



---

# MEMOIRE

*Présenté à*

L'École Nationale d'Ingénieurs de Sfax

*en vue de l'obtention du*

MASTERE

***Nouvelles Technologies des Systèmes Informatiques Dédiés***

*Par*

**Afef JMAL épouse MAÂLEJ**

*(Ingénieur Informatique)*

---

Évaluation de Politiques d'Auto-  
Adaptabilité basées sur la Mobilité des  
Services Web Orchestrés

---

*Soutenu le 30 Juillet 2009, devant le jury composé de :*

|                      |                   |
|----------------------|-------------------|
| M. Khalil DRIRA      | <i>Président</i>  |
| M. Ahmed HADJ KACEM  | <i>Rapporteur</i> |
| M. Mohamed JMAIEL    | <i>Encadrant</i>  |
| Mme. Soumaya MARZOUK | <i>Encadrant</i>  |

*A mes parents,  
A mon mari,  
A mon frère et mes sœurs,  
A tous ceux que je n'ai pas cités  
et qui ne me sont pas moins chers.*

# Remerciements

J'ai l'honneur de réserver cette page pour témoigner ma reconnaissance à tous ceux qui ont contribué à ce travail.

Je tiens, tout d'abord, à remercier vivement Monsieur Mohamed JMAIEL, professeur et directeur du Département de Génie Informatique et de Mathématiques Appliquées à l'Ecole Nationale d'Ingénieurs de Sfax (ENI-Sfax) et Madame Soumaya MARZOUK, assistante contractuelle à l'ENI-Sfax, pour leur assistance et leurs recommandations continues. C'est grâce à leur aide et leurs conseils précieux que ce travail a vu le jour. Qu'ils trouvent ici le témoignage de mes profondes et sincères gratitudees.

Je tiens à témoigner mes vifs remerciements à Monsieur Khalil DRIRA, chargé de Recherche à LAAS-CNRS, Toulouse, pour avoir accepté de présider le comité d'examen.

Je n'oublie surtout pas Monsieur Ahmed HADJ KACEM, professeur à la Faculté des Sciences Economiques et de Gestion de Sfax (FSEG-Sfax) pour l'honneur qu'il m'a fait en acceptant de lire et d'évaluer le manuscrit et en l'accompagnant de nombreuses suggestions pertinentes.

Enfin, j'exprime ma gratitude aux membres de l'unité de Recherche en Développement et Contrôle d'Applications Distribuées (ReDCAD) à l'ENI-Sfax, pour l'atmosphère amicale que nous avons partagée pendant les deux dernières années et pour m'avoir aidée tout au long de l'élaboration de ce projet.

# Table des matières

|   |           |
|---|-----------|
| <b>Introduction générale</b>  | <b>1</b>  |
| <b>1 Concepts de base</b>   | <b>3</b>  |
| 1.1 Introduction . . . . .  | 3         |
| 1.2 Contexte général . . . . .  | 4         |
| 1.2.1 Les services Web . . . . .  | 4         |
| 1.2.2 Composition des services Web . . . . .                              | 8         |
| 1.2.3 La mobilité forte . . . . .   | 12        |
| 1.2.4 L'auto-adaptabilité . . . . .                                       | 13        |
| 1.3 Problématique et objectifs . . . . .                                  | 15        |
| 1.4 Etat de l'art . . . . .   | 16        |
| 1.4.1 Cas des applications distribuées . . . . .                          | 16        |
| 1.4.2 Cas des services Web . . . . .                                      | 17        |
| 1.4.3 Synthèse . . . . .  | 19        |
| 1.5 Conclusion . . . . .  | 19        |
| <b>2 Démarche d'évaluation des actions de reconfiguration</b>             | <b>20</b> |
| 2.1 Introduction . . . . .  | 20        |
| 2.2 Principe de notre démarche . . . . .                                  | 20        |
| 2.3 Définition des actions d'adaptation . . . . .                         | 21        |
| 2.3.1 Principe de la mobilité . . . . .                                   | 21        |
| 2.3.2 Actions de checkpoint . . . . .                                     | 22        |
| 2.3.3 Actions de mobilité . . . . .                                       | 22        |
| 2.3.4 Synthèse . . . . .  | 24        |
| 2.4 Définition des paramètres décrivant le contexte d'exécution . . . . . | 26        |
| 2.4.1 Paramètres relatifs au processus d'orchestration . . . . .          | 26        |
| 2.4.2 Paramètres relatifs à l'environnement d'exécution . . . . .         | 26        |
| 2.4.3 Paramètres relatifs à la nature du problème rencontré . . . . .     | 27        |
| 2.4.4 Synthèse . . . . .  | 27        |
| 2.5 Formulation du problème . . . . .                                     | 29        |

