180	353	533	24,23
QuoteofTheDay	22	261	353	614	27,91
QueryIP	22	186	353	539	24,50
SMSTextMessaging	94	2461	1697	4158	44,23
XigniteStatistics	190	4932	3489	8421	44,32
RandomBushismService	22	105	348	453	20,59
ElectronicProductsFinder	22	180	353	533	24,23
zipinfo	34	1108	577	1685	49,56
XEMBLService	22	0	358	358	16,27
VideoGamesFinder	22	180	353	533	24,23
Shakespeare	22	180	353	533	24,23
BabelFishService	22	0	358	358	16,27
Server	34	108	581	689	20,26
IDutchservice	22	0	354	354	16,09
IEuroservice	34	0	586	586	17,24
pop	34	0	602	602	17,71
Fax	82	1008	1473	2481	30,26
SendEmailService	22	222	353	575	26,14
WorldTimeService	58	0	1025	1025	17,67
wwhelpservice	34	18	597	615	18,09
SendMessages	106	1692	1921	3613	34,08

XigniteQuotes	274	8018	5057	13075	47,72
FrenchAddressVerification	46	1509	801	2310	50,22
CSearch	46	1048	801	1849	40,20
WolframSearchService	46	1104	801	1905	41,41
foxcentral	226	18	4250	4268	18,88
Currencyws	34	360	577	937	27,56
Fasta	58	0	1066	1066	18,38
huzipService	22	93	361	454	20,64
XigniteRetirement	70	1938	1249	3187	45,53
SiteInspectService	22	111	361	472	21,45
IndianAddressVerification	46	1521	801	2322	50,48
PersistenceServiceService	286	1393	5060	6453	22,56
CEqImage	34	354	577	931	27,38
ApniUrdu	58	0	1026	1026	17,69
airport	58	666	1025	1691	29,16
ev	34	506	577	1083	31,85
TemperatureService	22	0	354	354	16,09
LOTTO_UK_NumberChecker	22	329	353	682	31,00
chat	22	18	353	371	16,86
check	34	604	577	1181	34,74
GoogleSearchService	46	555	845	1400	30,43
NFLNews	22	174	353	527	23,95
TerraService	202	4359	3713	8072	39,96
XMethodsQuery	70	543	1239	1782	25,46
WebChart	22	18	405	423	19,23
sekeywordService	22	93	353	446	20,27
XSpaceService	226	374	3853	4227	18,70
eSynapsFeed	22	174	353	527	23,95
HPcatalogService	22	108	353	461	20,95
DOTSGeoPhone	46	1227	801	2028	44,09
zipdistance	34	884	577	1461	42,97
Moyenne:	64,76	1093,88	1151,83	2245,71	31,56

A.3 Algorithme de génération d'interface abstraite

L'algorithme, ci-dessous, réalisé en Prolog permet de résoudre et fournir en sortie l'interface d'entrée minimale et l'interface de sortie maximale pour une opération abstraite.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% définition de l'architecture de l'opération abstraite %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% équivalences sémantiques en entrée.
hee([[[c],[f,g,a]],[[d],[g,h,a]],[[h],[e,b]]]).

% équivalences sémantiques en sortie.
hes([[[k],[n,o,i]],[[o],[p,m]],[[j],[n,m]]]).
% interfaces concrètes d'entrée
ope([f,g,h]).
ope([c,d,e]).
ope([a,b]).

% interfaces concrètes de sortie
ops([n,o]).
ops([k,p,m]).
ops([i,j]).


%%%%%%%%%%%%%%%%%%%%%%%%%
% fonctions utilitaires %
%%%%%%%%%%%%%%%%%%%%%%%%%

% vérifie si l'intersection de L1 et L2 est bien nulle
zero_inter(L1,L2):- forall(member(X,L1),\+member(X,L2)).

% cl est vrai si à partir de l'ensemble des noeuds de L et des hyper arcs
% dans HE, on peut atteindre tout point de X. Le dernier paramètre contient
% la liste des hyper-arcs qu'il faut utiliser.
cl(HE,L,X,[[I,O]|LE]):-member([I,O],HE), zero_inter(L,O), subset(I,L),
                       append(L,O,L2), cl(HE,L2,X,LE).
cl(_,L,L,[]).

% gen_subset est vraie si le second paramètre est un sous ensemble du
premier.
% Cette fonction peut être utilisée en mode (lié, libre) et donc est
utilisée
% pour générer l'ensemble des parties d'un ensemble connu
% représenté sous forme de liste.
gen_subset([X|L1], [X|L2]):-gen_subset(L1,L2).
gen_subset([X|L1], L2):-gen_subset(L1,L2).
gen_subset([],[]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Création de l'interface d'entrée abstraite minimale %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Cette fonction produit toutes les interfaces abstraites d'entrée
% qui sont valides (cf. notion de cohérence)
findall_IE(Lres,EnsHEactif):
              findall(IE,ope(IE),Largspart),flatten(Largspart,Largs),
              findall(X,gen_subset(Largs,X),L),
```

```
                findall([NY,Y,LY,LYE],
                        (member(Y,L),cl(EnsHEactif,Y,LY,LYE),length(Y,NY)),
                        LR),
                findall([NZ,Z,LZE],
                        (member([NZ,Z,LZ,LZE],LR),permutation(Largs,LZ)),
                        Lres).

% permet de sélectionner les interfaces abstraites de taille minimale
select_min([[NY,Y,LYE]|Lres],Lmin,Nmin):-
              Lres\=[], select_min(Lres,Lmintemp,Nmintemp),
              NY is min(NY,Nmintemp), NY\=Nmintemp, Nmin=NY, Lmin=[[Y,LYE]].
select_min([[NY,Y,LYE]|Lres],Lmin,Nmin):-
              Lres\=[], select_min(Lres,Lmin,Nmintemp),
              Nmintemp is min(NY,Nmintemp), Nmintemp\=NY, Nmin=Nmintemp.
select_min([[NY,Y,LYE]|Lres],Lmin,Nmin):-
              Lres\=[], select_min(Lres,Lmintemp,Nmintemp),
              NY = Nmintemp, Nmin=NY, Lmin=[[Y,LYE]|Lmintemp].
select_min([[NY,Y,LYE]],[[Y,LYE]],NY).



