Tolérance aux fautes dans SOA

Auteurs : N. Faci et H. Abdeljelil

Dans un premier temps, nous introduisons les concepts de base des services Web et sûreté de fonctionnement. Nous présentons ensuite l'état de l'art des travaux réalisés sur la tolérance aux fautes dans les Services Web. Enfin, nous discutons les différentes approches pour se positionner par rapport aux solutions existantes.

0.1 Généralités

0.1.1 Architecture Orientée-Service et Services Web

L'Architecture Orientée-Service (*en anglais*, Service-Oriented Architecture (SOA)) est un paradigme architectural présentant un intérêt particulier pour les technologies de l'information et le domaine de l'entreprise. Le Consortium World Wide Web (W3C), chargé du développement des standards sur le Web et leur évolutivité, donne la définition suivante de SOA : "*collections structurées de composants logiciels, appelés services, à invoquer avec des descriptions d'interface pouvant être publiées et découvertes*" **[?]**.

0.1.1.1 Principes SOA

Les composants de base de SOA sont : le fournisseur de service, le registre de service, et le client de service. Les fournisseurs de service sont des applications logicielles fournissant un service au client. Les fournisseurs publient une description des services offerts via un registre de services. Les clients consomment des services. Une application orientée-service peut agir comme fournisseur et consommateur de service à la fois. Les clients doivent être en mesure de trouver la description des services dont ils ont besoin et se lier à eux. Dans **[?]**, T. Erl propose plusieurs principes de conception sous-jacents à SOA comme suit :

— *Contrat de service standardisé* : Le contrat de service définit un accord entre le fournisseur et le consommateur, composé des éléments suivants : i) une représentation technique du service (i.e., son interface) comme les opérations, leurs paramètres d'entrée et de sortie et les contraintes sur les entrées ; ii) une description informelle des opérations sous formes de règles et contraintes d'utilisation du service (e.g., volume des données échangées) ; iii) un niveau de service (QoS et SLA) précisant les engagements du service (e.g., temps de réponse maximum attendu, plages horaires d'accessibilité, le temps de reprise après interruption, les procédures mises en œuvre en cas de panne, les procédures de prise en charge du support).

— *Faible couplage entre les services* : Les services maintiennent une relation minimisant les dépendances. Ils n'ont plus besoin d'un système de middleware réparti commun pour communiquer, mais seulement de **protocoles et technologies de communication** interopérables sur Internet.

— *Abstraction de service* : En dehors des différentes descriptions dans le contrat de service, les services cachent leur logique au monde extérieur. Le contrat de service ne doit contenir que les informations essentielles à son invocation.
— *Services réutilisables* : La logique est divisée en différents services avec comme objectif de promouvoir la réutilisation. Les services peuvent être ainsi partagés parmi différents domaines.
— *Services autonomes* : Les services contrôlent la logique d'exécution qu'ils encapsulent. Plus ce contrôle est fort, plus l'exécution d'un service est prédictible.
— *Services dépourvus d'état* : Les services minimisent la consommation de ressources en déléguant la gestion des informations d'état quand cela est nécessaire.
— *Services découvrables* : La description des services est complétée par un ensemble de métas données permettant leur découverte et leur interprétation de manière efficace et appropriée.
— *Services composables* : Les services sont conçus de manière à participer à des compositions de services. Les services peuvent être orchestrés pour améliorer l'agilité du processus métier implémenté.

0.1.1.2 Services Web

Les services Web sont l'une des technologies implémentant les principes SOA. Un service Web est une partie d'application faiblement couplée, indépendante, et mise à disposition sur le Web pouvant être décrite, publiée, découverte, coordonnée (ou composée) et configurée en utilisant des artefacts (e.g., XML [**?**]) pour une simplicité de développement d'applications hétérogènes dans des environnements distribués. Les services Web permettent à des applications de travailler ensemble à travers des protocoles Internet standards (e.g., HTTP - Hypertext Transfer Protocol [**?**]) afin d'automatiser des opérations métier sans l'intervention humaine [**?**, **?**]. Dans les services Web, les clients et les services sont des systèmes à part entière et indépendants des uns des autres. Les services sont normalement autonomes, développés et déployés par différents fournisseurs de service.

Décrire un service Web revient à définir l'ensemble des fonctionnalités sous formes d'opérations offertes par le service (i.e., son interface) à l'aide de langages de description tel que *WSDL* (Web Service Definition Language [**?**]) ou de ressources identifiables *URI* (Uniform Resource Identifier). Les services Web communiquent entre eux via des protocoles standard de communication tels que *SOAP* (Simple Object Access Protocol [**?**]) reposant sur le formalisme *XML* pour le format des messages échangés ou *REST* (Representational state transfer [**?**])

Les clients peuvent accéder et localiser les services Web dans un annuaire tel que *UDDI* (Universal Description Discovery and Integration) [**?**]. Un annuaire *UDDI* permet de localiser sur le réseau, un service Web accessible par l'intermédiaire du protocole *SOAP*.

La figure 1 montre les différentes étapes nécessaires à l'invocation d'un service Web :

Publication. Les fournisseurs mettent à disposition des services sur le Web et enregistrent la description de ces services dans un annuaire (étape 1).

Découverte. Le consommateur à la recherche d'un service répondant à ses besoins doit découvrir un service de *matching* parmi tous les fournisseurs. Il doit être en mesure de décrire clairement ses besoins (i.e., sa requête). La description du service désiré est comparée avec celle du service proposé. Si les deux services s'appartient, le service requis a été découvert avec succès. (étape 2)

Composition. Il est possible qu'aucun *matching* n'existe avec les besoins du consommateur. Dans ce cas, il est possible de composer des services existants pour répondre à ces besoins. La composition de service décrit la combinaison de deux ou plusieurs services en un service plus complexe. Il existe deux approches de composition de services décrites comme suit :

— *Chorégraphie*. Elle décrit la collaboration de services définie par un ensemble de règles d'interaction (ou protocoles) parmi les services sans aucune entité centrale de contrôle. Un langage largement utilisé est le Web Service Choreography Interface (WSCI) [**?**].

— *Orchestration*. Elle décrit la collaboration de services contrôlée par un composant central, appelé moteur de la composition. Ce moteur connait les règles pour les composer [**?**]. BPEL (acronyme de "'Business Process Execution Language"') utilise cette approche [**?**].

Binding. Après la découverte du service approprié délivrant la fonctionnalité souhaité, le service consommateur se lie à ce service pour l'exécution. A ce stade, des paramètres de sécurité (e.g., authentification, autorisation) doivent être configurés de part et d'autre. (étape 3)

Execution. Une fois le binding effectué, le service ou la composition de services peuvent être exécuté(e). Les paramètres d'entrée sont transmis au fournisseur de service et ceux de sortie sont renvoyés au consommateur. (étape 4)

Depuis leur apparition, les services Web sont de plus en plus demandés surtout dans le développement d'applications critiques relatives aux domaines tels que le contrôle aérien et la défense. Cependant ces applications peuvent être sujettes à des défaillances pouvant ainsi provoquer de réelles catastrophes humaines et/ou financières. Ainsi, assurer un fonctionnement sûr des services Web en dépit de l'occurrence de fautes devient l'une des questions les plus cruciales à laquelle nous nous intéressons dans cette thèse.

0.1.2 Sûreté de fonctionnement (SdF)

Cette section a pour objectif d'introduire les concepts liés au domaine de la SdF permettant de cerner plus précisément la tolérance aux fautes. Selon Avizienis et al. [**?**] la sûreté de fonctionnement est l'aptitude ou propriété d'un système à délivrer un service

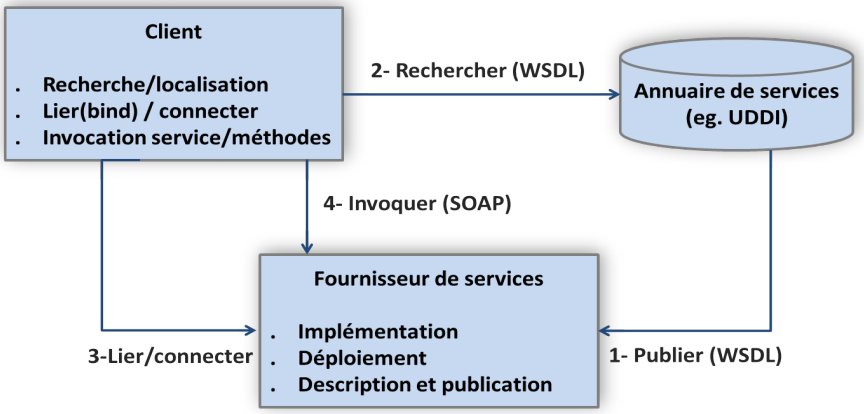

FIGURE 1 – **Interactions au sein d'une architecture orientée service**

de confiance justifiée. La notion de service délivré par un système correspond à son comportement perçu par son, ou ses utilisateurs (i.e., autre système en interaction avec celui considéré). Le service est dit *correct* si le service délivré accomplit la fonction du système décrite par la spécification fonctionnelle (i.e., ce à quoi le système est destiné).

0.1.2.1 Terminologie : défaillance, erreur, faute

Melliar-Smith [**?**] est le premier à avoir fait la distinction entre la défaillance, l'erreur et la faute. Lorsque le service délivré dévie du service *correct*, un évènement, appelé *défaillance (panne)* survient. Il existe plusieurs raisons possibles liées à cette déviation : i) sa non-conformité à la spécification, ou ii) l'inadéquation de la spécification par rapport à la fonction du système. Une *erreur* est la partie de l'état du système susceptible d'entraîner une défaillance. La **propagation** d'erreur, appelée *délai de latence* se manifeste avant l'apparition d'une défaillance. La *faute* est la cause (ou **activation**) supposée ou adjugée d'une erreur et peut être le résultat (ou **conséquence**) d'une défaillance.

Pour des raisons d'abstraction, un système est souvent décomposé en sous-systèmes appelés composants, pouvant être à leur tour des systèmes. Pour un niveau d'abstraction donné, les composants sont considérés comme élémentaires. Une *défaillance* d'un de ces composants peut être source de *faute* pour le système composite, pouvant causer une *erreur* dans le système. La chaîne cause (*faute* $\longrightarrow$ *erreur* $\longrightarrow$ *défaillance*) devient alors récursive.

0.1.2.2 Classification des défaillances

Les défaillances des composants d'un système composite sont considérées comme les fautes à combattre. Avizienis et al. [**?**] caractérisent ces défaillances selon plusieurs points de vue (voir 2) :

- **Domaine.** Il permet de spécifier quatre types de défaillances selon le comportement du composant :

— **Défaillance franche (Crash)**. Le composant cesse toute interaction avec les autres composants du système. La notion d'interaction dépend du modèle de système considéré (e.g., appel de procédure, envoi de message, écriture dans une mémoire partagée) ;
— **Défaillance temporelle (Omission)**. Le composant interagit avec les autres composants du système en dehors des fenêtres temporelles attendues. Cela concerne aussi bien des interactions ayant lieu trop tard (i.e., échéance manquée) ou trop tôt. La défaillance par crash est une défaillance temporelle, dans le sens où toutes les interactions seront réalisées trop tard (ou jamais en l'occurrence).
— **Défaillance en valeur**. Le composant interagit avec les autres composants du système avec des valeurs incorrectes.
— **Défaillance Byzantine**. Le composant défaille de manière arbitraire. Ce mode de défaillance regroupe l'ensemble de tous les modes de défaillance ainsi que leurs combinaisons (e.g., valeur erronée trop tard).