|          |  |           |
|----------|--|-----------|
| 2.6      | Démarche d'évaluation de politiques d'auto-adaptabilité . . . . .                                | 30        |
| 2.6.1    | Première question : Comment faire les checkpoints? . . . . .                                     | 31        |
| 2.6.2    | Deuxième question : Attendre ou migrer? . . . . .  | 39        |
| 2.6.3    | Troisième question : Quelles sont les instances à migrer? . . . . .                              | 44        |
| 2.6.4    | Synthèse : Politiques d'auto-adaptabilité des processus d'orchestration . . . . .                | 46        |
| 2.7      | Conclusion . . . . .   | 48        |
| <b>3</b> | <b>Mise en œuvre de notre solution</b>   | <b>49</b> |
| 3.1      | Introduction . . . . .   | 49        |
| 3.2      | Principe de la mobilité forte appliquée aux processus d'orchestration . . . . .                  | 49        |
| 3.2.1    | Architecture d'un processus d'orchestration mobile . . . . .                                     | 49        |
| 3.2.2    | Migration du processus d'orchestration . . . . .   | 55        |
| 3.3      | Etude de cas d'une agence de voyage . . . . .  | 63        |
| 3.3.1    | Réalisation d'une étude de cas : Processus d'orchestration BPEL d'une agence de voyage . . . . . | 63        |
| 3.3.2    | Transformation du processus réalisé en un processus fortement mobile . . . . .                   | 65        |
| 3.4      | Mise en oeuvre des différents mécanismes de checkpoint . . . . .                                 | 69        |
| 3.4.1    | Architecture de déploiement de notre solution . . . . .  | 69        |
| 3.4.2    | Evaluation de la performance du mécanisme de checkpoint périodique . . . . .                     | 71        |
| 3.4.3    | Evaluation de la performance du mécanisme de checkpoint adaptatif . . . . .                      | 73        |
| 3.5      | Conclusion . . . . .   | 80        |
|          | <b>Conclusion générale et perspectives</b>   | <b>80</b> |
|          | <b>Bibliographie</b>   | <b>82</b> |

# Table des figures

|      |   |    |
|------|---|----|
| 1.1  | Architecture des services Web . . . . .   | 6  |
| 1.2  | Déploiement, découverte et invocation de services Web . . . . .   | 7  |
| 1.3  | Architecture WSDL . . . . .   | 8  |
| 1.4  | Orchestration de services Web . . . . .   | 9  |
| 1.5  | Chorégraphie de services Web . . . . .  | 10 |
| 1.6  | Processus d'auto-adaptation . . . . .   | 14 |
| 2.1  | Illustration graphique du principe de la mobilité . . . . .   | 21 |
| 2.2  | Illustration graphique des scénarios de mobilité . . . . .  | 24 |
| 2.3  | Exemple d'utilisation d'un mécanisme sans checkpoint . . . . .  | 31 |
| 2.4  | Surcoût de la variation de la MTBF en fixant les positions de checkpoint  | 32 |
| 2.5  | Surcoût de la variation des positions de checkpoint en fixant la MTBF   | 34 |
| 2.6  | Surcoût de la variation du CV sur le temps d'exécution du processus<br>BPEL . . . . .                               | 35 |
| 2.7  | Surcoût de la variation du nombre de branches en parallèle du pro-<br>cessus BPEL . . . . .                         | 36 |
| 2.8  | Surcoût de la variation de la taille des variables du processus BPEL<br>pour 2 branches en parallèle . . . . .      | 37 |
| 2.9  | Surcoût de la variation du nombre de clients sur l'exécution d'un<br>même bloc "Flow" . . . . .                     | 39 |
| 2.10 | Cas d'une dégradation non grave au niveau du réseau . . . . .   | 41 |
| 2.11 | Cas d'une dégradation non grave au niveau du noeud . . . . .  | 42 |
| 2.12 | Cas d'une dégradation non grave au niveau du réseau et du noeud . .   | 43 |
| 2.13 | Variation du nombre d'instances en fonction du temps d'exécution . .  | 44 |
| 2.14 | Surcoût de la variation du TSLC sur le temps d'exécution du proces-<br>sus BPEL . . . . .                           | 45 |
| 2.15 | Surcoût de la variation du nombre de migration par instance sur le<br>temps d'exécution du processus BPEL . . . . . | 46 |
| 3.1  | Architecture basée sur le mécanisme de checkpoint périodique . . . .  | 51 |

|      |  |    |
|------|--|----|
| 3.2  | Architecture basée sur le mécanisme de checkpoint adaptatif . . . . .                          | 54 |
| 3.3  | Exemple d'Aspect déployé aux barrières naturelles . . . . .                                    | 54 |
| 3.4  | Exemple d'Aspect d'un checkpoint forcé . . . . .   | 55 |
| 3.5  | Règle de transformation des activités de base . . . . .  | 56 |
| 3.6  | Règle de transformation de l'activité link . . . . .   | 57 |
| 3.7  | Règle de transformation de l'activité while . . . . .  | 58 |
| 3.8  | Capture de l'état d'exécution du processus BPEL dans une structure séquentielle . . . . .      | 59 |
| 3.9  | Synchronisation des activités dans une structure « flow » . . . . .                            | 60 |
| 3.10 | Transformation des liens des activités au sein d'une structure « flow »                        | 61 |
| 3.11 | Transformation assurant la réception d'un ordre de checkpoint/migration . . . . .              | 62 |
| 3.12 | Extrait du code de base du processus BPEL de l'agence de voyage . .                            | 64 |
| 3.13 | Illustration graphique du processus prototype BPEL . . . . .                                   | 65 |
| 3.14 | Extrait du code du WSIM . . . . .  | 66 |
| 3.15 | Implémentation du « WSCM.wsdl » . . . . .  | 67 |
| 3.16 | Implémentation des méthodes du « WSCM.java » . . . . .   | 68 |
| 3.17 | Architecture de déploiement de notre solution . . . . .  | 70 |
| 3.18 | Illustration graphique des positions de checkpoint dans le processus BPEL transformé . . . . . | 71 |
| 3.19 | Evaluation à distance du mécanisme de checkpoint périodique . . . .                            | 72 |
| 3.20 | Aspects implémentant un checkpoint aux barrières de synchronisation naturelles . . . . .       | 74 |
| 3.21 | Aspects implémentant un checkpoint forcé . . . . .   | 76 |
| 3.22 | Aspect de recouvrement . . . . .   | 77 |
| 3.23 | Aspect de mobilité . . . . .   | 78 |
| 3.24 | Evaluation à distance du mécanisme de checkpoint adaptatif . . . . .                           | 79 |