% commande finale utilisée pour demander la liste des interfaces d'entrée
% abstraite minimales.
% attention: on fournit aussi la liste des équivalences sémantiques
% utilisées avec chaque interface abstraite
get_intE_abs(Lint,EnsHEactif):- findall_IE(Lres,EnsHEactif),
                                select_min(Lres,Linttemp,_),
                                list_to_set(Linttemp,Lint).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Création de l'interface de sortie abstraite maximale %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% Remarque : le prédicat ops(L) est tel que ops(L) est vrai si et seulement
si L est une interface de sortie pour une opération concrète.

% on obtient l'ensemble des interfaces de sortie valides
findall_IS(Lres,EnsHEactif) :- findall(L,ops(L),Lpart),

findall([Y,LY],(member(Y,Lpart),cl(EnsHEactif,Y,LY,_)),Lcandidates),
            findall([NX,X],(flatten(Lpart,LargsS),gen_subset(LargsS,X),
            X\=[],length(X,NX),
            forall(member([Z,_],Lcandidates),
                   (member([Z,LZ],Lcandidates),subset(X,LZ)))),Lres).


% permet de sélectionner les interfaces de sortie de plus grande taille
select_max([[NY,Y]|Lres],Lmax,Nmax):-
            Lres\=[], select_max(Lres,Lmaxtemp,Nmaxtemp),
            NY is max(NY,Nmaxtemp), NY\=Nmaxtemp, Nmax=NY, Lmax=[Y].
select_max([[NY,Y]|Lres],Lmax,Nmax):-
            Lres\=[], select_max(Lres,Lmax,Nmaxtemp),
            Nmaxtemp is max(NY,Nmaxtemp), Nmaxtemp\=NY, Nmax=Nmaxtemp.
select_max([[NY,Y]|Lres],Lmax,Nmax):-
            Lres\=[], select_max(Lres,Lmaxtemp,Nmaxtemp),
            NY = Nmaxtemp, Nmax=NY, Lmax=[Y|Lmaxtemp].
select_max([[NY,Y]],[Y],NY).
```

```
% retourne la liste des interfaces abstraites de sortie maximales.
get_intS_abs(Lres,EnsHEactif):-
            findall_IS(Lrestemp,EnsHEactif),select_max(Lrestemp,Lres,_).
```

A.4 Service Web Abstrait du moteur de recherches : Google et MSN

Cette section présente les différents fichiers nécessaires pour créer un connecteur à partir d'un Service Web Abstrait.

A.4.1 Le contrat du Service Web Abstrait

L'élément essentiel est le contrat du service Web abstrait. L'exemple présenté ci-dessous correspond à un Service Web abstrait de moteur de recherches. Celui-ci agrège deux services concrets : Le service de Google et le service de MSN. Ce service ne possède qu'une opération « doAbstractSearch ». Celle-ci est constituée en entrée de variables productrices et locales et en sortie de variables induites.

```
<!-- WSDL description of the Abstract Web APIs.
    The Abstract Web APIs are in beta release. All interfaces are subject to
    change as we refine and extend our APIs. Please see the terms of use
    for more information. -->
<definitions name="AbstractSearch"
          targetNamespace="http://www.iwsd.com"
          xmlns:typens="http://www.iwsd.com"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
          xmlns:wsd="http://salatge-wifi:8080/ManagementServices/DependabilityProperties.xsd"
          xmlns="http://schemas.xmlsoap.org/wsdl/">
   <!-- Types for search - result elements, directory categories -->
   <types>
      <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.iwsd.com">
         <xsd:complexType name="Resultat">
            <xsd:all>
               <xsd:element name="resume" type="xsd:string"/>
               <xsd:element name="url" type="xsd:string"/>
               <xsd:element name="titre" type="xsd:string"/>
            </xsd:all>
         </xsd:complexType>
         <xsd:complexType name="ListElementArray">
            <xsd:complexContent>
               <xsd:restriction base="soapenc:Array">
                  <xsd:attribute ref="soapenc:arrayType" wsdl:arrayType="typens:Resultat[]"/>
               </xsd:restriction>
            </xsd:complexContent>
         </xsd:complexType>
      </xsd:schema>
   </types>
   <!-- note, ie and oe are ignored by server; all traffic is UTF-8. -->
   <message name="abstractSearch">
      <part name="cleGoogle"       type="xsd:string"/>
      <part name="cleMSN"          type="xsd:string"/>
      <part name="recherche"       type="xsd:string"/>
   </message>
   <message name="abstractSearchResponse">
      <part name="listElements"   type="typens:ListElementArray"/>
   </message>
   <!-- Port for Abstract Web APIs, "AbstractSearch" -->
   <portType name="AbstractSearchPort">
      <operation name="doAbstractSearch">
         <input message="typens:abstractSearch"/>
         <output message="typens:abstractSearchResponse"/>
      </operation>
   </portType>
   <!-- Binding for Abstract Web APIs - RPC, SOAP over HTTP -->
   <binding name="AbstractSearchBinding" type="typens:AbstractSearchPort">
      <soap:binding style="rpc"
            transport="http://schemas.xmlsoap.org/soap/http"/>
```

```
      <operation name="doAbstractSearch">
        <input>
          <soap:body use="encoded"
            namespace="http://www.iwsd.com"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </input>
        <output>
          <soap:body use="encoded"
            namespace="http://www.iwsd.com"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </output>
      </operation>
    </binding>
    <!-- Endpoint for Abstract Web APIs -->
    <service name="AbstractSearchService">
      <port name="AbstractSearchPort" binding="typens:AbstractSearchBinding">
        <soap:address location="http://www.iwsd.com/ABSTRACT"/>
      </port>
      <wsd:ReplicatedAccessPoint location="http://api.google.com/search/beta2" name="GOOGLE">
        <wsd:wsdl>http://api.google.com/GoogleSearch.wsdl</wsd:wsdl>
        <wsd:type>quasi-equivalent</wsd:type>
        <wsd:mapp>http://salatge-wifi:8001/AbstractExample/Abstract2Google-request.xsl</wsd:mapp>
        <wsd:unmapp>http://salatge-wifi:8001/AbstractExample/Google2Abstract-response.xsl</wsd:unmapp>
      </wsd:ReplicatedAccessPoint>
      <wsd:ReplicatedAccessPoint location="http://soap.search.msn.com:80/webservices.asmx" name="MSN">
        <wsd:wsdl>http://soap.search.msn.com/webservices.asmx?wsdl</wsd:wsdl>
        <wsd:type>quasi-equivalent</wsd:type>
        <wsd:mapp>http://salatge-wifi:8001/AbstractExample/Abstract2MSN-request.xsl</wsd:mapp>
        <wsd:unmapp>http://salatge-wifi:8001/AbstractExample/MSN2abstract-response.xsl</wsd:unmapp>
      </wsd:ReplicatedAccessPoint>
    </service>
</definitions>
```

Les parties du document surlignées représentent les points d'accès concrets associés au service abstrait. Pour chacun d'entre eux, les champs « mapp » et « unmapp » contiennent les urls des scripts XSLT correspondant aux fonctions de mappage. Pour des raisons de clarté, seules celles de Google vont être détaillées par la suite.

A.4.2 Les scripts XSLT de mappage

A.4.2.1 Conversion d'une requête abstraite vers une requête concrète

Le document XML ci-dessous représente le script XSLT d'une requête abstraite vers une requête concrète de Google :