- **Détectabilité.** Elle réfère à la propriété qu'une défaillance soit signalée ou non à l'utilisateur du système. Un mécanisme de détection permet par exemple de vérifier l'exactitude du résultat délivré. La signalisation d'une défaillance peut être communiquée à tord à l'utilisateur. La signalisation est alors dite *fausse alerte*.
- **Consistance.** Les défaillances sont dites *consistantes* si le service incorrect est perçu de façon identique par tous les utilisateurs du système. Sinon elles sont *inconsistantes*. Par exemple, les défaillances d'omissions sont consistantes contrairement aux défaillances Byzantines.
- **Conséquence.** Les conséquences de défaillances sont classées selon leur niveau de gravité (ou sévérité) dépendant largement du type d'applications. En général, deux niveaux sont distingués (i.e., *mineur* et *catastrophique*) et se définissent par rapport au bénéfice fourni par le service rendu en absence de défaillance, et les conséquences de défaillances. Les défaillances sont dites *mineures* lorsque leurs conséquences ont un coût similaire à celui de la prestation du service correct. Les défaillances sont dites *catastrophiques* lorsque le coût de la défaillance est conséquente (e.g., perte humaine, crash financier).

0.1.2.3 Classification de fautes

Avizienis et al. [**?**] classifient les fautes élémentaires selon huit points de vue : *phase de création/occurrence*, *frontière du système*, *cause et persistance de la faute*, *dimension du système*, *objectif du concepteur/développeur ainsi que son intention et sa compétence* (voir 3).

Les fautes peuvent être soit introduites au moment de la conception et du développement du système (appelées *fautes de conception*) soit apparaître au moment de l'exécution du système (appelées *fautes opérationnelles*). Les fautes sont *internes* (resp. *externes*) si elles

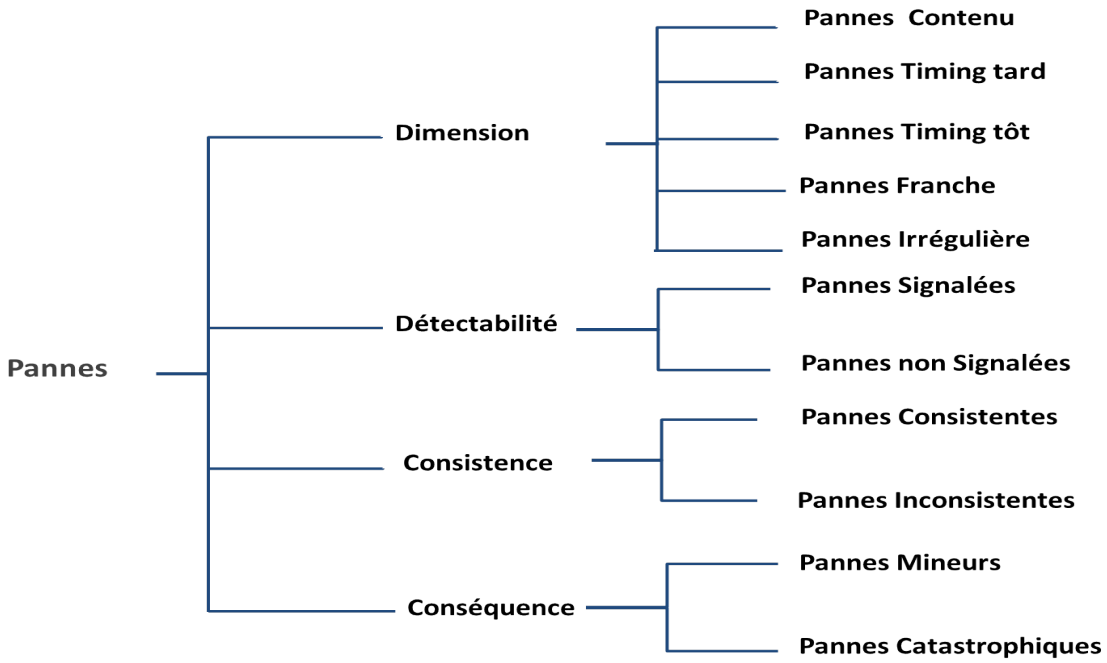

FIGURE 2 – **Classification des Défaillances dans les service Web**

sont produites à l'intérieur (resp. extérieur) du système. La cause des fautes est *naturelle* si elles sont provoquées par un phénomène naturel. Elle est *non-naturelle* si l'être humain intervient dans le processus de création/occurrence de fautes. Les fautes persistent soit temporairement (dites *transitoires*) soit définitivement (dites *permanentes*). Par exemple, les composants défaillants re-fonctionnent normalement après la disparition des causes de la faute.

Les fautes peuvent être aussi définies par rapport à l'environnement de développement ou d'exécution (appelé dimension) du système. Ainsi, les fautes peuvent être soit de nature *logicielle* (e.g., composants défaillants) soit *matérielle* (e.g., machines défaillantes). Les fautes introduites par des humains sont *malicieuses* si l'objectif de ces derniers est de nuire au système tel que : accéder à des données confidentielles, ou dégrader le fonctionnement du système. Elles sont *non-malicieuses* dans le cas contraire. Les fautes sont *intentionnelles* si le concepteur/développeur a sciemment pris de mauvaises décisions mais sans aucun objectif de nuire le système. Les fautes sont accidentelles si le concepteur/développeur a pris des décisions dont les effets de bord sont involontaires. Les fautes peuvent être aussi dues à un manque de compétences de la part du concepteur/développeur.

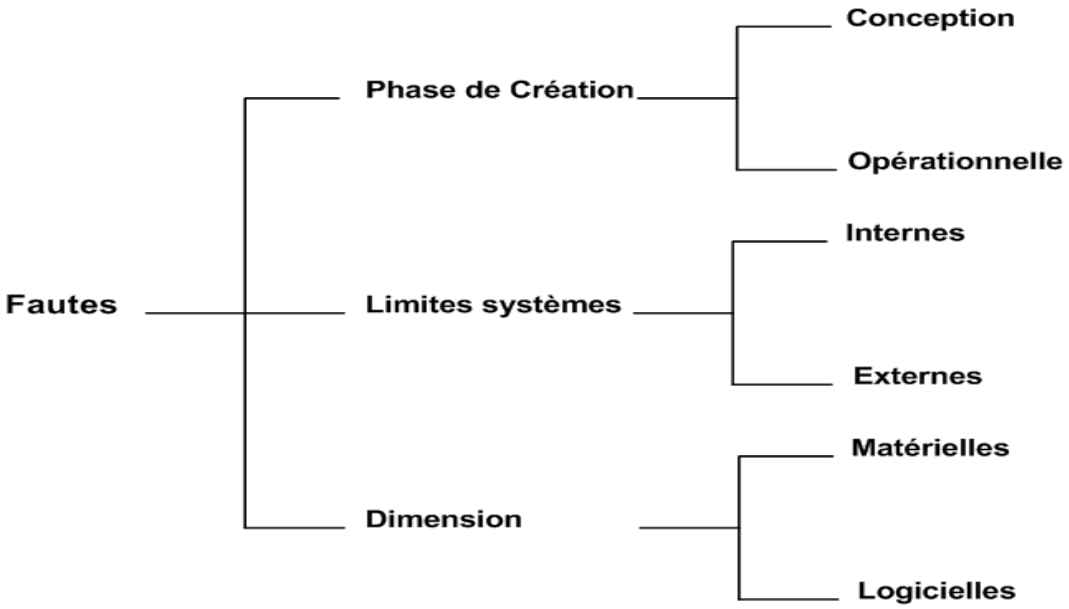

FIGURE 3 – **Classification des fautes**

0.1.2.4 Tolérance aux fautes en bref

La tolérance aux fautes est l'une des méthodes de SdF à laquelle nous nous intéressons dans cette thèse. Elle consiste à éviter la défaillance du système en utilisant des mécanismes *de détection et recouvrement/compensation d'erreur* causée par la défaillance d'un ou plusieurs composants (considérée comme des fautes pour le système global). Il est essentiel de bien cerner la nature des fautes que l'on cherche à tolérer afin de construire des mécanismes de tolérance qui contiennent la diversité appropriée pour être efficaces. De nombreuses classifications des fautes existent, chacune faisant appel à un traitement spécifique de tolérance aux fautes. Ainsi les fautes de conception ne peuvent être tolérées par une simple réplication du composant sur plusieurs machines, mais un développement de plusieurs versions de ce composant par des équipes différentes serait plus approprié.

La tolérance aux fautes vise à garantir la *fiabilité* de services et leur *disponibilité* dans un contexte réparti. Un service défaille s'il devient *non-fiable* et/ou *non-disponible*. La *fiabilité* est la capacité qu'un système distribué continue de fonctionner en dépit de fautes (i.e., présence de composants défaillants) [**?**]. Carter et al. [**?**] définissent trois facteurs dont dépend la fiabilité d'un système : i) *fiabilité des machines* sur lesquelles ses composants s'exécutent ; ii) *fiabilité* de ses composants ; et iii) *fiabilité* des liaisons réseau et du degré de congestion et de collision. La fiabilité se mesure comme la probabilité qu'un système ne défaille sur une periode d'exécution $[t_1, t_2]$ (voir Equation 1) où $MTTF$ est le temps moyen jusqu'à la prochaine défaillance (*en anglais*, Mean Time To Failure). $MTTF$ correspond à une densité de probabilité de défaillances, avec une distribution exponen-

tielle, pour décrire l'occurrence des défaillances dans le système.

$$F(t_1 < X \leq t_2) = 1 - \int_{t_1}^{t_2} MTTF(t)dt \tag{1}$$

La *disponibilité* se réfère, quant à elle à la capacité d'un système à fournir des résultats au moment où ils sont nécessaires ou utiles à l'utilisateur [**?**]. Pendant la période d'exécution, le système est dans un état soit *actif* soit *en réparation*. Réparer un système consiste à le remettre en état pour rendre un service correct). La disponibilité se mesure comme la probabilité qu'un système soit disponible (i.e., fournisse un service correct) sur une période $[t_1, t_2]$ (Equation 2 où $MTTR$ est le temps moyen de réparation (*en anglais*, Mean Time To Repair) et $MTBF$ est le temps moyen entre deux défaillances ($MTTF + MTTR$) (*en anglais*, Mean Time Between Failures).

$$A(t_1 < X \leq t_2) = \int_{t_1}^{t_2} \frac{MTTF(t)}{MTBF(t)}dt \tag{2}$$

En dépit de cette grande hétérogénéité de situations de fautes, différents aspects de tolérance aux fautes ont été abordés par deux familles de techniques : *préventive* (i.e., éviter les erreurs) et *curative* (i.e., corriger les erreurs). La *redondance* et la *diversité* sont des exemples de techniques préventives incontournables dans le domaine de la tolérance aux fautes. La seconde famille fait souvent appel à un mécanisme de *diagnostic des fautes* pour identifier le composant du système responsable de l'erreur ayant conduit à la défaillance d'un autre composant.

Dans la première famille, nous distinguons deux techniques : *redondance*, et *diversité*. D'une part, la *redondance* consiste à construire dans un système une sorte de "*capacité additionnelle*" de service, *partielle* (e.g., correction d'erreur ou vérification en ligne d'une condition de validité) , ou *totale* (e.g., déploiement sur plusieurs machines). La redondance est utilisée aussi bien pour la détection d'erreur que pour leur compensation. En effet, des erreurs sont détectables par exemple, lors de la comparaison de résultats fournis par plusieurs répliques. Aussi, l'état erroné d'un système possède une redondance suffisante pour masquer l'erreur, permettant ainsi la poursuite de son exécution. Par exemple, une des répliques reprend le fonctionnement du composant défaillant à partir du dernier *checkpoint*.

D'autre part, la *diversité* consiste à s'assurer que la "*capacité additionnelle*" de service soit indépendante du service à dupliquer par rapport aux processus de création et d'activation des fautes [**?**]. Aucune redondance ne peut protéger le composant en question de toute type de faute conduisant à des défaillances en mode commun (i.e., de manière identique et au même moment). À partir de ce constat, le concept de *diversité* devient essentiel, et est étroitement lié aux fautes à tolérer. Contrairement à la redondance pour laquelle une formalisation des protocoles est bien établie, la diversité reste un terrain fertile de recherche.