# Liste des tableaux

|     |   |    |
|-----|---|----|
| 2.1 | Comparaison des scénarios de mobilité . . . . .                       | 25 |
| 2.2 | Table de l'ensemble de paramètres décrivant le contexte d'exécution . | 27 |
| 2.3 | Formulation du problème . . . . .                                     | 30 |
| 2.4 | Politique de checkpoint . . . . .                                     | 47 |
| 2.5 | Politique de mobilité par instance . . . . .                          | 47 |

# Introduction Générale

Les services Web demeurent aujourd'hui un acteur principal dans la mise en œuvre des applications distribuées. De telles applications sont souvent déployées à large échelle telle que les environnements de grille. Le dynamisme et la volatilité des ressources de ces environnements engendrent des risques de dégradation des QdS<sup>1</sup> voire même des pannes des services déployés.

Ainsi, des mécanismes d'auto-réparation et d'auto-adaptabilité doivent être mis en place afin d'assurer la continuité et les QdS requises. Pour ce faire, le processus d'adaptation ou de réparation doit passer par quatre phases allant de l'étape de monitoring, celle se préoccupant de la collecte des informations pertinentes, passant par la phase d'analyse, celle responsable du diagnostic, tout en finissant par les phases de planification et d'exécution des actions de reconfiguration.

Dans cette perspective, une solution a été proposée dans le cadre des travaux de thèse de Mme Soumaya MARZOUK [29]. Elle consiste à la mobilité forte des services Web orchestrés et se base sur la capture de leurs états d'exécution (checkpoint). Cette solution permet aux processus d'orchestration de migrer vers d'autres nœuds et de reprendre leur exécution suspendue à partir du dernier checkpoint. L'approche précédemment présentée permet d'élaborer plusieurs scénarios de checkpoint et de mobilité. Cette variété de scénarios rend problématique le choix des actions appropriées qui s'adaptent à chaque contexte d'exécution. Ce qui correspond à la phase de planification du processus d'auto-réparation.

Dans ce contexte, plusieurs travaux existants dans le cadre des applications distribuées réalisent la phase de planification en se basant sur des actions de mobilité figées [10, 22, 28, 36, 20, 21, 27, 17, 15]. En effet, un seul mécanisme est utilisé quelque soit le contexte d'exécution de ces applications. Dans le cadre des services Web orchestrés, la mobilité n'a pas été proposée comme une action d'adaptation. Cependant, l'action planifiée est la substitution ou la duplication du service Web défaillant par un autre assurant les mêmes fonctionnalités [11, 25, 18, 24, 23, 30, 34, 32, 35]. En outre, ces travaux n'ont pas envisagé le cas où le processus d'orchestration constitue la source du problème. Dans une telle situation, le processus d'orchestration sera

---

<sup>1</sup>Qualité de Service

redémarré quelque soit l'instant de sa migration.

Notre contribution dans ce projet consiste alors à définir et à valider un ensemble de politiques d'auto-adaptabilité opérant dans la phase de planification, ceci en suivant une démarche d'évaluation dépendant de différents contextes d'exécution dans le cas des processus d'orchestration.

En effet, cette démarche consiste à définir les actions de reconfiguration possibles et les paramètres décrivant le contexte d'exécution. Ensuite, nous procédons à élaborer la correspondance entre ces actions et ces paramètres en réalisant plusieurs expérimentations sous différentes conditions. Cette étude nous a permis de définir des politiques de checkpoint et mobilité constituées d'un ensemble de règles décidant la méthode et la fréquence de checkpoint nécessaires ainsi que le scénario de mobilité adéquat en cas de problème.

Pour ce faire, nous avons commencé par l'implémentation de différents mécanismes de checkpoint et de mobilité des services Web orchestrés appliquée pour une étude de cas concernant une agence de voyage. Ensuite, nous avons mené une étude expérimentale afin de déterminer d'une part les puissances et les faiblesses de ces mécanismes, d'autre part d'évaluer l'apport de leur utilisation dans différentes situations telles qu'une dégradation des performances causée par le nœud, le réseau ou l'application. En outre, ces expérimentations ont été réalisées en variant les conditions d'exécution telles que la fréquence de panne de l'environnement d'exécution.

Les résultats de ces expérimentations témoignent dans un premier temps de la faisabilité et de l'efficacité des mécanismes précités sous plusieurs conditions d'exécution. Dans un deuxième temps, cette étude a permis l'élaboration de politiques de checkpoint et de mobilité déterminant ainsi les actions appropriées à adopter dans une situation donnée.