```
<?xml version="1.0" encoding="utf-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:soap-enc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:ns1="http://www.iwsd.com">
   <xsl:template match="soap-env:Envelope">
      <xsl:copy>
         <xsl:apply-templates />
      </xsl:copy>
   </xsl:template>
   <xsl:template match="soap-env:Body">
      <xsl:copy>
         <xsl:apply-templates />
      </xsl:copy>
   </xsl:template>
   <xsl:template match="ns1:doAbstractSearch">
      <ns1:doGoogleSearch soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <xsl:attribute name="xmlns:ns1">urn:GoogleSearch</xsl:attribute>
         <xsl:apply-templates />
```

```
        <start href="#id0"/>
        <maxResults href="#id1"/>
        <filter href="#id2"/>
        <restrict xsi:type="xsd:string">
        </restrict>
        <safeSearch href="#id3"/>
        <lr xsi:type="xsd:string">
        </lr>
        <ie xsi:type="xsd:string">
        </ie>
        <oe xsi:type="xsd:string">
        </oe>
      </ns1:doGoogleSearch>
      <multiRef id="id2" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                xsi:type="xsd:boolean" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">true</multiRef>
      <multiRef id="id1" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                xsi:type="xsd:int" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">10</multiRef>
      <multiRef id="id0" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                xsi:type="xsd:int" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">0</multiRef>
      <multiRef id="id3" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                xsi:type="xsd:boolean" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">true</multiRef>
   </xsl:template>
   <xsl:template match="recherche">
      <q xsi:type="xsd:string">
         <xsl:attribute name="xmlns:ns1">urn:GoogleSearch</xsl:attribute>
         <xsl:apply-templates />
      </q>
   </xsl:template>
   <xsl:template match="cleGoogle">
      <key xsi:type="xsd:string">
         <xsl:attribute name="xmlns:ns1">urn:GoogleSearch</xsl:attribute>
         <xsl:apply-templates />
      </key>
   </xsl:template>
   <xsl:template match="cleMSN">
   </xsl:template>
</xsl:stylesheet>
```

A.4.2.2 Conversion d'une réponse concrète vers une réponse abstraite

Le document XML ci-dessous représente le script XSLT d'une réponse concrète de Google vers une réponse abstraite :

```
<?xml version="1.0" encoding="utf-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:soap-enc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:ns1="urn:GoogleSearch">
   <xsl:template match="soap-env:Envelope">
      <xsl:copy>
         <xsl:apply-templates />
      </xsl:copy>
   </xsl:template>
   <xsl:template match="soap-env:Body">
      <xsl:copy>
         <xsl:apply-templates />
      </xsl:copy>
   </xsl:template>
   <xsl:template match="ns1:doGoogleSearchResponse">
      <ns1:doAbstractSearchResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <xsl:attribute name="xmlns:ns1">http://www.iwsd.com</xsl:attribute>
         <xsl:apply-templates />
      </ns1:doAbstractSearchResponse>
   </xsl:template>
   <xsl:template match="resultElements">
      <listElements>
```