Dans la seconde famille, une fois le diagnostic établi, le mécanisme de recouvrement

se charge de reconfigurer le système ou ré-exécuter les composants responsables de l'erreur et ceux défaillants, les retirer ou les remplacer. Aussi, deux formes de recouvrement d'erreur existent :

— *Reprise* : le système revient à un état antérieur cohérent à partir duquel il reprend son fonctionnement.
— *Poursuite* : le système continue son fonctionnement souvent de manière dégradée, à partir d'un état ultérieur. Ce mode de recouvrement est très dépendant du type d'application.

Nous venons de présenter les grands principes de la tolérance aux fautes. Nous allons maintenant nous intéresser à l'utilisation de ces principes dans le cas particulier des services Web.

0.2 Tolérance aux fautes dans un environnement SOA

Le développement des applications basées service tolérantes aux fautes font souvent appel à des solutions capables de faire face à des crashs et/ou comportements anormaux des services Web. Par la suite, nous introduisons une taxonomie des fautes spécifiques à SOA ainsi que les différents types de détection d'erreur et de recouvrement associés.

0.2.1 Taxonomie des fautes dans SOA

Plusieurs taxonomies de fautes, raffinant celle de Avizienis et al. [**?**] ont été présentées pour des applications logicielles distribuées classiques (e.g., [**?**]). Le principal intérêt de ces taxonomies est de construire des modèles de fautes (certains dépendants du domaine d'application) nécessaires à la *détection d'erreur* et au *diagnostic de fautes*. L'identification des classes de fautes permet d'identifier des patterns de réaction (e.g., recouvrement) au lieu de devoir prendre des décisions au cas par cas.

Cependant, les caractéristiques des services Web diffèrent considérablement de celles des applications classiques. Les services Web sont faiblement couplés et dynamiques de nature, et interagissent de manière dynamique sans une connaissance a priori des uns des autres. Par conséquent les fautes (i.e., défaillances au niveau du composant) dans un environnement SOA peuvent être fréquentes. Ainsi, une étape clé pour un mécanisme de détection/diagnostic est d'identifier les différents types de fautes dans un tel environnement et de fournir une taxonomie claire de tous ces types.

Chan et al. [**?**] proposent une taxonomie de fautes dans une composition de services Web. La taxonomie proposée peut être utilisée pour différencier les différentes défaillances observées, et par conséquent constitue une aide pour le recouvrement. Les auteurs subdivisent les fautes en une autre dimension, en plus de celle de Avizienis et al. [**?**], avec comme catégories : *physiques*, *de développement* et *d'interaction*. La première catégorie comporte les défaillances au sein de l'infrastructure du réseau ou côté serveur.

Dans la seconde, les fautes sont introduite dans le système par son environnement (e.g., paramètre incorrect, changement d'interface). Finalement, nous retrouvons les fautes de contenu (i.e., les résultats délivrés par le service composite sont différents de ceux attendus) et de timing (i.e., les résultats sont fournis en dehors de l'intervalle prévu de temps). Les auteurs combinent les deux dimensions pour leur associer des effets observables (i.e., défaillances), correspondant aux causes des fautes.

Dans la littérature Bruing et al. [**?**] proposent une taxonomie de fautes typiques dans SOA. Les fautes étudiées incluent les fautes *matérielles*, les fautes *logicielles*, les fautes *réseaux*, et les fautes d'*interactions*. Les auteurs partent du fait que les fautes peuvent apparaître au cours des différentes étapes SOA : *publication*, *découverte*, *composition*, *binding*, et *exécution*. Si ces fautes sont actives, elles causeront des erreurs conduisant à une défaillance à moins que la structure du système soit capable de traiter ces erreurs (e.g., masquage de fautes via une redondance/diversité de service).

La taxonomie commence par les classes de fautes les plus générales, associées chacune à une étape SOA, jusqu'à celles plus spécifiques. Cette généralisation permet une plus large couverture de toutes les fautes possibles. La taxonomie doit être raffinée pour chaque domaine d'application permettant à un système de réagir face à des fautes spécifiques au domaine d'application. L'exemple d'agence de voyage est utilisé pour illustrer différentes situations de fautes selon la taxonomie proposée.

Dans notre thèse, nous nous intéressons plus particulièrement aux fautes introduites au moment d'exécution (`Execution fault`) plus particulièrement une entrée incorrecte (`incorrect input`), une faute dans le service `Service faulty`. Des fautes introduites au moment de publication telles que les fautes de description de service (`Service description fault`). Ce type de faute survient quand la description du service est incorrect (pour des raisons volontaires par exemple un fournisseur désire donner une meilleur réputation à son service pour qu'il soit invoqué ou involontaires, par exemple une description incompatible du service une date peut être donnée par DDMMYY ou YYMMDD). La dernière classe auquel nous nous intéressons est les fautes introduites au moment de liaison tel que les fautes de liaison à un mauvais service (`Bound to wrong Service`). L'activation de ces fautes peut conduire à des erreurs qui causent une défaillance de service de type Byzantine (résultat incorrect `Incorrect Result`).

Nous nous intéressons aussi aux défaillances par crash qui peuvent être dues au crash du service, crash du serveur ou une faute de communication (réseau).

Cette classification est importante dans le sens où elle fournit une connaissance pour déployer des services Web fiables. Cependant, il manque une implémentation de cette taxonomie pour détecter les erreurs dans le SOA.

Cheun et al. [**?**] définissent une taxonomie de causes en considérant les caractéristiques du paradigme service (e.g., entités faiblement couplées [**?**]), les standards SOA (e.g., SOAP [**?**], WS-BPEL [**?**]), et les éléments SOA (e.g., Enterprise Service Bus [**?**], moteur BPEL [**?**]). Ils identifient trois types de causes et leurs sous-types : : *causes imbriquées* (e.g.,

signature d'opération avec des types de données non-définis), *causes d'interaction* (e.g., inconsistence et/ou incompatibilité dans le binding entre deux composants), et *causes exécutives* (e.g., paramètre ou ressource requise au moment de l'exécution). Ils justifient cette classification par des différences lors des traitements de ces causes (i.e., détection d'erreur, diagnostic de fautes, et recouvrement/adaptation).

Pour le premier type, seul le composant conduisant à la cause est sujet au recouvrement ou à un changement. Pour le second type, le recouvrement/adaptation est appliqué à des composants devant interagir entre eux mais avérés incompatibles. Pour remédier au troisième type, un processus plus élaboré est nécessaire pour déterminer le composant responsable de la cause. Contrairement à Chan et al. [**?**], les auteurs proposent une formalisation de cette taxonomie ainsi qu'un prototype pour gérer les fautes de service.

0.2.2 Mécanismes de tolérance aux fautes dans SOA

Dans cette section, nous nous intéressons aux mécanismes nécessaires à la mise en œuvre de stratégies de tolérance aux fautes : *détection d'erreur* et *recouvrement/compensation d'erreur*.

0.2.2.1 Détection d'erreur

Dans [**?**], Yan et al. suggèrent de réutiliser la masse de connaissance sur la détection d'erreur dans les systèmes industriels à évènements discrets. Ils présentent une discussion sur la détection d'erreur dans les orchestrations de services Web au moment de l'exécution. Les auteurs supposent que les défaillances ne sont pas observables. La granularité dans le modèle d'exécution est au niveau processus.

Dans [**?**], Li et al. proposent une approche de détection d'erreur au moment de l'exécution. Les erreurs considérées proviennent des services Web défaillants et des données potentiellement corrompues dans des orchestrations de processus WS-BPEL. L'idée principale est de considérer ces processus comme des systèmes à évènements discrets et les faire correspondre à des réseaux de Petri colorés.

Dans [**?**], Borrego et al. proposent un framework de détection d'erreur pour des processus métier décrits avec *BPMN*, contenant une couche de diagnostic. Les auteurs considèrent des processus composés de plusieurs activités susceptibles de renvoyer des résultats incorrects, dus à des données en entrée incorrectes. La fonctionnalité correcte de chaque activité est spécifiée par un sensemble de règles de conformance. Ces règles sont transformées en des problèmes de satisfaction de contraintes. Les erreurs sont alors détectées au moyen d'un résolveur de contraintes.

Dans [**?**], Frantz et al. proposent une solution pour détecter les erreurs dans les solutions EAI (acronyme de *Enterprise Application Integration*), type de workflows exogènes dans lesquels les messages provenant d'un sous-ensemble d'applications sont acheminés vers des processus pour éventuellement les transformer et transmettre les résultats à

un autre sous-ensemble d'applications. Ces solutions EAI peuvent être spécifiées comme une orchestration (i.e., processus central gérant les échanges de messages) ou bien une chorégraphie (i.e., échanges de messages paire à paire entre les processus et les applications). Dans une orchestration, le processus centrale a une vue globale précise de l'état d'exécution de la solution EAI. Il peut être ainsi utilisé pour identifier les processus et applications impliqués dans une erreur. L'approche de monitoring proposée s'appuie sur des méta-informations sur les processus et ports de communication impliquées. Elle contient deux sous-systèmes (*Gestionnaire d'évènement* et *Détecteur d'erreur*) et deux bases de données (*Graphe* et *File de labeur*). Le *Gestionnaire d'évènement* capture les évènements de lecture/écriture des messages en termes de succès et échec. Il construit à partir de ces évènements la structure du *Graphe de labeur* stockant l'échange des messages entre les processus et les relations parent-fils. Le *Détecteur d'erreur* analyse ce graphe pour trouver les corrélations dans lesquelles un message spécifique est impliqué et les vérifier. La *File de labeur* sert de buffer entre le *Gestionnaire d'évènement* et le *Détecteur d'erreur* pour leur permettre de travailler de manière asynchrone. Pour vérifier les corrélations, le *Détecteur d'erreur* se base sur deux types de règles à savoir les règles `Built-In` et `User-Defined` en rapport avec les erreurs de communication ou de deadline, et les erreurs structurelles dépendant de la sémantique du processus ou de la solution EAI envisagée (i.e., corrélations pour lesquelles des messages sont manquants ou plus que prévu). Une partie de la vérification d'une corrélation (sub) s'appuie sur un ensemble de règles `User-Defined` (Les règles définies par les utilisateurs, elles sont utilisées pour exprimer des relations de fonction et des propriétés, qui ne sont pas explicitement disponible dans les spécifications des documents de conception) et `Built-In` (représentent les règles prédéfinies dans le système, vérification de division par Zéro par exemple) .

Contrairement aux autres approches, cette solution est indépendante du modèle d'exécution (processus ou tâche) et de la forme de la composition (orchestration ou chorégraphie).

0.2.2.2 Recouvrement et compensation d'erreur

Depuis une décennie, les mécanismes pour rendre les compositions de services Web tolérantes aux fautes ont suscité un vif intérêt dans la communauté de recherche. Les approches existantes se sont principalement intéressées au recouvrement par reprise, et plus spécifiquement aux transactions. Cependant, l'autonomie des services Web et la latence du Web ont influencé considérablement l'orientation de recherche exigeant des modèles transactionnels et techniques de recouvrement par reprise plus flexibles et complexes.

Recouvrement par reprise. Le modèle transactionnel classique a prouvé son succès dans l'application de la SdF aux systèmes distribués fermés. Il est beaucoup exploité pour implémenter des services Web primitifs (i.e., non-composites). Cependant, il s'avère inapproprié pour rendre les compositions de services Web

tolérantes aux fautes pour deux raisons : **(i)** La gestion des transactions (réparties sur l'ensemble des services Web) demande une coopération parmi les supports transactionnels des services Web individuels. **(ii)** Le verrouillage des ressources jusqu'à la terminaison de la transaction imbriquée est en général inappropriée pour les services Web (clients avec des délais d'attente très restreints). Une solution pour ce problème est les modèles transactionnels améliorés, faisant référence à des transactions imbriquées ouvertes (décomposition de la transaction en sous-transactions pouvant s'engager indépendamment. En cas d'échec de la transaction, la vérification de la propriété de l'atomicité ($ACID$), demande une compensation des sous-transactions engagées dans cette transaction. Cependant, les services Web doivent fournir des opérations de compensation pour toutes les opérations qu'ils fournissent (langages BPEL et WSCI). Il est clair qu'une cascade de compensation est à prévoir lors d'une cascade d'annulation (e.g., [**?**]). En plus des solutions côté-client à la coordination des transactions distribuées imbriquées, des auteurs comme Hass et al. [**?**] (WS-Transaction) proposent des protocoles transactionnels distribués supportant le déploiement des transactions sur le Web sans imposer des verrous trop longs sur les ressources Web.