Ce mémoire, articulé autour de trois chapitres, décrit la solution que nous proposons pour la génération du plan d'actions d'auto-adaptation le plus approprié pour chaque contexte d'exécution. Le premier chapitre est consacré pour présenter le contexte général de notre projet, ceci en définissant un ensemble de technologies clés concernant les services Web et en exposant les défis qu'ils peuvent rencontrer dans des environnements dynamiques. Une autre partie sera réservée pour présenter les solutions proposées dans la littérature pour remédier à ces problèmes. Dans le deuxième chapitre, nous allons présenter notre démarche d'évaluation des politiques d'auto-adaptation des processus d'orchestration opérant dans la phase de planification. Dans le troisième chapitre, nous mettons l'accent sur l'intérêt de la solution de la mobilité forte appliquée aux services Web orchestrés. Ensuite, nous procédons à une description des implémentations et des évaluations des mécanismes de checkpoint et de mobilité utilisés.

# Chapitre 1

## Concepts de base

### 1.1 Introduction

De nos jours, les services Web deviennent de plus en plus utilisés pour la mise en œuvre d'applications distribuées. Néanmoins, le caractère dynamique des ressources des environnements où sont déployés les services Web réduit les performances de l'exécution. En effet, à tout moment une ressource peut disparaître, apparaître ou subir une dégradation de performances, causant ainsi le ralentissement de l'exécution des applications ou même l'interruption de leur exécution. De ce fait, le profit des ressources disponibles dans un tel environnement nécessite l'utilisation d'une stratégie d'auto-adaptation du système au changement de l'état des ressources, ceci afin de remédier aux fautes et d'augmenter les performances de l'exécution. Nous visons dans ce qui suit à offrir une solution pour l'auto-adaptabilité des applications distribuées à base de services Web orchestrés.

La première partie de ce chapitre est consacrée pour décrire le contexte général de notre projet. En effet, nous présentons tout d'abord un ensemble de concepts de base abordant les technologies des services Web et de leur composition. Puis, nous entamons cette partie par une explication de la mobilité forte comme étant une solution pour remédier aux problèmes dus au dynamisme de l'environnement d'exécution des services Web composés, ainsi nous définissons ensuite la notion de l'auto-adaptabilité dans ce contexte. La deuxième partie décrit la problématique posée et fixe les objectifs à atteindre dans ce projet. Dans la troisième partie, nous mettons l'accent sur les travaux de recherche qui s'intéressent au problème d'auto-adaptation dans le cadre des applications distribuées et des services Web.

## 1.2 Contexte général

A l'origine, le commerce électronique concernait principalement la vente de biens et de services au client (B2C<sup>1</sup>), mais rapidement ces échanges se sont étendus aux échanges entre entreprises (B2B<sup>2</sup>). Créés pour faciliter les échanges commerciaux, les services Web prennent leurs racines dans l'informatique distribuée et dans l'avènement du Web. La technologie des services Web a pour objectif d'uniformiser la présentation des services offerts par une entreprise et d'en rendre l'accès transparent pour tout type de plate-forme, au travers d'un certain nombre de standards d'interopérabilité. Cette notion de services Web désigne essentiellement une application mise à disposition sur Internet par un fournisseur de service, et accessible par les clients au travers de protocoles Internet standard.

### 1.2.1 Les services Web

Plusieurs définitions des services Web ont été mises en œuvre par différents auteurs. Ci-après, nous citons une définition généralement acceptée et fournie par le consortium W3C<sup>3</sup> [9] : « A Web Service is a software application identified by a URI<sup>4</sup>, whose interfaces and bindings are capable of being defined, described and discovered by XML<sup>5</sup> artefacts, and which supports direct interactions with other software applications using XML based messages via Internet-based protocols ».

Plus informellement, nous pouvons dire que les services Web sont des applications qui définissent un ensemble d'interfaces, s'appuyant sur XML, et qui peuvent interagir dynamiquement entre elles et avec d'autres applications grâce à l'échange de messages basés sur XML utilisant les protocoles de transport Internet disponibles.

#### 1.2.1.1 Architecture et infrastructure des services Web

L'architecture des services Web est une architecture orientée composant (SOA<sup>6</sup>). L'architecture SOA est un modèle qui définit un système par un ensemble de composants logiciels distribués qui fonctionnent de concert afin de réaliser une fonctionnalité globale préalablement établie. Les composants dans un système distribué n'opèrent pas dans un même environnement de traitement et sont obligés de communiquer par échanges de messages afin de solliciter des services dans le but d'accomplir le résultat souhaité.

---

<sup>1</sup>Business to Consumer

<sup>2</sup>Business to Business

<sup>3</sup>World Wide Web Consortium

<sup>4</sup>Uniform Resource Identifier

<sup>5</sup>eXtensible Markup Language

<sup>6</sup>Service Oriented Architecture

La définition de l'architecture des services Web consiste à mettre en évidence les concepts, les relations entre ces concepts ainsi qu'un ensemble de contraintes entre ces concepts. Les principaux concepts intervenant dans l'architecture des services Web sont :

- le fournisseur du service : désigne le serveur qui héberge les services déployés ;
- le client du service : représente l'application cliente qui invoque le service ;
- le service : désigne les fonctionnalités d'un agent logiciel qui implémente le service ;
- la description du service : c'est la spécification du service exprimée dans un langage de description interprétable par les machines, c'est-à-dire une description technique dans laquelle le service est vu en termes de messages, de types, de protocoles de communication et d'une adresse physique ;
- les messages : c'est la plus petite unité d'échange entre les clients et les services. La structure des messages qui permettent l'invocation d'un service doit être exprimée dans la description du service ;
- la ressource : désigne l'identifiant du service, c'est-à-dire son URI. Bien que le Web soit constitué de plates-formes totalement hétérogènes ou les intérêts des différents acteurs du marché s'entremêlent, ceci ne l'a pas empêché de se développer et d'être universel. Ce succès est dû essentiellement à l'édiction d'un ensemble de standards ouverts, dont les plus connus sont le protocole HTTP<sup>7</sup> et le format MIME<sup>8</sup>. Ensemble, ils offrent un mécanisme d'échange de données de toutes sortes quelle que soit la nature des plates-formes impliquées. Le développement des services Web a suivi la même approche en mettant en place une campagne de standardisation qui a touché les Aspects les plus importants du développement d'un modèle d'intégration d'applications hétérogènes.