```
            <xsl:apply-templates />
        </listElements>
    </xsl:template>
    <xsl:template match="item">
        <item xsi:type="ns1:Resultat">
            <xsl:apply-templates />
        </item>
    </xsl:template>
    <xsl:template match="URL">
        <url>
            <xsl:apply-templates />
        </url>
    </xsl:template>
    <xsl:template match="title">
        <titre>
            <xsl:apply-templates />
        </titre>
    </xsl:template>
    <xsl:template match="summary">
        <resume>
            <xsl:apply-templates />
        </resume>
    </xsl:template>
    <xsl:template match="directoryCategories" />
    <xsl:template match="documentFiltering" />
    <xsl:template match="endIndex" />
    <xsl:template match="estimateIsExact" />
    <xsl:template match="estimatedTotalResultsCount" />
    <xsl:template match="cachedSize" />
    <xsl:template match="directoryTitle" />
    <xsl:template match="hostName" />
    <xsl:template match="relatedInformationPresent" />
    <xsl:template match="snippet" />
    <xsl:template match="searchComments" />
    <xsl:template match="searchQuery" />
    <xsl:template match="searchTime" />
    <xsl:template match="searchTips" />
    <xsl:template match="startIndex" />
    <xsl:template match="fullViewableName" />
</xsl:stylesheet>
```

A.5 Service Web avec fonctions de gestion d'état

Le document XML ci-dessous est le contrat WSDL du Service Web de gestion des « comptes bancaires ». Ce service contient une opération de crédit et de débit sur différents comptes clients du prestataire. Ce document contient les opérations de base nommées « credit » et « debit » plus les opérations « save_state » et « restore_state » capables respectivement de sauver et restaurer l'état du prestataire sur une réplique.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- edited with XMLSpy v2006 sp2 U (http://www.altova.com) by SALATGE (ME) -->
<wsdl:definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="urn:BanqueMeta" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:intf="urn:BanqueMeta"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns="urn:BanqueMeta" xmlns:y="urn:BanqueMeta"
targetNamespace="urn:BanqueMeta">
  <wsdl:types>
    <xs:schema targetNamespace="urn:BanqueMeta" xmlns:ns="urn:BanqueMeta">
      <xs:complexType name="State">
        <xs:sequence>
          <xs:element name="id" type="xs:int" nillable="false"/>
          <xs:element name="serviceInfo" type="ns:ServiceInformation"/>
          <xs:element name="content" type="xs:anyType" nillable="false"/>
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="Authentification">
        <xs:sequence>
          <xs:element name="login" type="xs:string" nillable="false"/>
          <xs:element name="password" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="ServiceInformation">
        <xs:sequence>
          <xs:element name="ServiceName" type="xs:string" nillable="true"/>
          <xs:element name="OperationName" type="xs:string" nillable="true"/>
          <xs:element name="location" type="xs:anyURI" nillable="true"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
  </wsdl:types>
  <wsdl:message name="restoreStateRequest">
    <wsdl:part name="userAccount" type="ns:Authentification"/>
    <wsdl:part name="state" type="ns:State"/>
  </wsdl:message>
  <wsdl:message name="restoreStateResponse">
    <wsdl:part name="idState" type="xs:integer"/>
  </wsdl:message>
  <wsdl:message name="exceptionRestoreState"/>
  <wsdl:message name="saveStateRequest">
    <wsdl:part name="userAccount" type="ns:Authentification"/>
    <wsdl:part name="serviceName" type="ns:ServiceInformation"/>
  </wsdl:message>
  <wsdl:message name="saveStateResponse">
    <wsdl:part name="state" type="ns:State"/>
  </wsdl:message>
  <wsdl:message name="exceptionSaveState"/>
  <wsdl:message name="debitRequest">
    <wsdl:part name="compte" type="xsd:string"/>
    <wsdl:part name="d" type="xsd:int"/>
  </wsdl:message>
  <wsdl:message name="debitResponse">
    <wsdl:part name="solde" type="xsd:int"/>
  </wsdl:message>
  <wsdl:message name="creditRequest">
    <wsdl:part name="compte" type="xsd:string"/>
    <wsdl:part name="c" type="xsd:int"/>
  </wsdl:message>
  <wsdl:message name="creditResponse">
    <wsdl:part name="solde" type="xsd:int"/>
```

```
   </wsdl:message>
   <wsdl:message name="consultRequest">
      <wsdl:part name="compte" type="xsd:string"/>
   </wsdl:message>
   <wsdl:message name="consultResponse">
      <wsdl:part name="solde" type="xsd:int"/>
   </wsdl:message>
   <wsdl:portType name="banqueTemp">
      <wsdl:operation name="debit" parameterOrder="compte d">
         <wsdl:input name="debitRequest" message="impl:debitRequest"/>
         <wsdl:output name="debitResponse" message="impl:debitResponse"/>
      </wsdl:operation>
      <wsdl:operation name="credit" parameterOrder="compte c">
         <wsdl:input name="creditRequest" message="impl:creditRequest"/>
         <wsdl:output name="creditResponse" message="impl:creditResponse"/>
      </wsdl:operation>
      <wsdl:operation name="consult" parameterOrder="compte">
         <wsdl:input name="consultRequest" message="impl:consultRequest"/>
         <wsdl:output name="consultResponse" message="impl:consultResponse"/>
      </wsdl:operation>
      <wsdl:operation name="restoreState">
         <wsdl:input name="restoreStateRequest" message="impl:restoreStateRequest"/>
         <wsdl:output name="restoreStateResponse" message="impl:restoreStateResponse"/>
         <wsdl:fault name="restoreFault" message="impl:exceptionRestoreState"/>
      </wsdl:operation>
      <wsdl:operation name="saveState">
         <wsdl:input name="saveStateRequest" message="impl:saveStateRequest"/>
         <wsdl:output name="saveStateRequest" message="impl:saveStateResponse"/>
         <wsdl:fault name="saveFault" message="impl:exceptionSaveState"/>
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:binding name="banqueTempSoapBinding" type="impl:banqueTemp">
      <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="debit">
         <soap:operation/>
         <wsdl:input>
            <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://DefaultNamespace"/>
         </wsdl:input>
         <wsdl:output>
            <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:BanqueMeta"/>
         </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="credit">
         <soap:operation/>
         <wsdl:input>
            <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://DefaultNamespace"/>
         </wsdl:input>
         <wsdl:output>
            <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:BanqueMeta"/>
         </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="consult">
         <soap:operation/>
         <wsdl:input>
            <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://DefaultNamespace"/>
         </wsdl:input>
         <wsdl:output>
            <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:BanqueMeta"/>
         </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="restoreState">
         <wsdl:input>
            <soap:body use="literal"/>
         </wsdl:input>
         <wsdl:output>
            <soap:body use="literal"/>
         </wsdl:output>
```

```
            <wsdl:fault name="restoreFault">
               <soap:fault name="exceptionRestoreState" use="literal"/>
            </wsdl:fault>
         </wsdl:operation>
         <wsdl:operation name="saveState">
            <wsdl:input>
               <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
               <soap:body use="literal"/>
            </wsdl:output>
            <wsdl:fault name="saveFault">
               <soap:fault name="exceptionSaveState" use="literal"/>
            </wsdl:fault>
         </wsdl:operation>
      </wsdl:binding>
      <wsdl:service name="banqueTempService">
         <wsdl:port name="banqueTemp" binding="impl:banqueTempSoapBinding">
            <soap:address location="http://localhost:8080/axis/BanqueMeta"/>
         </wsdl:port>
      </wsdl:service>
   </wsdl:definitions>
```

Références

[1] W3C, "Simple Object Access Protocol (SOAP) 1.2," http://www.w3.org/TR/soap/, 2003.

[2] W3C, "Web Services Definition Language (WSDL) 1.1," http://www.w3.org/TR/wsdl, 2001.

[3] OASIS, "Universal Description, Discovery, and Integration (UDDI) - Version 3," UDDI Spec Technical Committee Draft, http://uddi.org/pubs/uddi_v3.htm, October 2004.

[4] L. Maesano, C. Bernard, and X. L. Galles, *Services Web avec J2EE et .NET (Conception et Implementation)*: Edition Eyrolles, 2003.

[5] F. Salles, M. R. Moreno, J. C. Fabre, and J. Arlat, "Metakernels and fault containment wrappers," *29th IEEE International Symposium on Fault-Tolerant Computing (FTCS-29), Madison (USA)*, pp. 22-29, 15-18 June 1998.