Recouvrement par poursuite. Le mécanisme de gestion d'exception est largement utilisé pour la mise en œuvre de ce type de recouvrement dans les compositions de services Web. Dans BPEL, les activités peuvent être dotées de Handlers d'Exception qui assure la continuation de l'exécution en cas d'erreur. Cependant, quand une activité au sein d'un processus concurrent signale une exception, toutes les autre activités intégrées terminent, et seulement le handler correspond est exécuté. Par conséquent, le recouvrement d'erreur prend en compte réellement qu'une seule exception et n'est pas en mesure d'assurer un recouvrement à un état correct. Le seul cas possible est lorsque l'effet de toutes les activités en échec est défait (Pas possible dans un contexte de services Web). Une solution à ce problème est de structurer la composition de services Web en termes d'actions atomiques coordonnées (en anglais, Coordinated Atomic (CA) Actions). Dans [**?**], les actions atomiques sont utilisées pour contrôler la concurrence coopérative et implémenter un recouvrement coordonné d'erreur sont utilisées pour contrôler la concurrence coopérative et implémenter un recouvrement coordonné d'erreur. Deux cas de figure se présentent : i) les participants atteignent la fin de l'action et produisent un résultat normal ; ii) en présence d'exception(s), ils sont tous impliqués dans leur handling coordonnée. Si la gestion d'exception réussit, l'action finit avec succès. Mais si le handling n'est pas possible, alors la responsabilité pour le recouvrement est transmise à l'action contenante où une exception d'action externe est propagée. Les CA actions fournissent un mécanisme pour développer des services Web composite tolérants aux fautes : une CA action spécifie la réalisation collaborative d'une fonction donnée par des services composés, et les

services Web correspondent à des ressources externes. Cependant, comme pour les transactions, les propriétés ACID sur des ressources externes ne sont pas appropriées dans le cas de services Web. Tartanoglu et al. [**?**] Introduisent la notion de Web Service Composition Action (WSCA) qui relaxe les besoins transactionnels sur les ressources externe.

Recouvrement par réplication La réplication de services Web services avec état peut être réalisée de plusieurs manières : (i) active (ou machine à états) [LCR06] ; (ii) passive (ou copie primaire) [MSR77] ; iii) semi-active (i.e., combinaison des deux). La réplication active est basée sur l'idée d'envoyer la requête à tous les réplicas et attendre toutes les réponses. Ce type nécessite l'ordonnancement de la requête, et la suppression des invocations imbriquées. Elle garantit une synchronisation des états, cependant, elle suppose que toutes les opérations produisent des résultats et transitions d'état déterministes. Cette limitation peut être contournée en déclarant explicitement toutes les fonctions non-déterministes et redirigeant les invocations vers un unique service. Ce dernier est responsable de la diffusion des changements de l'état interne à tous les autres réplicas. Cette approche, appelée semi-active est une composition des idées des deux approches machine à états et copie primaire. Dans la technique copie primaire, toutes les requêtes sont envoyées à un seul service appelé primaire. Ce service met à jour automatiquement tous les réplicas secondaires. Quand il défaille, une réplique secondaire est sélectionnée comme nouveau primaire. En vertu de la possibilité de réseaux ad-hoc de se diviser et fusionner, il devient évident que les approches actives et semi-active sont inappropriées (synchronisation très coûteuse pour maintenir tous les réplicas dans le même état interne, besoin de maintenir le trafic du réseau aussi bas que possible). Dans [**?**] les auteurs utilise la technique copie primaire pour réaliser leur système de réplication. La synchronisation des répliques s'effectue via un protocole (Simple Replicator Protocol (SRP)). Un réplicateur de services Web est conçu comme une collection de modules coopérant via une base de données pour stocker toutes les données pertinentes, pour structurer la tâche de réplication. Les modules consomment peu de mémoire et font partie de toute instance du réplicateur. Le résultat est une communauté pair-à-pair de nœuds dans laquelle les élections décide de la distribution des tâches nécessaires, de préférence sur les nœuds les plus puissants, tandis que d'autres restent oisifs.

Dans cette thèse, nous nous intéressons à la compensation par diversité faisant l'objet de recherches actives dont nous analysons un échantillon de travaux en Section 0.3.

Compensation par diversité. Traditionnellement, la recherche en tolérance aux fautes logicielle s'articule autour de deux approches : N-versioning (en anglais, N-Version programming (NVP)) de Avizienis [**?**] et blocs de recouvrement (en anglais, Recovery Blocks (RB)) de Randell [**?**]. Dans la première approche, N-versions d'un programme exécutent de multiples implémentations d'une même fonctionnalité

mais ayant des conceptions diverses en parallèle (i.e., différentes équipes de développement, différents outils). Un vote majoritaire est effectué pour obtenir les résultats final à retourner à l'utilisateur (ou application appelante). Dans la seconde approche, les RB's invoquent des implémentations redondantes de manière séquentielle si la sortie d'un RB ne réussit pas le test de conformité/validité. La technique de NVP semble être une bonne candidate aussi bien à des objectifs de SdF que de performance, malgré le coût de développement de version multiples. Dans le domaine des services Web, Looker et Munro [**?**] ont été les premiers à introduit NVP. Sommerville [**?**] a suivi avec son approche de tolérance aux fautes à base de conteneur configurable avec une politique spécifiant le type de mécanisme de tolérance aux fautes appliqué aux services le contenant. Dobson [**?**]a utilisé quant à lui les mécanismes fournis par WS-BPEL pour supporter NVP et RB.

0.3 Overview des approches existantes de tolérant aux fautes dans les services Web

Dans cette section, nous allons nous focaliser sur les études s'intéressant à différents aspects problématiques de la technique de compensation sous ses deux formes : réplication et diversité de services Web. Durant la dernière décade, un grand nombre de travaux ont mis l'accent sur la première forme, laissant ainsi un champ fertile de recherche à explorer pour l'application du concept de diversité à la conception de services Web tolérants aux fautes. Rappelons que l'un des avantages de la diversité est d'éviter que les services Web sémantiquement équivalents (i.e., versions de service conçues/développées indépendamment par différents fournisseurs) ne défaillent pas de la même manière et au même moment contrairement à une réplication classique (i.e., minimisation considérable de la probabilité d'occurrence de fautes communes parmi les différentes versions).

0.3.1 Approches FT basées sur la réplication

0.3.1.1 ReplicationManager

Chan et al. [**?**] présentent une gestion dynamique de la réplication afin d'améliorer la sûreté de fonctionnement des services Web. Fig. 4 montre l'architecture proposée pour tolérer les défaillances *franches*.

Elle se base sur la technique de réplication des Services Web en mode passif. Le composant majeur de cette architecture est le Gestionnaire de réplication, agissant comme le coordinateur des services Web. Il est responsable de la création des services Web, de choix du service Web primaire (plus rapide, plus robuste..etc) utilisant un algorithme

de sélection, de l'enregistrement du *WSDL* et *UDDI*, de la vérification de la disponibilité du service Web primaire par la technique du *Watchdog* et enfin de la sélection de nouveau primaire si le service primaire échoue.

Les services Web sont répliqués et déployés sur différentes machines, mais seul un service Web, appelé primaire fournit la réponse à une requête. Le gestionnaire de réplication (en anglais, ReplicationManager (`GR`)), qui représente la partie centrale de cette architecture, sélectionne le premier service primaire à invoquer. Si le service primaire tombe en panne, le GR utilise un algorithme appelé *Anycasting algorithme* pour sélectionner un nouveau primaire.

`GR` se charge alors de mettre à jour le `WSDL` par l'adresse du nouveau service Web primaire. Ainsi, les clients peuvent toujours accéder au service Web avec la même `URL` et ce de manière transparente en dépit des défaillances.

Chan et al. considèrent leur approche comme dynamique, dans la mesure où une réplique s'engage à la fois pour être exécuter, tendis que les autres répliques se trouvent dans un état en veille. Si le service primaire échoue, une autre réplique peut être utilisée immédiatement avec peu d'impact sur le temps de réponse

Cette approche présente plusieurs avantages à savoir le basculement transparent vers un autre service dans le cas d'échec de premier, des expérimentations qui montrent comment la redondance améliore la disponibilité des services Web. Cependant, cette approche n'est pas applicable pour le cas des fautes Byzantines. Pour cela, elle devrait posséder une spécification sur la valeur de résultat à retourner.

0.3.1.2 ServiceGlobe

Keidl et al. [**?**] proposent une architecture flexible, nommé `ServiceGlobe` pour supporter une exécution fiable des services pouvant être intégrée (ou plug-in) dans les plates-formes existantes de services. Le `dispatcher` représente le cœur de cette architecture, il fourni des services de réplication automatique. Ces services peuvent être interne ou externe, les services internes sont spécifiques et natifs pour l'architecture `ServiceGlobe`, et les services externes sont les services actuellement déployés sur Internet, qui ne sont pas fournis par `ServiceGlobe` lui-même. Une réplication automatique schématisée dans la Fig. 5 permet d'assurer une haute-disponibilité des services Web. La réplication automatique consiste à basculer d'une manière automatique les instances exécutées sur des machines avec une grande charge du travail vers des machines moins chargées. Cette technique a des avantages : Premièrement, le développement des services sans avoir à considérer la haute disponibilité et l'équilibrage de charge est beaucoup plus facile. Deuxièmement, chaque service existant peut être transformé facilement en un service disponible, comme il n'y a pas de partage de données entre les différentes instances de ce service ou il a une sorte de contrôle de concurrence, par exemple l'utilisation d'une base de données comme un back-end.

En effet, `ServiceGlobe` permet de faire basculer des instances d'exécution de ser-

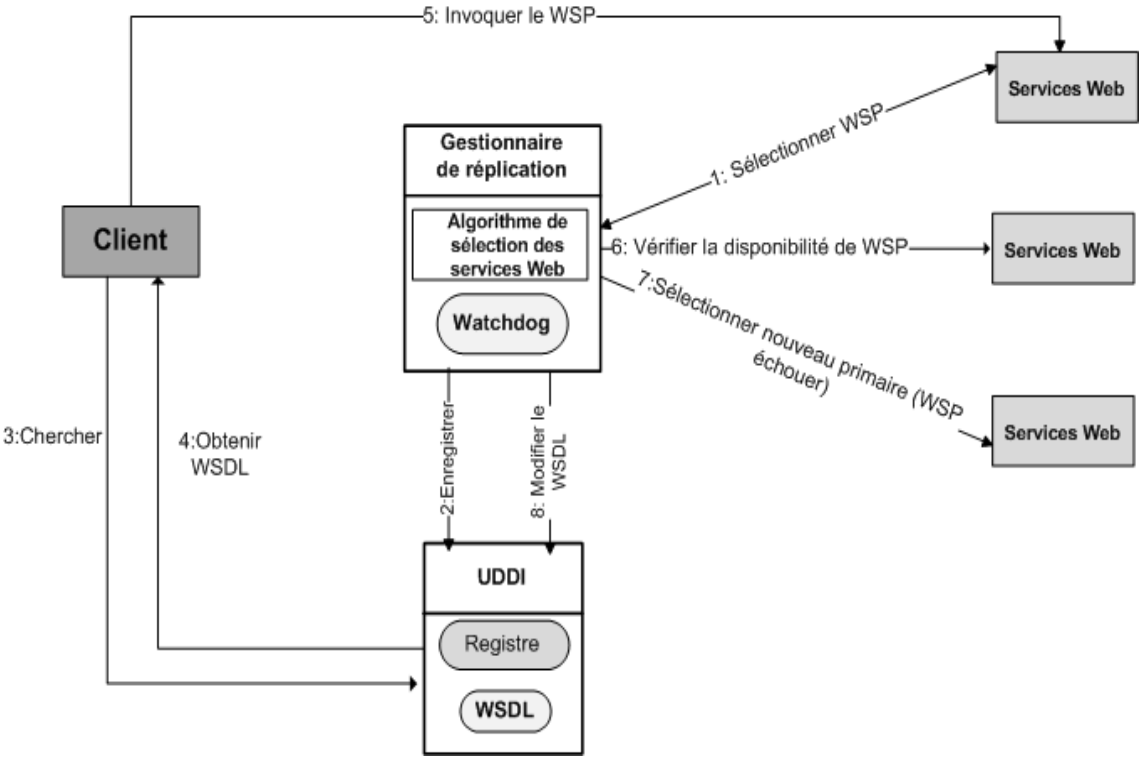

FIGURE 4 – **Architecture proposée pour la sûreté de fonctionnement dans les services Web**

vices Web sur des machines avec une grande charge de travail vers d'autres moins chargées afin de respecter un certain équilibre de charge. Ce basculement est effectué de manière transparente.