L'avantage de ce modèle est de présenter ces services comme des boîtes noires. En fait, les entrées-sorties d'un service sont gérées au sein de messages dont nous connaissons le format grâce à des interfaces clairement exposées mais sur lesquelles l'implémentation interne du traitement n'influe pas au niveau de la structure. Ceci permet un haut niveau de modularité et d'interopérabilité. L'avantage du modèle de message est qu'il permet de s'abstraire de l'architecture, du langage ou encore de la plate-forme qui va supporter le service : il suffit juste que le message respecte une structure donnée pour qu'il puisse être utilisé.

L'originalité de l'infrastructure des services Web consiste à mettre en place ces services exclusivement sur la base des protocoles les plus répandus sur Internet. Ces protocoles sont répartis selon quatre axes représentés dans la figure 1.1 :

- Couche de transport : cette couche s'occupe de transporter les messages entre

---

<sup>7</sup>Hyper Text Transfer Protocol

<sup>8</sup>Multipurpose Internet Mail Extension

- applications en utilisant les protocoles HTTP, FTP<sup>9</sup> ou SMTP<sup>10</sup> ;
- Message XML : il s’agit de formaliser les messages à l’aide d’un vocabulaire XML commun (SOAP<sup>11</sup>) ;
- Description des services : description de l’interface publique des services Web (WSDL<sup>12</sup>) ;
- Recherche de services : centralisation des services et de leur description dans un référentiel commun (UDDI<sup>13</sup>).

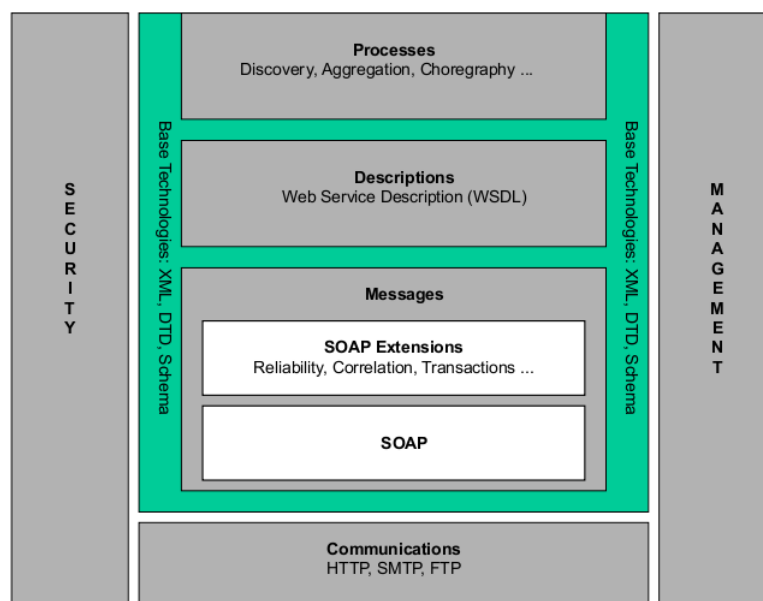


FIG. 1.1 – Architecture des services Web

### 1.2.1.2 Déploiement, recherche et invocation de services Web

Le cycle de vie d’un service Web se déroule de la façon suivante : une fois créé, le service est déployé sur le réseau (local ou Internet). Puis un utilisateur ayant des besoins spécifiques va rechercher un service correspondant à ses besoins à l’aide d’un annuaire spécialisé. Enfin, une fois le service trouvé, l’utilisateur va invoquer le service : une communication va être mise en place entre l’utilisateur et le service Web. Ce cycle de vie, représenté dans la figure 1.2, fait appel à trois grandes technologies : SOAP [8], WSDL [7], UDDI [3].

<sup>9</sup>File Transfer Protocol

<sup>10</sup>Simple Mail Transport Protocol

<sup>11</sup>Simple Object Access Protocol

<sup>12</sup>Web Services Definition Language

<sup>13</sup>Universal Description Discovery and Integration

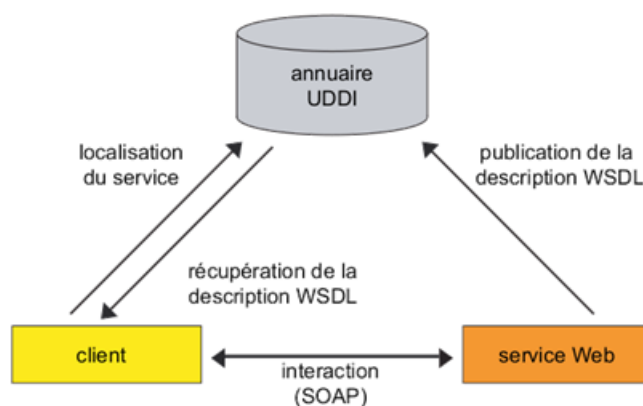


FIG. 1.2 – Déploiement, découverte et invocation de services Web

**Protocole SOAP.** Le protocole SOAP est un protocole de communication basé sur XML qui permet aux services Web d'échanger des informations dans un environnement décentralisé et distribué tout en s'affranchissant des plates-formes et des langages de programmation utilisés. Il définit un ensemble de règles pour structurer les messages envoyés. Mais SOAP ne fournit aucune instruction sur la façon d'accéder aux services Web. C'est le rôle du langage WSDL.