[6] I. Algirdas Avižienis and B. R. Jean-Claude Laprie, Carl Landwehr, IEEE, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11-33, 2004.

[7] J.-C. Laprie, "Dependable Computing: Concepts, limits, challenges," *In 25th IEEE International Symposium on Fault-Tolerant Computing - Special Issue* pp. 42-54, 1995.

[8] A. Avizienis, "Design of Fault-Tolerant Computers," *Fall Joint Computer Conférence*, pp. 733-743, 1967.

[9] T. E. Anderson and P. A. Lee, "Fault Tolerance - Principles and practice," *Prentice Hall*, 1981.

[10] F. Leymann, "Web Services Flow Language (WSFL 1.0)," *IBM Software Group, www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf* Mai 2001.

[11] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. T. (Editor), I. Trickovic, and S. Weerawarana., "Business Process Execution Language for Web Services, Version 1.1," 2003.

[12] L. F. Cabrera, G. Copeland, M. Feingold, R. W. Freund, T. Freund, J. Johnson, S. Joyce, C. Kaler, J. Klein, D. Langworthy, M. Little, A. Nadalin, E. Newcomer, D. Orchard, I. Robinson, J. Shewchuk, and T. Storey, "Web Services Coordination (WSCoordination), Version 1.0," August 2005.

[13] IBM, "WS-Transaction," *http://www-128.ibm.com/developerworks/library/specification/ws-tx/*, November 2004.

[14] W3C, "Extensible Markup Language (XML) 1.0 (Third Edition)," *W3C Recommendation, http://www.w3.org/TR/REC-xml/*, 04 February 2004.

[15] J. Griffin, A. Associates, C. Hage, and C. H. Associates, "EDI Meets the Internet," *RFC 1865, http://www.ietf.org/rfc/rfc1865.txt*, January 1996.

[16] W3C, "Web Services Definition Language (WSDL) 1.2," W3C Working Draft, http://www.w3.org/TR/2003/WD-wsdl12-20030611/, http://www.w3.org/TR/2003/WD-wsdl12-20030611/.

[17] W3C, "XML Schema," 28 Octobre 2004.

[18] W3C, "SOAP Version 1.2 Part 0: Primer."

[19] W3C, "SOAP Version 1.2 Part 1: Messaging Framework."

[20] W3C, "SOAP Version 1.2 Part 2: Adjuncts."
[21] W3C, "SOAP Version 1.2 Specification Assertions and Test Collection."
[22] "Axis2," *The Apache SoftWare Foundation, http://ws.apache.org/axis2/*, 11 January 2006.
[23] "Axis 1.3," *The Apache Software Foundation, http://www.apache.org/dyn/closer.cgi/ws/axis/1_3*, October 2005.
[24] Microsoft, "ASP.NET, http://msdn.microsoft.com/asp.net/reference/," *Microsoft ASP.NET Developer Center*.
[25] "Apache Tomcat," *Apache Software Foundation, http://tomcat.apache.org/*.
[26] Macromedia, "ColdFusion MX 7, http://www.macromedia.com/software/coldfusion/."
[27] "Java Web Services Developer Pack (Java WSDP) Version 2.0, http://java.sun.com/webservices/jwsdp/index.jsp."
[28] "Internet Information Services," *Microsoft Windows Server, http://www.microsoft.com/WindowsServer2003/iis/default.mspx*, 2003.
[29] "BEA WebLogic," *BEA System, http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic* .
[30] "IBM WebSphere," *IBM Software Support, http://www-306.ibm.com/software/websphere/support/*.
[31] "Oracle. Oracle 8 (tm) Server Replication. ," 1997.
[32] "ZenWorks," *Novell, http://www.novell.com/fr-fr/products/zenworks/*.
[33] A. Asaduzzaman, "Building XML Web Services with PHP NuSOAP " *Dev Articles, http://www.devarticles.com/c/a/PHP/Building-XML-Web-Services-with-PHP-NuSOAP/*, 2003.
[34] "JBoss Application Server," *JBoss, The Professionnale Open Source Company, http://www.jboss.com/products/jbossas*.
[35] "Sun Java System Application Server," *Sun Microsystem, http://www.sun.com/software/products/appsrvr/index.xml*.
[36] T. Dierks and C. Allen, "The TLS Protocol --- Version 1.0. ," *IETF RFC 2246*, January 1999.
[37] S. Kent and R. Atkinson, "IPSEC - Security Architecture for the Internet Protocol " *RFC 2401*, November 1998.
[38] IBM and Micorosft, "WS-Security," http://www-106.ibm.com/developerworks/webservices/library/ws-secure, 2002.
[39] P. Koopman and J. DeVale, "Comparing the Robutness of POSIX Operatng Systems," *29th Annual International Symposium on Fault-Tolerant Computing*, 1999.
[40] J. Arlat, J.-C. Fabre, M. Rodriguez, and F. Salles, "Dependability of COTS microkernel-based systems," *IEEE Transactions on computer Systems*, vol. Vol n°51, pp. 138-163, 2002.
[41] E. Marsden, J.-C. Fabre, and J. Arlat, "Dependability of CORBA systems: service characterization by fault injection," *in 21st IEEE Symposium on Reliable Distributed Systems (SRDS'2002), Osaka (Japon)*, pp. pp.276-285, 2002.
[42] E. Marsden, "Caractérisation de la sûreté de fonctionnement de systemes à base d'intergiciel.," in *INP Toulouse*. Toulouse: LAAS-CNRS, 2004.
[43] N. Looker, M. Munro, and J. Xu, "WS-FIT: A Tool for Dependability Analysis of Web Services," *1st Workshop on Quality Assurance and Testing of Web-Based Applications, COMPSAC, Hong Kong*, 28-30 Sep 2004.

[44] N. Looker and J. Xu, "Assessing the Dependability of OGSA Middleware by Fault Injection," *In 22nd International Symposium on Reliable Distributed Systems (SRDS'03), Florence, Italy*, pp. 293-302, 2003.

[45] N. Looker, L. Burd, S. Drummond, J. Xu, and M. Munro, "Pedagogic Data as a Basis for Web Service Fault Models," *IEEE International Workshop on Service-Oriented System Engineering, Beijing, China*, October 20-21, 2005.

[46] D. Cotroneo, C.Di Flora, and S. Russo, "Improving Dependability of Service Oriented Architectures for Pervasive Computing," *In The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*, 2003.