Les auteurs présentent aussi une technique de sélection dynamique des services en fonction de `TModel`. Dans l'UDDI, chaque service est affecté à un `TModel` fournir un classement de sa fonctionnalité et une description formelle de ses interfaces. Par conséquent, un service peut être appelé une mise en œuvre ou une instance de son `TModel`. Dans la sélection dynamique des services, une application n'a plus besoin de connaitre les points d'accès codés en dur pour accéder au service, mais il suffit d'appeler le `TModel` associé à ce service. Cette tâche est effectuée par le `Negociator`, Ajoute à cela l'application doit connaitre la fonctionnalité à invoquer plutôt que la mise en œuvre effective du service. Elle a pour résultat une spécification technique des services devant être invoquées. Cette spécification consiste à fournir une couche d'abstraction du service Web réel, ainsi que les contraintes qui permettre au service Web d'influencer la sélection dynamique [**?**].

L'appel de `TModel` dans les différents modes (i.e., `Unicast`, `Multicast`, `Broadcast`) correspond aux différents modes d'invocation classiques (i.e., actif et passif). Dans le mode actif deux variantes sont proposées : i) `Multicast` où un sous-ensemble de services est activé, et ii) `Broadcast` où l'ensemble des services est invoqué.

Pour mettre en œuvre la technique de réplication automatique et celle de sélec-

tion dynamique, les auteurs développent un composant, appelé `dispatcher` agissant comme un *proxy* pour les services. Ce `dispatcher` se charge de surveiller un ensemble d'hôtes sur lesquelles les mêmes instances des services s'exécutent pour équilibrer la charge de travail sur chaque hôte. Si aucun hôte n'est disponible, le `dispatcher` peut enregistrer les messages entrants dans un *buffer*, ou bien les rejeter en envoyant une réponse "**temporairement indisponible**" au demandeur du service. Ce choix est fait en fonction de la configuration du `dispatcher`.

Cette approche basée sur l'invocation des instances de service identiques améliore considérablement la disponibilité de services. Cependant, elle se limite à des fautes liées au serveur et non pas au service lui-même.

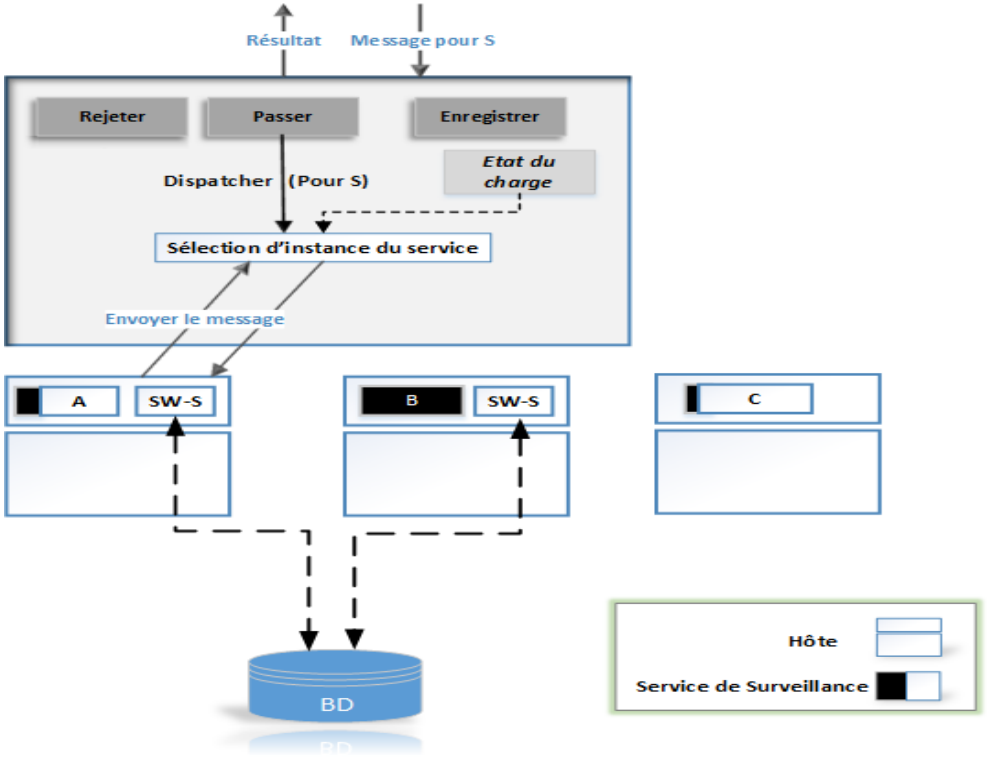

FIGURE 5 – **Architecture de dispatcher**

0.3.2 Approches basées sur la diversité

0.3.2.1 Connecteurs

Salatge et Fabre [?] proposent une infrastructure nommée IWSD (acronyme de "*Infrastructure for Web Services Dependability*") pour tolérer les défaillances *franches* et *en valeur* de services Web. Elle se base sur des connecteurs SFTC (acronyme de "*Specific Fault Tolerance Connectors*") assurant une communication fiable entre les clients et les prestataires de service (voir Fig. 6). Cette architecture est composée des éléments suivants : Le connecteur qui 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 qui peut être considéré comme une machine virtuelle sur laquelle s'exécutent les connecteurs. Il est composé de deux modules distincts : (i) Le serveur d'exécution qui intercepte les requêtes des clients et charge le connecteur souhaité à partir de l'analyse du message reçu, et (ii) Le moniteur de surveillance qui est en charge de vérifier l'état de la plate-forme IWSD et d'évaluer l'état courant des Services Web en exécution. Le dernier composant est 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. IWSD contrôle et exécute les connecteurs dans des applications critiques. cette tâche du contrôle est associé au serveur d'exécution qui fournit trois services : **le Service d'exécution** qui est le responsable de la vérification de la syntaxe des messages (Une exception SOAP est automatiquement générée dans le cas d'une réception d'un message mal-formé), de l'authentification et de la récupération de toutes les informations relatives au connecteur désiré. Le deuxième composant, **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. Le troisième composant est **le Chien de garde** 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.

Ces connecteurs implémentent plusieurs variantes des stratégies passive et active, utilisant aussi bien des répliques identiques que sémantiquement équivalentes.

Pour la spécification et la mise en œuvre des SFTC, les auteurs introduisent un langage spécifique appelé DeWel. Ce langage offre les abstractions et les notations appropriées capables de réaliser des actions de tolérance aux fautes efficaces par des mécanismes de recouvrement ou de signalement d'erreur. Des assertions en termes d'exceptions SOAP peuvent être déclarées dans le connecteur. Lorsque ces dernières ne sont pas satisfaites, des mécanismes de gestion des erreurs (e.g., collecter des informations sur les erreurs et politiques de recouvrement) peuvent être activées pour traiter les exceptions au sein du service et celles se produisant lors de la communication entre clients et prestataires de service.

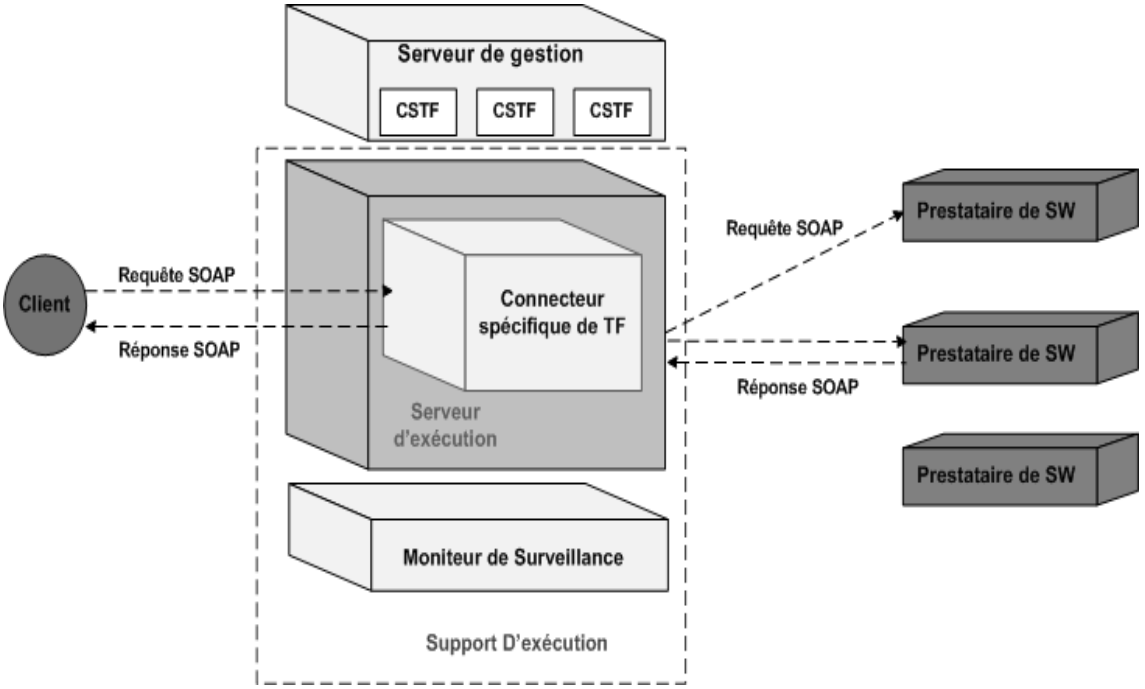

FIGURE 6 – **IWSD, une infrastructure pour la sûreté de fonctionnement des Services Web**

0.3.2.2 Containers

Dobson et al. [**?**] proposent une approche basée sur les `Containers` de tolérance aux fautes dans les architectures `SOA` (voir Fig. 7). Les auteurs considèrent plusieurs exemples de fautes tels que l'insuffisance de ressources, des fautes dans la mise en œuvre des services et la perturbation du réseau. Ils utilisent la technique de diversité des services composant les applications `SOA`. Un `Container` joue le rôle d'un intermédiaire entre le client et les fournisseurs des services Web.

Le `Container` agit comme un *proxy* pour les services réels. Par conséquent, un message en route vers un service déployé sera intercepté par le `Container`, ce qui ajoute la caractéristique de la tolérance aux fautes au service. Cette action est transparente par rapport au client et fournisseurs du service [**?**]. Le `Container` est configuré avec une politique précisant le type de mécanisme de tolérance aux fautes à appliquer aux services le contenant (ré-exécution du service (retry), Recovery-Bolck [**?**], redondance).

Le fonctionnement d'un `Container` ressemble à celui d'un serveur d'application *EJB*, il déploie des objets *proxy*, sous la forme d'un service transparent. Ces services de *proxy* interceptent les messages envoyés à des services réels. Cette interception est réalisée à travers le déplacement du *endpoint* (il représente l'adresse de service habituellement représenté sous la forme d'un URI), le *endpoint* du service réel est remplacé par le *endpoint* du service de *proxy*. Le client doit connaitre le *endpoint* du service de *proxy*. Contrairement aux composants *EJB*, les services "contenus" ne doivent pas résider à l'intérieur du `Container`.