**Description WSDL.** La spécification WSDL joue un rôle important dans l'interopérabilité des composants services Web. Comme illustré dans la figure 1.3 et moyennant un schéma uniforme obéissant à une sémantique bien définie, cette spécification permet aux composants de définir ce qui est nécessaire à leur invocation. La spécification WSDL est définie selon une sémantique totalement indépendante du modèle de programmation de l'application. Elle sépare clairement la définition abstraite du service (échange de messages) de ses mécanismes de liaison (définition des protocoles applicatifs). Cette dernière caractéristique permet aux composants d'interagir même si l'application a été modifiée, ce qui est un point important pour assurer l'interopérabilité des services.

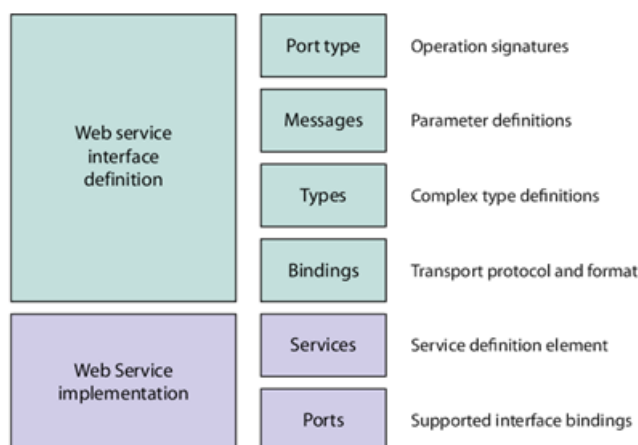


FIG. 1.3 – Architecture WSDL

**Référentiel UDDI.** Dans un environnement ouvert comme Internet, la description d'un service Web n'est d'aucune utilité s'il n'existe pas de moyen de localiser aussi bien les services Web que leurs descriptions WSDL : c'est le rôle des référentiels UDDI. La spécification UDDI constitue une norme pour les annuaires de services Web. Les fournisseurs disposent d'un schéma de description permettant de publier des données concernant leurs activités, la liste des services qu'ils offrent et les détails techniques de chaque service. La spécification offre également une API<sup>14</sup> aux applications clientes, pour consulter et extraire des données concernant un service et/ou son fournisseur.

### 1.2.2 Composition des services Web

L'intérêt des services Web est de permettre à une entreprise d'exporter, via le réseau Internet, ses compétences et son savoir-faire, ou encore d'ouvrir de nouveaux marchés et de nouveaux supports à la vente. Nous pouvons citer, par exemple, Google qui permet aux développeurs du monde entier d'utiliser le célèbre moteur de recherche directement dans leurs applications. Inversement, ces services Web permettent aux entreprises d'utiliser à moindre frais les compétences et le savoir-faire des autres entreprises.

Mais pour exploiter au maximum les avantages de ce nouveau type d'applications, la communauté des services Web doit se doter d'outils donnant la possibilité de pouvoir assembler différents services entre eux. Cet assemblage, de la même façon qu'un développeur agence les méthodes de son programme, a pour but d'effectuer des tâches qu'un simple service Web ne saurait résoudre. C'est la composition de services Web.

<sup>14</sup>Application Programming Interface

L'approche actuelle de la composition de services Web consiste à définir des processus métiers, i.e., des enchaînements réutilisables de services. Cette approche est la plus utilisée par le monde industriel.

### 1.2.2.1 Description et fonctionnement

Un service Web est dit composé ou composite lorsque son exécution implique des interactions avec d'autres services Web afin de faire appel à leurs fonctionnalités. La composition de services Web spécifie quels services ont besoin d'être invoqués, dans quel ordre et comment gérer les conditions d'exception.

La composition des services Web peut se faire de deux manières : orchestration [26] et chorégraphie [33].

#### 1.2.2.1.1 Orchestration

L'orchestration décrit l'interaction des services au niveau de messages, incluant la logique métier et l'ordre d'exécution des interactions. Les services Web n'ont pas de connaissance (et n'ont pas besoin de l'avoir) d'être mêlés dans une composition et de faire partie d'un processus métier. Seulement le coordinateur de l'orchestration a besoin de cette connaissance.

La figure 1.4 montre le flux de travail (eng. workflow) dans l'orchestration des services Web. Un coordinateur prend le contrôle de tous les services Web impliqués et coordonne l'exécution des différentes opérations des services Web qui participent dans le processus.

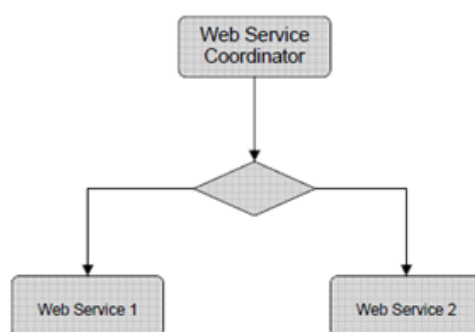


FIG. 1.4 – Orchestration de services Web

#### 1.2.2.1.2 Chorégraphie

Contrairement à l'orchestration, la chorégraphie n'a pas un coordinateur central. Chaque service Web mêlé dans la chorégraphie connaît exactement quand ses

opérations doivent être exécutées et avec qui l'interaction doit avoir lieu.