[47] A. Gorbenko, V. Kharchenko, P. Popov, A. Romanovsky, and A. Boyarchuk, "Development of Dependable Web Services out of Undependable Web Components," *Schoolof Computing Science, University of Newcastle upon Tyne, Tecnical Report Series CS-TR-863*, october 2004.

[48] L. Silva, H. Madeira, and J. G. Silva, "Software Aging and Rejunevation in a SOAP-based Server," *In Proceedings of IEEE Network Computing and Applications*, 2006.

[49] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, "Web Services Agreement Specification (WS-Agreement)," *Draft 18 Version 1.1*, May 2004.

[50] M. Yu-jie, C. Jian, Z. Shen-sheng, and Z. Jian-hong, "Interactive Web service choice-making based on extended QoS model," *Proceedings of the 2005 The Fifth International Conference on Computer and InformationTechnology (CIT'05)*, pp. 1130 - 1134 Sept. 2005.

[51] M. Merzbacher and D. Patterson, "Measuring End-User Availability on the Web: Practical Experience," *International Performance and Dependability Symposium*, June 2002.

[52] D. A. Menascé, "QoS Issues in Web Services," *IEEE Internet Computing* vol. 6, pp. 72-75

[53] S. Y. D. K. S. Han;, "WS-QDL containing static, dynamic, and statistical factors of Web services quality," *Proceedings of the IEEE International Conference on Web Services (ICWS'04), 2004.*, pp. 808 - 809 July 2004.

[54] O. Hasan and B. Char, "A deployment-ready solution for adding quality-of-service features to web services " *The 2004 International Research Conference on Innovations in Information Technology*, 2004.

[55] N. Aghdaie and Y. Tamir, "Client-Transparent Fault-Tolerant Web Service," *20th IEEE International Performance, Computing, and Communications Conference*, pp. 209-216, 2001.

[56] N. Aghdaie and Y. Tamir, "Fast Transparent Failover for Reliable Web Service," *Proceedings of the international Conference on Parallel and distributed computing and systems*, 2003.

[57] N. A. a. Y. Tamir, "Implementation and Evaluation of Transparent Fault-Tolerant Web Service with Kernel-Level Support," *11th IEEE International Conference on Computer Communications and Networks*, pp. 63-68, 2002.

[58] N. Aghdaie and Y. Tamir, "Performance Optimizations for Transparent Fault-Tolerant Web Service," *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, 2003.

[59] S. Todd, F. Parr, and M. H. Conner, "A Primer for HTTPR," *IBM Internet Software. Austin*, 2002.

[60] E. Alwagait and S. Ghandeharizadeh, "DeW : A Dependable Web Services Framework," *14th International Workshop on Research Issues on Data Engineering:*

Web Services for E-Commerce and E-Government Applications (RIDE'04), Boston, Massachusetts, 2004.

[61] M. Jeckle and B. Zengler, "Active UDDI - an Extension to UDDI for Dynamic and Fault-Tolerant Service Invocation," *Web, Web-Services, and Database Systems 2002*, pp. 91-99, 2003.

[62] D. Liang, C.-L. Fang, and C. Chen, "FT-SOAP: A Fault-tolerant web service," *Tenth Asia-Pacific Software Engineering Conference, Chiang Mai, Thailand*, 2003.

[63] R. Murty, "JULIET: a distributed fault tolerant load balancer for .NET Web services," *Proceedings of the IEEE International Conference on Web Services (ICWS'04), 2004.*, vol. 778 - 781, 6-9 July 2004.

[64] V. Dialani, S. Miles, L. Moreau, D. D. Roure, and M. Luck, "Transparent fault tolerance for web services base architectures," *In Proceedings of 8th International Europar Conference (EURO-PAR'02) 2400 (-), Paderborn, Germany.*, pp. pages pp. 889-898, 2002.

[65] G. T. Santos, L. C. Lung, and C. Montez, "FTWeb: A Fault Tolerant Infrastructure for Web Services," *In the Proceedings of the 2005 Ninth IEEE International EDOC Enterprise Computing Conference (EDOC'2005)* 2005.

[66] P. Narasimhan and J. Garon, "Fault Tolerant CORBA Tutorial," *OMG Workshop on Embedded & Real-time Distributed Object Systems*, 2001.

[67] X. Defago, A. Schiper, and P. Urban, "Totally ordered broadcast and multicast algorithms: a comprehensive survey," *Technical Report DSC/2000/036, Dept. of Communication Systems, EPFL*, 2000.

[68] SUN, "Web Services Reliable Messaging TC WS-Reliability," http://www.oasis-open.org/committees/download.php/5155/WS-Reliability-2004-01-26.pdf, 2003.

[69] X. Ye and Y. Shen, "A middleware for replicated Web services " *Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, 11-15 July 2005

[70] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan, "Thema: Byzantine-fault-tolerant middleware for Web-service applications," *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium* pp. Page(s):131 - 140 2005.

[71] R. Rodrigues, M. Castro, and B. Liskov, "BASE: Using Abstraction to Improve Fault Tolerance," *Proceedings of the 18th {ACM} Symposium on Operating System Principles*, vol. 35, pp. 15--28, 2001.

[72] N. Looker, M. Munro, and J. Xu, "Increasing Web Service Dependability Through Consensus Voting," *the 2nd International Workshop on Quality Assurance and Testing of Web-Based Applications, COMPSAC, Edinburgh, Scotland*, July 25-28, 2005.

[73] Microsoft, "Web Services Reliable Messaging Protocol (WS-ReliableMessaging)," http://www-106.ibm.com/developerworks/webservices/library/ws-rm/, 2004.

[74] W3C, "Web Service Choreography Interface (WSCI) 1.0," *W3C Note, http://www.w3.org/TR/wsci/*, 8 August 2002.

[75] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection," *ACM Transactions on Software Engineering and Methodology*, 1997.

[76] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller, "Devil: An IDL for hardware programming.," *In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, San Diego, California*, pp. pages 17-30, Oct. 2000.