La diversité des services est fournie en mettant en liaison les services d'un `Container` et le marché des services. De cette façon, le recouvrement peut être réalisé à un coût faible. Toutefois, la solution proposée n'est pas encore assez mûre pour une mise en application réelle. Par exemple, la notion de similarité entre les services Web est très succinctement abordée. En outre, les mécanismes de recouvrement proposés ne sont pas appropriés aux services avec état.

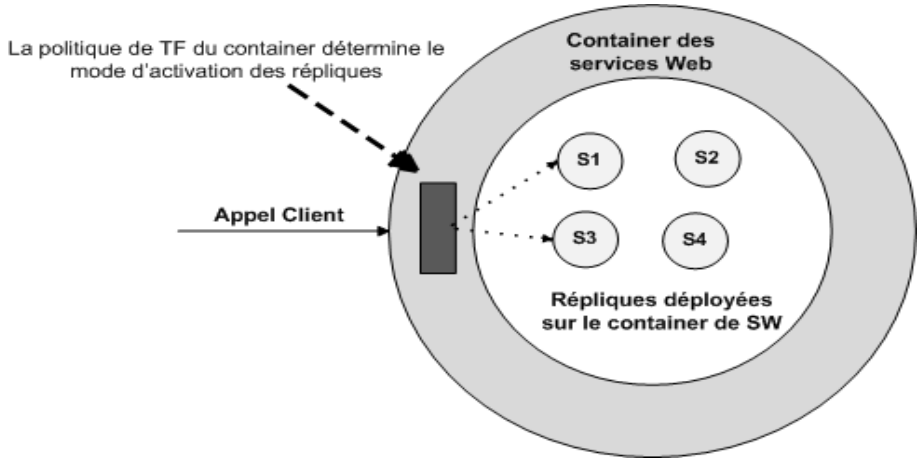

FIGURE 7 – **Le principe du container des Services Web**

0.3.2.3 Composition horizontale et verticale

Gorbenko et al. [**?**] présentent une analyse de modèles de compositions fiables de services Web selon deux perspectives : *verticale* en termes de fonctionnalités et *horizontale* en termes de sûreté de fonctionnement (voir Fig. 8). Selon la perspective *verticale*, les auteurs utilisent la réplication et la diversité des services Web reliés par des patterns de composition. Les différents patterns sont : (i) `Reliable concurrent execution`; Toutes les répliques (versions) sont invoquées concurremment, et leurs réponses sont adjugées par le voteur. (ii) `Fast concurrent execution`; Tous les services disponibles sont invoquées simultanément, mais le résultat du premier service correcte est retourné, cette approche améliore le temps de réponse (iii) `Adaptive concurrent execution`; Tous (ou certaines) répliques (ou versions) sont exécutées simultanément. Le middleware est configuré pour attendre un certain nombre de réponses, mais pas plus d'un délai prédéfini.(vi)`Sequential execution`; la réplique (version) d'un service Web n'est invoquée sauf si la réponse reçue par le service primaire est incorrecte. Selon la perspective horizontale, l'objectif est d'améliorer la tolérance aux fautes particulièrement la disponibilité des services Web. Cette amélioration est réalisée grâce aux différents patterns utiliés, ainsi que l'utilisation des services Web similaires. L'architecture avec la composition horizontale comprend un composant appelé médiateur pour le vote parmi les réponses de tous les services Web divers et renvoie une réponse adjugée au client. Les auteurs définissent plusieurs objectifs de SdF à atteindre telles que la

disponibilité, l'exactitude et la qualité de la réponse.

Ils proposent des modèles sous forme de protocoles de SdF pour des compositions fiables, ainsi que différentes stratégies pour invoquer des services identiques ou similaires en mode séquentiel ou parallèle, avec des ébauches de procédure non formelles pour le vote des réponses. Cependant, des patterns supplémentaires de composition doivent être mis au point pour implémenter ces protocoles. Aussi, de nouvelles activités de contrôle permettant au processus métier de supporter la diversité et effectuer un vote doivent également être envisagées.

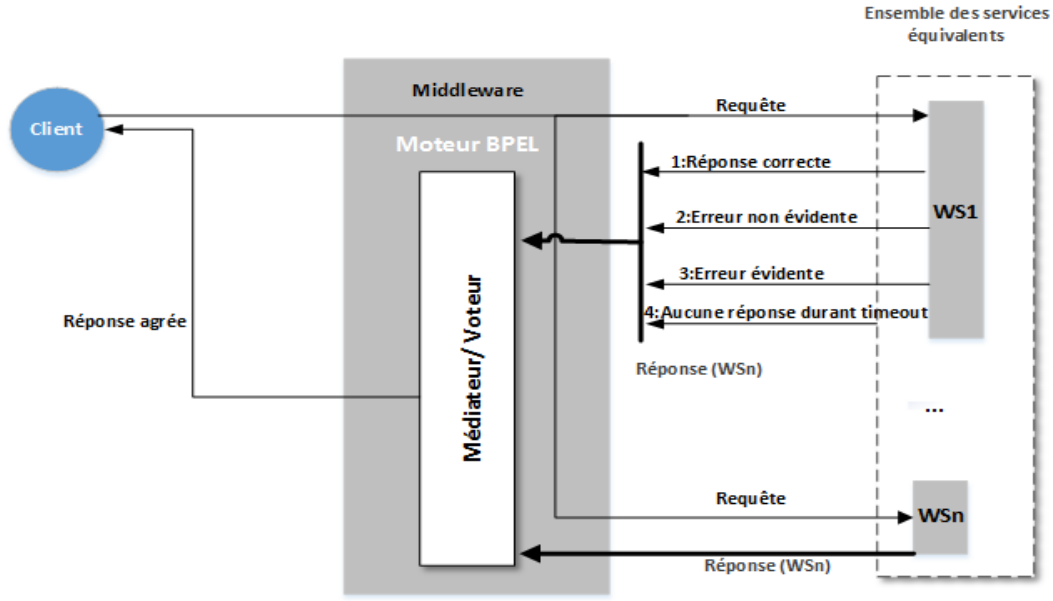

FIGURE 8 – **Architecture de composition verticale et horizontale**

0.3.2.4 WS-Mediator

Li and al. [**?**] proposent une architecture basée sur le concept de `WS-Mediator` pour supporter la résilience explicite et l'intégration dynamique des services Web. L'expression `résilience explicite` désigne l'utilisation explicite de l'information (métadonnées) sur les caractéristiques des composants d'un système pour guider la prise de décision au moment de la conception ou de l'exécution. Dans les SOA le terme adresse spécifiquement des questions de sûreté de fonctionnement pour garantir des applications hautement disponible [**?**].

Le `WS-Mediator` consiste à un ensemble du sous-médiateur (`Sub-Mediator` en anglais) interconnectés entre eux formant une architecture de recouvrement (voir Fig. 9). Les `Sub-Mediators` sont globalement répartis sur l'Internet pour surveiller la fiabilité des services Web. Ils présentent les caractéristiques de sûreté de fonctionnement des services Web d'un point de vue client. Ils peuvent être implémentés de manière identique ou différente. Le client invoque le `Sub-Mediator` comme le point d'entrée

à l'architecture du WS-Mediator. Le Sub-Mediator intercepte l'interaction entre le client et le service composant et effectue le calcul de la résilience explicite et il applique une technique de tolérance aux fautes pour améliorer la fiabilité de la composition des services.

Les auteurs utilisent les techniques de diversité et de réplication des services Web. Le WS-Mediator est une solution déployée sur une infrastructure distribuée entre un ensemble de clients et de services Web. Le mécanisme de recouvrement se compose de plusieurs sub-Mediators fonctionnellement identiques et géographiquement répartis.

Dans le WS-Mediator, les sub-Mediators interceptent les appels des clients ou d'autres sub-Mediators. Chacun des sub-Mediators stocke les informations sur les services Web et les méta-données (l'adresse endpoint du service, les méthodes de liaison de message requis). Les méta-données représentent les caractéristiques de sûreté de fonctionnement des services Web, telles que le temps de réponse moyen, les principaux types de défaillances représentant leur comportement dans une base de données.

Les sub-Mediators surveillent les services Web à partir de différents localisations géographiques. Le WS-Mediator surveille les différents services Web avec leurs méta-données de fiabilité collectées et analysées par les différents sub-Mediators. Ces données contiennent le taux de disponibilité, le temps moyen de réponse, nombre du test, le nombre du test terminé avec succès. Elles sont utilisées pour sélectionner les services Web ; les services Web sont classés selon les valeurs de leurs méta-données.

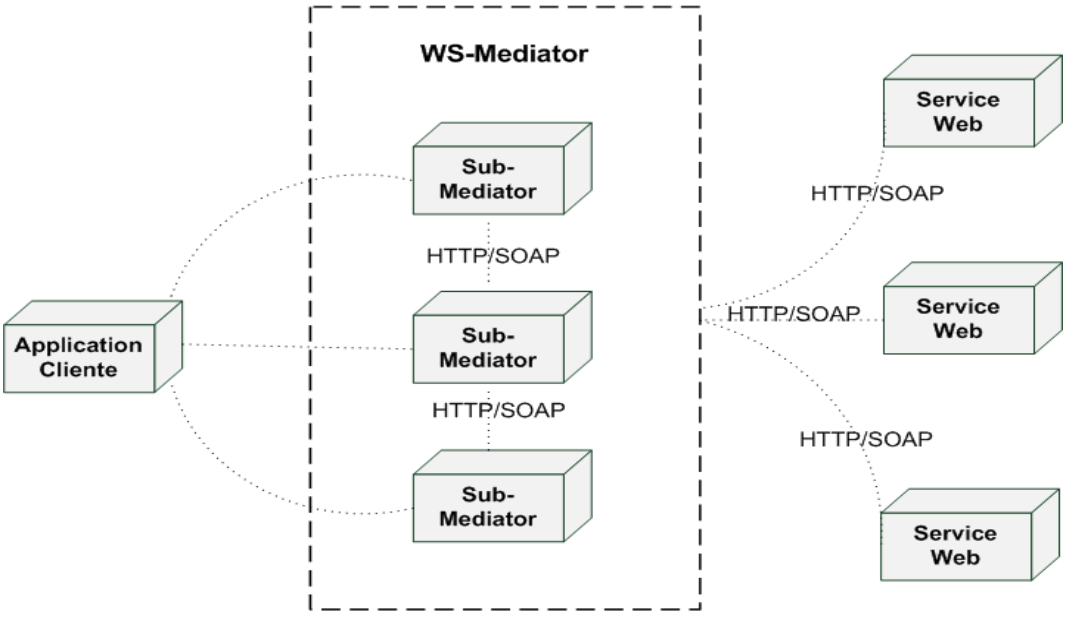

FIGURE 9 – **Architecture de Web service-Mediator**

0.3.3 Approche adaptative de tolérance aux fautes

Dans la littérature, trois stratégies sont généralement proposées : i) dynamique, c.-à-d. sensible à la *QoS* ; ii) adaptative, c.-à-d. sensible au contexte ; et iii) combinaison

de deux. D'après [**?**] la notion d'adaptabilité utilise le contexte pour opérer des changements structurelles et comportementales dans les applications tenant en compte différents environnements, différentes vues sur les données, ou encore diverses circonstances extérieures. Par la suite nous présentons plusieurs travaux de recherche se basant sur cette notion dans le domaine de la tolérance aux fautes.