La chorégraphie est un effort de collaboration dans lequel chaque participant du processus décrit l'itération qui l'appartient. Elle trace la séquence des messages qui peut impliquer plusieurs services Web. La collaboration dans la chorégraphie des services Web peut être représentée comme illustré dans la figure 1.5 :

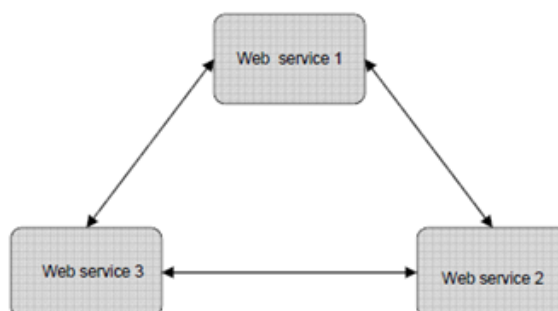


FIG. 1.5 – Chorégraphie de services Web

Depuis la perspective de la composition des services Web, l'orchestration est un rapprochement plus flexible que la chorégraphie :

- Le responsable ou coordinateur de tout le processus métier est connu.
- Les services Web peuvent être incorporés sans soucis, parce qu'ils n'ont pas conscience d'appartenir à un processus métier.

### 1.2.2.2 Langage de définition : Business Process Language for Web Services (BPEL4WS)

BPEL<sup>15</sup> [2] est un langage pour les processus métiers basés sur XML. Il soutient la technologie des services Web, tel que, SOAP [8], WSDL [7], UDDI [3], WS-Reliable Messaging [4], WS-Addressing [5], WS-Coordination [6] et WS-Transaction [6]. Il est conçu pour permettre de charger/partager les données distribuées, même à travers des organismes multiples, en employant une combinaison de services Web.

BPEL décrit l'interaction des processus métiers basés sur les services Web, à la fois au sein des entreprises et entre elles. Les entreprises utilisatrices du langage BPEL pourront ainsi définir leurs processus métiers et en garantir l'interopérabilité non seulement à l'échelle de l'entreprise, mais également avec leurs partenaires commerciaux, au sein d'un environnement de services Web. BPEL rend possible l'interopérabilité entre des activités commerciales basées sur différentes technologies.

Écrit par des créateurs des systèmes de BEA<sup>16</sup>, IBM<sup>17</sup>, et le Microsoft, la spéci-

<sup>15</sup>Business Process Execution Language

<sup>16</sup>Business Enterprise Architecture

<sup>17</sup>International Business Machine

fication BPEL combine et remplace le WSFL<sup>18</sup> d'IBM et le XLANG<sup>19</sup> du Microsoft. Parfois, BPEL est appelé comme BPELWS ou BPEL4WS.

Ainsi, le langage BPEL est le résultat de l'unification et de l'évolution de différentes tentatives de standardisation des définitions des processus métiers. Il est le standard le plus complet pour la description des processus métiers. Ce langage est le plus soutenu industriellement et le mieux accepté par les développeurs. En plus, il existe plusieurs outils de développement qui peuvent nous aider dans le cadre de notre travail. Pour ces raisons, BPEL est le langage que nous avons choisi pour le déroulement de ce projet. Ainsi, il convient de le présenter avec plus de détail. En effet, BPEL supporte deux types différents de processus [26] :

- Les processus exécutables : nous permettent de spécifier les détails du processus métier. Ils peuvent être exécutés au moyen d'un moteur d'orchestration. Dans la majorité des cas, BPEL est utilisé dans ce type de processus.
- Les processus abstraits : nous permettent de spécifier l'échange de messages entre partenaires du processus. Ils ne sont pas exécutables.

Le processus BPEL spécifie l'ordre exact de l'invocation des services Web participants dans la composition. Cette invocation peut se faire soit en parallèle soit séquentiellement. Nous pouvons conditionner le comportement, par exemple, lorsque l'invocation d'un service Web peut être dépendante du résultat d'une invocation antérieure. La construction des boucles, la déclaration de variables, la copie et l'assignement des valeurs, etc. sont aussi réalisables. De plus, il est possible de combiner toutes ces constructions et de définir des processus métiers complexes d'une manière algorithmique [26].

Comme nous venons d'indiquer, un processus métier correspond à une séquence d'opérations ou plus exactement à un flux d'activités. Ces activités peuvent faire intervenir un à plusieurs services Web. Le langage BPEL permet l'utilisation de deux types d'activités : les activités de base et les activités structurées. Les activités de base de ce langage permettent :

- d'invoquer une opération d'un service Web tiers (activité *invoke*),
- de présenter la composition comme un nouveau service Web avec l'activité *receive* pour décrire la réception d'une requête et l'activité *reply* pour générer une réponse. Les activités structurées utilisent les activités de base pour décrire :
- des séquences ordonnées (*sequence*) et des exécutions en parallèle (*flow*),
- des branchements (*switch, if*) et des boucles (*while*),
- des chemins alternatifs (*pick*).