[77] L. R. a. G. Muller, "Improving driver robustness: an evolution of the Devil approach.," *In The International Conference on Dependable Systems and Networks, Göteborg, Sweden, IEEE Computer Society*, pp. 131-140, July 2001.
[78] S. Thibault, C. Consel, and G. Muller, "Safe and efficient active network programming," *In 17th IEEE Symposium on Reliable Distributed Systems, West Lafayette, Indiana*, pp. 135-143, Oct. 1998.
[79] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and C. Pu, "Spidle: a DSL approach to specifying streaming applications. ," *Proceedings of the second international conference on Generative programming and component engineering, Erfurt, Germany* pp. 1-17, 2003
[80] C. Consel and R. Marlet, "Architecturing Software Using A Methodology for Language Development," *In International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '98), LNCS 1490, Pisa, Italy*, pp. 170-194, 1998.
[81] A. v. Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM SIGPLAN Notices.*, vol. 35(6), 2000.
[82] S. P. Harbison and G. L. Steele, "C, a reference manual (2nd ed.)," *Prentice-Hall, Inc.*, 1986.
[83] D. L. McGuinness and F. v. Harmelen., "Web ontology language (OWL) overview," *http://www.w3.org/TR/owl-features/, W3C Recommendation.*, February 2004.
[84] D. Martin, M. Burstein, O. Lassila, M. Paolucci, T. Payne, and S. McIlraith, "Describing Web Services using OWL-S and WSDL," October 2003.
[85] W. C. Recommendation, "XSL Transformations (XSLT)," 16 November 1999.
[86] J. D. Davidson and D. Coward, "Java™ Servlet Specification, v2.2," *Sun Microsystem*, 1999.
[87] B. Randell, "System structures for software fault tolerance," *IEEE Transactions on Software Engineering*, 1975.
[88] D. Kristol and L. Montulli, "HTTP State Management Mechanism," *RFC 2109, http://www.ietf.org/rfc/rfc2109.txt*, 1997.
[89] C. Kohlhoff and R. Steele, "Evaluating SOAP for High Performance Business Applications: Real-Time Trading Systems," *in WWW2003, Budapest, Hungary*, 2003.
[90] N. Salatge and J.-C. Fabre, "DeWeL: Un langage dédié pour la sûreté de fonctionnement des services web," *New Technologies for Distributed Systems (NOTERE'06), Toulouse, France*, June 6-9, 2006.
[91] N. Salatge and J.-C. Fabre, "DeWeL: a language support for fault tolerance in service oriented architectures," *International Workshop on Engineering of Fault Tolerant Systems (EFTS'2006), Luxembourg (Luxembourg)*, Juin 2006.
[92] B. Meyer, "Eiffel: programming for reusability and extendibility," *SIGPLAN Not.*, vol. 22, pp. 85-87, 1987.
[93] G. Kiczales, J. d. Rivières, and D. G. Bobrow, "The art of the metaobject protocol," *MIT Press*, 1991.
[94] G. Kiczales, Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J, "Aspect-Oriented Programming," *In Proc. of ECOOP*, 1997.
[95] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ " *In Proc. of ECOOP* 2001.
[96] L. Réveillère, "Approche langage au développement de pilotes de périphériques robustes. Note: Best Ph.D. Thesis award from ACM SIGOPS France.," Thèse de doctorat, Université de Rennes 1, France, December 2001.

[97] S. Thibault, "Domain-Specific Languages: Conception, Implementation, and Application. ," *PhD Thesis. University of Rennes 1, France*, October 1998.
[98] J. Neighbors, "Software Construction Using Components," *Ph.D. Thesis, ICS-TR-160, University of California at Irvine*, 1980.
[99] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler, "An Empirical Study of Operating System Errors," *Symposium on Operating Systems Principles*, pp. 73-88, 2001.
[100] C. Consel and L. Reveillère, "A Programmable Client-Server Model: Robust Extensibility via DSLs," *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003), IEEE Computer Society Press, Montréal, Canada*, pp. 70-79, November 2003.
[101] L. Réveillère, "Approce langage au développement de pilotes de périphériques robustes.," *Thèse de doctorat, Université de Rennes 1, France*, Dec. 2001.
[102] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang., "Cyclone: A Safe Dialect of C," *USENIX Annual Technical Conference, Monterey, CA* June 2002.
[103] "Xercs C++ Documentation," *The Apache SoftWare Foundation, http://xml.apache.org/xerces-c/pdf.html*.
[104] S. Loughran and E. Smith, "Rethinking the Java SOAP Stack," *IEEE International Conference on Web Services (ICWS) 2005, Orlando, Florida, USA*, 12-15 July 2005.
[105] R. Fielding, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1," *RFC 2616, http://www.w3.org/Protocols/HTTP/1.1/rfc2616.pdf*, 1999.
[106] J. Gray, "Why Do Computers Stop and What Can Be Done About It?," *Symposium on Reliability in Distributed Software and Database Systems*, pp. 3-12, 1986.
[107] Graham D Parrington, Santosh K Shrivastava, Stuart M Wheater, and M. C. Little, "The Design and Implementation of Arjuna," *USENIX Computing Systems Journal*, vol. 8, pp. 253-306, Summer 1995.
[108] L. F. Cabrera, G. Copeland, M. Feingold, R. W. Freund, T. Freund, J. Johnson, S. Joyce, C. Kaler, J. Klein, D. Langworthy, M. Little, A. Nadalin, E. Newcomer, D. Orchard, I. Robinson, T. Storey, and S. Thatte, "Web Services Atomic Transaction (WSAtomicTransaction)," *Version 1.0, http://www-128.ibm.com/developerworks/library/specification/ws-tx/#atom*, August 2005.
[109] L. F. Cabrera, G. Copeland, M. Feingold, R. W. Freund, T. Freund, J. Johnson, S. Joyce, C. Kaler, J. Klein, D. Langworthy, M. Little, A. Nadalin, E. Newcomer, D. Orchard, I. Robinson, J. Shewchuk, and T. Storey, "Web Services Coordination (WSCoordination)," *Version 1.0, http://www-128.ibm.com/developerworks/library/specification/ws-tx/#atom*, August 2005.
[110] F. Cappello, F. Desprez, M. Dayde, E. Jeannot, Y. Jegou, S. Lanteri, N. Melab, R. Namyst, P. Primet, O. Richard, E. Caron, J. Leduc, and G. Mornet, "Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform " *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, Grid'2005, Seattle, Washington, USA* November 13-14, 2005.
[111] A. L. Peterson and A. T. Roscoe, "The design principles of PlanetLab " *SIGOPS Oper. Syst. Rev. , ACM Press*, vol. 40, pp. 11-16, 2006
[112] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Mutation analysis," *Technical Report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA* September 1979.
[113] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: help for the practicing programmer," *IEEE Computer, 11 (4)* pp. 34-41, 1878.