0.3.3.1 Kalimucho

Louberry et al. [**?**] s'appuient sur des applications à base de composants et de connecteurs. Chaque composant et le connecteur associé sont encapsulés dans un conteneur pour observer le contexte (e.g., QoS) et contrôler le cycle de vie de composant/connecteur (e.g., démarrage ou arrêt, migration). Les auteurs proposent une plate-forme distribuée, nommée Kalimucho. Cette plate-forme peut déclencher suite à des changements de contexte des modifications de structure et de déploiement de l'application via cinq services de base (ajout, suppression, connexion à un dispositif, déconnexion et migration). Kalimucho permet : (1) une capture du contexte (e.g., événements provenant de l'application) ; (2) une proposition de redéploiement (ou reconfiguration) fiable (i.e., avoir connaissance de tous les composants logiciels et des dispositifs disponibles et teste si le déploiement respecte les exigences QoS) ; (3) une gestion de l'hétérogénéité entre les dispositifs où seront déployés les composants de l'application. Kalimucho peut déployer ainsi une nouvelle configuration d'un service en évaluant un ensemble de configurations possibles respectant l'utilité et trouver un déploiement demandant le moindre changement dans le temps. Pour trouver un tel déploiement, il met en œuvre une heuristique pour le déploiement contextuel.

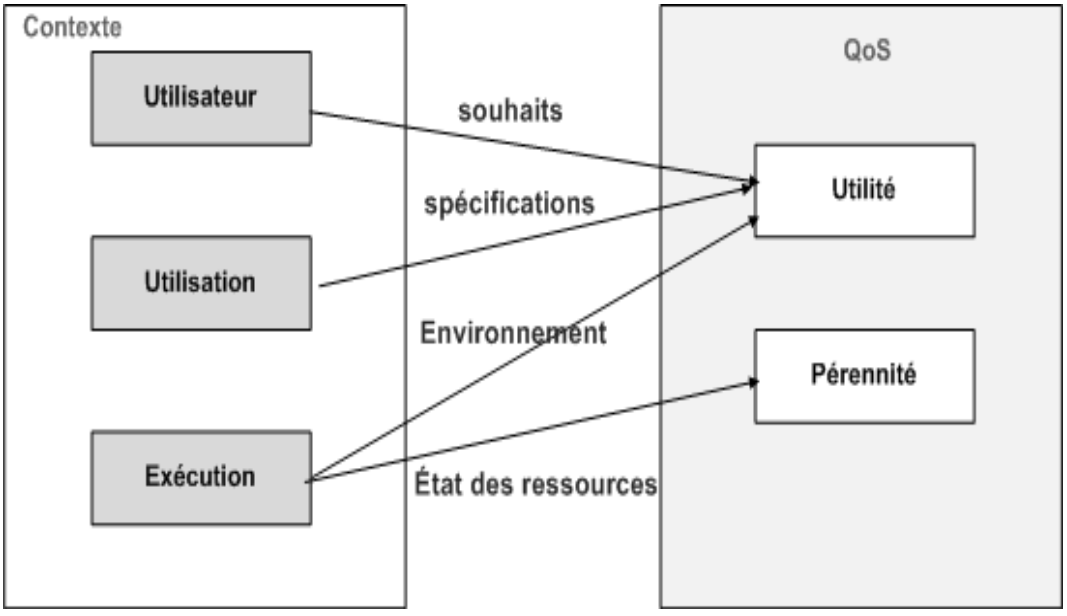

FIGURE 10 – **Interaction entre le contexte et la QoS**

0.3.3.2 Modèle de réservation de services Web

Le travail présenté dans [?] met l'accent sur la conception des compositions de services Web robustes. Dans cette approche les auteurs proposent un mécanisme de réservation de services Web provenant de différents fournisseurs.

L'algorithme proposé permet aux consommateurs de services d'exécuter des processus d'affaires de manière fiable en présence de fortes contraintes de temps. Les services les plus critiques leur sont appliqués un mécanisme de redondance pour maximiser le profit obtenu lors de leur terminaison. Cet algorithme réserve les services pour une partie du processus d'affaire, afin de conserver une certaine souplesse en cas de défaillance 11.

Les auteurs utilisent différentes stratégies d'approvisionnement (parallèle, séquentielle, et hybride) pour réserver d'une manière automatique un nombre défini de services Web pour chaque tâche. Ce nombre est d'abord défini par l'agent du système. Par contre, les services Web sont choisis de manière aléatoire à partir d'une liste de services capables d'accomplir une tâche donnée.

L'approvisionnement parallèle des services Web pour une tâche particulière permet au consommateur d'augmenter la probabilité globale du succès, ainsi que de diminuer la durée d'exécution de la tâche. Toutefois, l'introduction d'une telle redondance conduit aussi à un coût global plus élevé dans la mesure où tous les services Web doivent être payés.

Les auteurs prennent en considération plusieurs facteurs importants pour décider du nombre de services à réserver. Un grand degré de redondance est suggéré lorsque les services sont peu fiables. Ils équilibrent le coût de la redondance avec ses avantages. Quand les services sont chers, le degré de redondance sera ajusté en prenant moins de services. Cependant, quand les services ne sont pas chers, un plus grand nombre est proposé pour la stratégie parallèle. Les auteurs prennent aussi en compte l'importance de l'application pour décider combien dépenser pour les services et la redondance appropriée. Enfin, la structure de flux de travail est un autre facteur important. Par exemple, à la fin de la transaction les auteurs s'appuient sur des degrés plus élevés de redondance, afin de s'assurer que le grand investissement dans les services antérieurs ne soit pas perdu. Les auteurs introduisent aussi des modalités de pénalités pour les fournisseurs dans le cas où ils ne respectent pas leurs contrats de réservation.

Cette approche est importante dans le sens où elle adapte le degré de redondance en fonction de plusieurs facteurs clé. Cependant elle ne détermine pas quels sont les services à réserver, et sur quelle base ils sont choisis. Dans notre approche, une sélection des services Web les plus fiables est proposée, en fonction des exigences clients et de l'historique d'exécution des services Web dans les différents groupes de diversité.

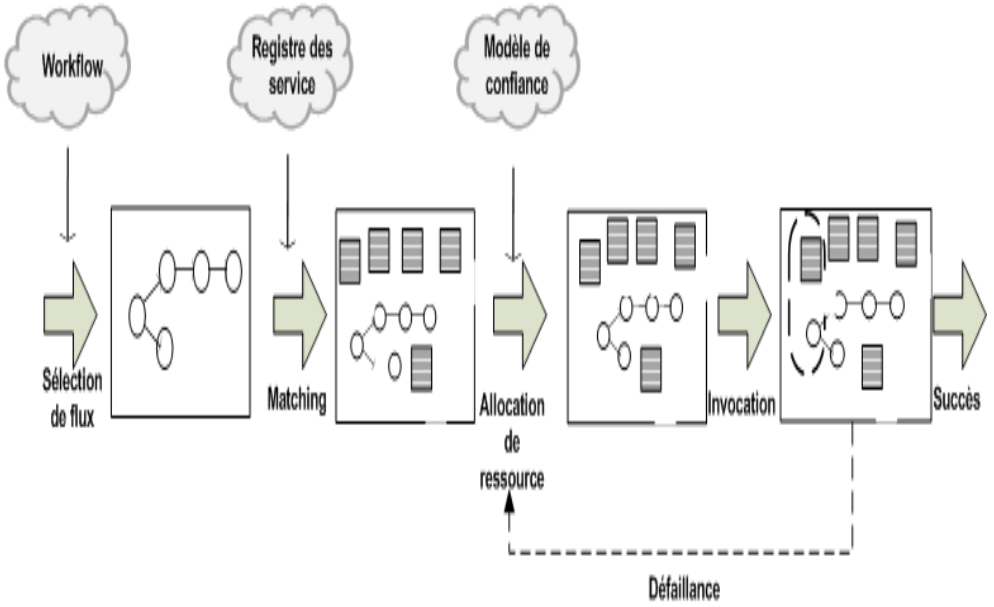

FIGURE 11 – **Cycle de vie d'un workflow**

0.3.3.3 Suitable Adaptation State (SAS)

Fabre and al. [**?**] abordent la question cruciale d'adaptation des logiciels en ligne, consistant à déterminer si le système est dans un état adaptable ou pas. En effet, il s'agit d'établir les propriétés et les moyens de contrôler l'activité du système, pour atteindre un état dans lequel la modification de la configuration du logiciel peut être effectuée sans l'introduction d'un comportement incorrect 12. Pour cela, les auteurs définissent la notion de SAS (Suitable Adaptation State) représentant un état dans lequel la modification d'un composant peut être effectuée en toute sécurité.

Chaque composant individuel dans une configuration a son propre modèle réseau de Petri qui représente à la fois ses interactions avec d'autres composants et la causalité entre les tâches internes. L'approche proposée permet de décider quand le système est adaptable ou pas et d'orienter le système vers un état adaptable. Elle est appliquée à l'adaptation en ligne des mécanismes de tolérance aux fautes. En effet ces mécanismes sont considérés comme un ensemble de composants pouvant ainsi être adaptés en ligne selon certaines conditions d'utilisation particulières et d'évolution de la configuration du système.

L'adaptation de la tolérance aux fautes est souhaitable pour la maintenabilité du logiciel, mais aussi pour faire face aux changements dans le contexte opérationnel (e.g., environnement, hypothèses sur les fautes, manque de ressources, etc.). L'adaptation consiste à passer d'une stratégie à une autre en réponse à certains changements dans des conditions environnementales et des ressources.

Cette approche est basée sur l'étude de cas sur de mécanismes/middlewares de tolérance aux fautes où les conditions environnementales et de ressources permettent

de décider si un switch vers une autre stratégie de tolérance aux fautes est nécessaire. Cependant, elle ne prend pas en considération le changement du contexte d'un point de vue de nombre de fautes observées. En réalité, une application tolérante aux fautes est définie en termes de nombre de répliques, variant selon le nombre de fautes et le type de fautes observées dans le système.

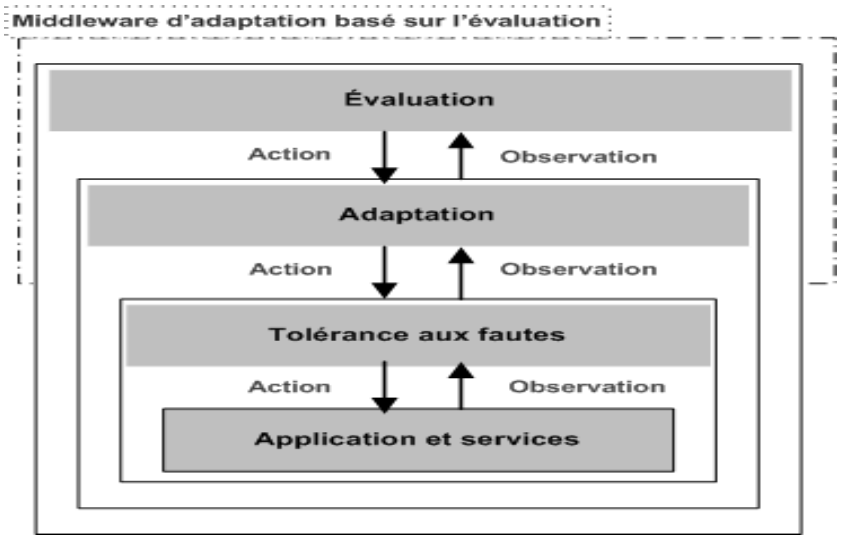

FIGURE 12 – **Overview de middelware d'adaptation**

0.4 Discussion

Parmi les nombreuses études visant à améliorer la sûreté de fonctionnement des services Web au niveau composant et composite, plusieurs approches se basent sur la réplication classique. Cependant, peu de travaux utilisent la diversité pour rendre les services Web composant et composite tolérants aux fautes. En règle générale, il est impossible de faire face à toute sorte de fautes dans une seule et unique solution. Par conséquent, les différentes approches sont basées sur des hypothèses de fautes à tolérer. La diversité est la meilleure solution pour éviter les fautes de mode commun pouvant persister dans des répliques identiques. Il existe deux façons d'appliquer la diversité des services, le protocole séquentiel tel qu'il est utilisé dans le *RecoveryBlocks* et le protocole parallèle tel qu'il est utilisé dans le $N - VersionProgramming$

Les stratégies de diversité ci-dessus ont été utilisées dans quelques solutions existantes. Contrairement aux mécanismes et protocoles de tolérance aux fautes basés sur la réplication qui sont bien formalisés et le contexte de leur application bien défini, ceux de la diversité restent un problème ouvert. De plus, les travaux existants manquent de détails sur la sélection des protocoles adéquats pour tolérer un type particulier de fautes, ainsi que le nombre de services Web à diverses utilisés en tenant compte de la criticité

de la fonctionnalité implémentée par les services.