[114] R. DeMillo, D. Guindi, K. King, M. M. McCracken, and J. Offutt, "An Extended Overview of the Mothra Software Testing Environment," *2nd Workshop on Software Testing, Verification, and Analysis, Banff, Canada* pp. 142-151 July 1988.
[115] M. R. Girgis and M. R. Woodward, "An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria," *Proceedings Workshop on Software Testing, Banff*, pp. 64-73, July 1986.
[116] H. Agrawal, R. Demillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, and E. Spafford, "Design of Mutant Operators for the C Programming Language," *Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana,SERC-TR-41-P*, 1989.
[117] R. Siblini and N. Mansour, "Testing Web services," *Computer Systems and Applications, 2005. The 3rd ACS/IEEE International Conference on 2005*, pp. 135

Résumé :

Les Services Web (SW) constituent la technologie de base pour le développement d'Architectures Orientées Services (AOS). Ces architectures, de plus en plus répandues sur le Net, permettent de mettre en place des applications semi-critiques à échelle planétaire. Elles se basent sur la notion de relation de "service" formalisée par un contrat qui unit le client et le prestataire de services. Dans ce type d'applications, les développeurs d'applications orientées services regardent les Services Web comme des COTS (Component Off-The Shell) et ignorent donc leurs implémentations et leurs comportements en présence de fautes.

Dans ce but, cette thèse introduit, dans ce nouveau contexte, la notion de « connecteurs spécifiques de tolérance aux fautes » (SFTC – Specific Fault Tolerance Connectors) capable d'implémenter des applications sûres de fonctionnement à partir de Services Web supposés non-fiables. Composants logiciels insérés entre les clients et les prestataires, les SFTC implémentent des filtres et différentes techniques de détection d'erreurs ainsi que des mécanismes de recouvrement qui sont déclenchés quand les Services Web ne satisfont plus les caractéristiques de sûreté demandées. L'originalité de cette approche est d'utilisé une particularité intéressante du Web qui est la redondance inhérente des services qui s'y trouvent. Cette propriété a permis de définir le concept de Services Web Abstrait (SWA) et de mettre en place des stratégies de recouvrement à l'aide de services équivalents. Ainsi, les « connecteurs » et les « SWA », introduits dans les architectures orientées services, fournissent une approche plus adaptable pour permettre, à des mécanismes de sûreté, d'être définis au cas par cas pour une utilisation donnée du Service Web et d'être modifiés selon la criticité de l'application.

Ainsi, mes travaux de recherches ont permis de fournir aux développeurs d'Architectures Orientées Services:

1) le langage nommé DeWeL pour décrire les caractéristiques de sûreté de fonctionnement du connecteur. Ce langage dédié à la réalisation de connecteur impose de fortes restrictions inspirées du monde des logiciels critiques du ferroviaire et de l'avionique afin de diminuer de façon drastique des fautes de développement.
2) l'infrastructure IWSD pour dynamiquement contrôler et exécuter les connecteurs dans des applications critiques. Cette plateforme, réalisé en mode duplex pour tolérer les fautes matérielles et logicielles, permet d'exécuter de façon sûres les connecteurs crées et d'obtenir également des informations sur le caractère non-fonctionnel et opérationnel des Services Web ciblés. Ces informations sont capitales pour permettre aux développeurs de connecteurs d'affiner leurs actions de tolérances aux fautes pour l'application ciblée.

Environ deux cents connecteurs ont été implémentés afin de réaliser des tests sur la performance et la robustesse du langage et de la plate-forme permettant ainsi de valider cette approche capable de déployer des applications orientées services semi-critiques tolérant les fautes à l'aide de SWA.

Mots clés : Services Web (XML, SOAP, WSDL, UDDI, BPEL4WS), Architectures Orientées Services (AOS), Domain-Specific Language (DSL), Tolérance aux fautes, Connecteurs, Programmation par aspect, Serveurs d'applications.

Abstract :

Web Services (WS) technology is the basis for the development of Service Oriented Architectures (SOA). These architectures are increasingly widespread on the Net and enable developers to implement semi-critical planetary applications. Such applications are based on the notion of service and its attached contract. The contract links a client and a service provider. In this kind of applications, application developers look at Web Services as COTS (Commercial Off-The-Shelf) components, consequently they ignore their implementation and their behaviour in the presence of faults.

The thesis proposes the notion of Specific Fault Tolerance Connectors (SFTC) to implement dependable applications out of unreliable Web Services. The connectors intercept client-provider requests and implement filtering, error detection techniques (e.g. runtime assertions) together with recovery mechanisms that are triggered when the WS does not satisfy anymore the dependability specifications. The originality of this approach relies on using an interesting feature of the Web that is the inherent redundancy of services. To take advantage of this, we define the concept of Abstract Web Services (AWS) to implement recovery strategies using equivalent services. Thus, “connectors” and “AWS” allow dependability mechanisms to be defined on a case-by-case basis for a given WS usage and possibly dynamically changed according to the needs.

Two specific techniques and tools have been designed and implemented to help developers of Services Oriented Architectures:

1) A domain specific language named DeWeL (DEpendable Web services Language) describing the dependability features of a connector. This language is devoted to the realization of connectors. It imposes strong restrictions enforced by standards for critical railway and avionics software in order to reduce software development faults.

2) A support infrastructure named IWSD (Infrastructure for Web Services Dependability) to dynamically manage and run connectors in real applications. This platform is implemented in duplex mode to tolerate crash faults and provides core services. In particular, it provides support to run connectors and get information on the non-functional and operational behaviour of the targeted Web Services. This information is essential to help developers of connector to adjust their fault-tolerance actions to specific application needs.

Approximately two hundred connectors have been implemented in order to realize performance and robustness tests of the language and the platform. These experiments show the interest of our approach to deploy fault-tolerant service oriented applications using AWS.

Keywords: Web Services (XML, SOAP, WSDL, UDDI, BPEL4WS), Services Oriented Architectures (SOA), Domain-Specific Language (DSL), Fault Tolerance, Connectors, Aspect Oriented Programmation, Application Server.