En effet, il n'existe aucune garantie dans une composition de services Web que l'ensemble des services la composant soit fiable. Un degré de criticité doit être attribué pour mesurer la gravité de la défaillance sur la composition. Aussi, le choix du nombre des services Web à exécuter dépend de cette mesure. D'autre part, une mauvaise sélection des services Web conduit à de mauvais résultats. Une sélection des services appropriés améliore considérablement la sûreté de fonctionnement de l'ensemble de l'application. Par exemple, dans le protocole parallèle les services peuvent achever un consensus avec une valeur erronée si l'algorithme sélectionne les mauvais services similaires. Aussi, les travaux utilisant la diversité comme moyen pour tolérer les fautes, discutent rarement des stratégies de sélection.

Pour résumer, plusieurs problèmes subsistant encore dans la tolérance aux fautes dans les services composants et composites et non traités de manière satisfaisante :

1. Dans un premier temps, Comment concevoir et superviser le fonctionnement d'une application à base de services qui soit tolérante aux fautes et qui utilise la notion de diversité.
2. La détermination de nombre de répliques similaires pour chaque fonctionnalité, selon sa criticité.
3. La configuration de la diversité de services pour une meilleure sûreté de fonctionnement des applications à base de services. Plus précisément, comment identifier les meilleurs k services dans une diversité pour un protocole donné, comment les services interagissent entre eux (vote, ordonnancement).

0.5 Conclusion

La sûreté de fonctionnement des services Web est un domaine de recherche actif et important. L'architecture distribuée avec le couplage faible des services Web a apporté des avantages pour le développement d'applications distribuées plus flexibles et hétérogènes. Cependant, une telle architecture est par nature peu fiable. La recherche sur la sûreté de fonctionnement des applications à base de services Web doit faire face à deux types de défaillances à savoir les défaillances par crash et les défaillances Byzantines.

De nombreuses approches ont été développées pour assurer la fiabilité du service Web. Cependant, notre analyse montre que des limites de ces solutions empêchent leur efficacité. De nouvelles solutions sont nécessaires pour améliorer la fiabilité des applications de service Web du point de vue des clients afin de minimiser les problèmes causés par les défaillances des services. De nouvelles techniques sont également nécessaires pour améliorer l'efficacité de ces solutions en utilisant explicitement des stratégies de diversité de services et en utilisant les services les plus fiables pour assurer des exécutions des services sûrs de fonctionnement.

Ayant motivé notre choix d'utilisation de la diversité comme une solution alternative à la réplication, les chapitres suivants présentent notre solution basée sur la diversité des services Web pour l'amélioration de leur fiabilité.

Bibliographie

[ACKL87] A. Avižienis, W.C. Carter, H. Kopetz, and J.C. Laprie. *The Evolution of fault-tolerant computing*. Dependable computing and fault-tolerant systems. Springer-Verlag, 1987. :1987

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1) :11–33, January 2004. :2004

[AoNuTCS07] T. Anderson and University of Newcastle upon Tyne. Computing Science. *The ReSIST Resilience Knowledge Base*. Technical report series. University of Newcastle upon Tyne, Computing Science, 2007. :2007

[Avi85] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. Software Eng.*, 11(12) :1491–1501, 1985. :1985

[BCP+] S. Bonomi, V. Colaianni, F. Patrizi, D. Pozzi, R. Russo, and M. Mecella. Swsce – an automatic semantic web service composition engine. 4

[BLGC10] Diana Borrego, María Teresa Gómez López, Rafael M. Gasca, and Rafael Ceballos. Improving the diagnosability of business process management systems using test points. In *Business Process Management Workshops*, pages 194–200, 2010. :2010

[BWM07] Stefan Bruning, Stephan Weissleder, and Miroslaw Malek. A fault taxonomy for service-oriented architecture. In *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium*, HASE '07, pages 367–368, Washington, DC, USA, 2007. IEEE Computer Society. :2007

[CBS+09] K. S. Chan, Judith Bishop, Johan Steyn, Luciano Baresi, and Sam Guinea. Service-oriented computing - icsoc 2007 workshops. chapter A Fault Taxonomy for Web Service Composition, pages 363–375. Springer-Verlag, Berlin, Heidelberg, 2009. :2009

[Cha02] J.M. Chauvet. *Services Web avec SOAP, WDSL, UDDI, ebXML...* Solutions d'entreprise. Eyrolles, 2002. :2002

[Cha04] David A. Chappell. *Enterprise service bus - theory in practice*. O'Reilly, 2004. :2004

[CLK12] Du Wan Cheun, Hyun Jung La, and Soo Dong Kim. A taxonomic framework for autonomous service management in service-oriented architecture. *Journal of Zhejiang University - Science C*, 13(5) :339–354, 2012. :2012

[CLM06] Pat Pik-Wah Chan, Michael R. Lyu, and Miroslaw Malek. Making services fault tolerant. In *ISAS*, pages 43–61, 2006. :2006

[Con] World Wide Web Consortium. Hypertext transfer protocol (http). 3

[CR08] Yuhui Chen and Alexander Romanovsky. Improving the dependability of web services integration. *IT Professional*, 10(3) :29–35, 2008. :2008

[CS01] Fabio Casati and Ming-Chien Shan. Models and languages for describing and discovering e-services. In *SIGMOD Conference*, page 626, 2001. :2001

[DGR04] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Softw. Eng.*, 30(12), 2004. 10

[DHS05] Glen Dobson, Stephen Hall, and Ian Sommerville. A container-based approach to fault tolerance in service-oriented architectures, 2005. :2005

[DJ07] S. Dustdar and L. Juszczyk. Dynamic replication and synchronization of web services for high availability in mobile ad-hoc networks. *Service Oriented Computing and Applications*, 1(1), 2007. 15

[DK10] Rébecca Deneckère and Elena Kornyshova. La variabilité due à la sensibilité au contexte dans les processus téléologiques. In *INFORSID*, pages 161–176, 2010. :2010

[Erl07] Thomas Erl. *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007. :2007

[FB02] Dieter Fensel and Christoph Bussler. Semantic web enabled web services. *Lecture notes in computer*, 2342 :1–2, 2002. :2002

[FCMJ12] R. Z. Frantz, R. Corchuelo, and C. Molina-Jiménez. A proposal to detect errors in enterprise application integration solutions. *Journal of System and Software*, 85(3), 2012. 12

[Fie] Roy Thomas Fielding. http ://www.ics.uci.edu/ fielding/pubs/dissertation/rest-arch-style.htm. 3

[FKP10] Jean-Charles Fabre, Marc-Olivier Killijian, and Thomas Pareaud. Towards on-line adaptation of fault tolerance mechanisms. In *EDCC*, pages 45–54, 2010. :2010

[GKR09] Anatoliy Gorbenko, Vyacheslav S. Kharchenko, and Alexander Romanovsky. Using inherent service redundancy and diversity to ensure web services dependability. In *Methods, Models and Tools for Fault Tolerance*, pages 324–341. 2009. :2009

[HB04] Hugo Haas and Allen Brown. Web Services Glossary - W3C Working Group Note. Technical report, World Wide Web Consortium - W3C, 11 February 2004. :2004

[HSS+04] Jason O. Hallstrom, Nigamanth Sridhar, Paolo A. G. Sivilotti, Anish Arora, and William M. Leal. A container-based approach to object-oriented product lines. *JOURNAL OF OBJECT TECHNOLOGY*, 2004. :2004

[HXW+07] Yu Huang, Chunxiang Xu, Hanpin Wang, Yunni Xia, Jiaqi Zhu, and Cheng Zhu. Formalizing web service choreography interface. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops - Volume 02*, AINAW '07, pages 576–581, Washington, DC, USA, 2007. IEEE Computer Society. :2007

[Jur06] Matjaz B. Juric. *Business Process Execution Language for Web Services BPEL and BPEL4WS 2nd Edition*. Packt Publishing, 2006. :2006

[KSK] Markus Keidl, Stefan Seltzsam, and Alfons Kemper. Reliable web service execution and deployment in dynamic environments. In *In Proceedings of the International Workshop on Technologies for E-Services (TES*, pages 104–118. 18

[KSK03] Markus Keidl, Stefan Seltzsam, and Alfons Kemper. Reliable web service execution and deployment in dynamic environments. In *TES*, pages 104–118, 2003. :2003

[Lap04] Jean-Claude Laprie. Dependable computing : Concepts, challenges, directions. In *COMPSAC*, page 242, 2004. :2004

[LCR06] Peter Li, Yuhui Chen, and Alexander Romanovsky. Measuring the dependability of web services for use in e-science experiments. In *Proceedings of the Third international conference on Service Availability*, ISAS'06, pages 193–205, Berlin, Heidelberg, 2006. Springer-Verlag. :2006

[LHS08] Lingxi Li, Christoforos N. Hadjicostis, and R. S. Sreenivas. Designs of bisimilar petri net controllers with fault tolerance capabilities. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 38(1) :207–217, 2008. :2008

[LMX05] Nik Looker, Malcolm Munro, and Jie Xu. Increasing web service dependability through consensus voting. In *COMPSAC (2)*, pages 66–69, 2005. :2005

[LRD11] Christine Louberry, Philippe Roose, and Marc Dalmau. Kalimucho : Contextual deployment for qos management. In *DAIS*, pages 43–56, 2011. :2011

[MSR77] P. M. Melliar-Smith and Brian Randell. Software reliability : The role of programmed exception handling. In *Language Design for Reliable Software*, pages 95–100, 1977. :1977

[OAS04] *Universal Description, Discovery, and Integration (UDDI) - Version 3*, October,2004. UDDI Spec Technical Committee Draft. :2004

[Org07] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*, April 2007. 11

[PTDL03] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing. *Communications of the ACM*, 46, 2003. 11

[Ran75] Brian Randell. System structure for software fault tolerance. *IEEE Trans. Software Eng.*, 1(2) :221–232, 1975. :1975

[RRS+97] Alexander Romanovsky, Brian Randell, Robert J. Stroud, Jie Xu, and Avelino Francisco Zorzo. Implementation of blocking coordinated atomic actions based on forward error recovery. *Journal of Systems Architecture*, 43(10) :687–699, 1997. :1997

[SF07] Nicolas Salatge and Jean-Charles Fabre. Fault tolerance connectors for unreliable web services. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN'07, pages 51–60, Washington, DC, USA, 2007. IEEE Computer Society. :2007

[SHD05] Ian Sommerville, Stephen Hall, and Glen Dobson. Dependable service engineering ; a fault-tolerance based approach dependable service engineering : A fault-tolerance based abstract approach, 2005. :2005

[SOA02] *SOAP Version 1.2 Part 0 : Primer*, 2002. Working Draft. :2002

[SPJ11] Sebastian Stein, Terry R. Payne, and Nicholas R. Jennings. Robust execution of service workflows using redundancy and advance reservations. *IEEE Trans. Serv. Comput.*, 4(2) :125–139, April 2011. :2011

[TIRL03] F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy. Coordinated forward error recovery for composite web services. In *In Symposium on Reliable Distributed Systems (SRDS*, 2003. 15

[toE00] toExcel. *Extensible Markup Language (Xml) 1.0 Specifications : From the W3c Recommendations*. iUniverse, Incorporated, 2000. :2000

[YD07] Yuhong Yan and Philippe Dague. Modeling and diagnosing orchestratedweb service processes. In *ICWS*, pages 51–59, 2007. :2007