BPEL permet aussi de déclarer des variables, avec *variable*, et de définir des liens

---

<sup>18</sup>Web Services Flow Language

<sup>19</sup>XML Business Process Language

des services Web partenaires, avec *partnerLink*. Nous pouvons définir un *partnerLink* comme un lien généré entre le processus et un service Web pendant l'invocation de ce dernier ou, aussi, comme un lien créé entre le client qui invoque un processus BPEL et le processus lui-même. Il faut, obligatoirement, avoir au moins un client *partnerLink*.

Ce langage intègre également un mécanisme de gestion des exceptions (*throw*, *catch*), ainsi qu'un mécanisme de compensation (*scope*) qui permet d'annuler une transaction dans son intégralité lorsque celle-ci échoue. Il constitue une couche supérieure au langage de description WSDL. Il utilise, en effet, WSDL pour définir les opérations de services Web élémentaires à appeler et pour présenter le processus métier comme un nouveau service Web.

### 1.2.3 La mobilité forte

La mobilité est un terme général désignant la migration de tout composant logiciel à travers un réseau informatique d'une ressource à une autre pour être exécuté à sa destination.

Les applications distribuées utilisent des ressources et des services distants via un réseau. En conséquence, elles doivent être capables de réagir dynamiquement aux changements de leur environnement. La mobilité de code est un mécanisme qui permet à des programmes d'être envoyés vers des sites distants et exécutés à leur arrivée. Ce type de mobilité a comme avantage d'offrir une meilleure utilisation de la capacité du réseau (ainsi les machines possèdent des charges égales), et une diminution du trafic par une minimisation des interactions à travers le réseau.

Les services Web constituent un élément de base dans l'implémentation d'applications distribuées. De telles applications sont souvent déployées sur large échelle telle que les environnements de grille. Le dynamisme et la volatilité des ressources de ces environnements engendrent des risques de dégradation des QoS causant parfois des pannes des services déployés.

Dans de telles situations, la mobilité forte constitue une solution efficace permettant l'auto-adaptation des services Web au dynamisme de leur environnement. En effet, il s'agit d'un type particulier de mobilité permettant aux services de migrer, vers d'autres nœuds plus performants, durant leur exécution tout en gardant consistant leur état d'exécution. Les services suspendus peuvent alors reprendre leur exécution à partir de leur point d'interruption [14]. Pour qu'un service soit fortement mobile, la migration du service doit prendre en compte le code, l'état des données et l'état de son exécution.

Dans ce contexte, une solution a été proposée afin de remédier aux problèmes causés par l'instabilité des environnements d'exécution des services Web orchestrés

en particulier. Elle consiste en l'auto-adaptabilité basée sur la mobilité forte de ce type de services. Cette approche s'inscrit dans le cadre des travaux de thèse de Mme Soumaya MARZOUK [29]. En effet, cette solution permet aux processus d'orchestration de migrer vers d'autres nœuds plus performants et de reprendre leur exécution suspendue à partir de leur point d'interruption. Plusieurs mécanismes de checkpoint et scénarios de mobilité sont élaborés dans cette approche. Il convient de préciser qu'il est possible de varier les instants et les méthodes de checkpoint. En plus, la mobilité peut prendre plusieurs formes, telles que la ré exécution de tout le code dans le nœud destinataire suite à la migration du processus. Aussi, la mobilité peut consister à la migration instantanée puis le retour au dernier checkpoint (rollback). Une autre variété de la mobilité consiste à attendre le prochain checkpoint puis migrer. Il est aussi envisageable de faire un checkpoint instantané juste avant de migrer et de poursuivre l'exécution à partir de celui-ci. Outre cela, nous pouvons migrer une ou plusieurs instances du processus d'orchestration vers différents nœuds en cas de problème.

#### 1.2.4 L'auto-adaptabilité

Nous qualifions un processus d'orchestration d'auto-adaptable lorsqu'il réalise lui-même sa propre adaptation à son environnement. Ceci suppose que le processus possède une connaissance de son contexte d'exécution, à tous les niveaux, y compris les données de l'exécution en cours, le moyen de se modifier lui-même, enfin la capacité à décider quand et comment il doit s'adapter par rapport à son état courant.

Dans ce contexte, le processus d'auto-adaptabilité s'avère intéressant en tant qu'un mécanisme permettant d'assurer un fonctionnement acceptable des services Web basiques ou composés. En effet, le processus d'auto-réparation consiste à surveiller le service Web dans une première phase. Les résultats de cette première étape seront par la suite envoyés à la phase d'analyse. Cette dernière se charge du diagnostic du service afin de caractériser son état. Les rapports de diagnostic ainsi générés seront exploités pour reconfigurer le service. Les décisions prises lors de la phase de planification seront mises en place lors de l'exécution. La figure 1.6 résume le processus d'auto-réparation ainsi que ses quatre phases.

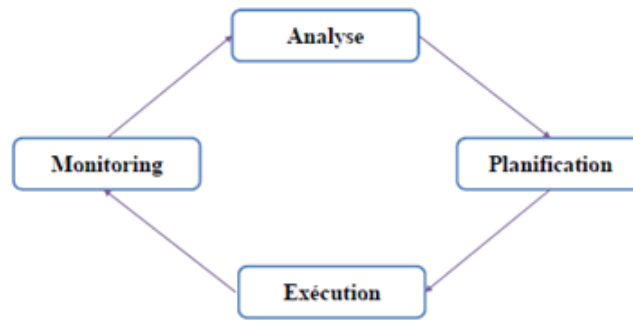


FIG. 1.6 – Processus d’auto-adaptation

#### 1.2.4.1 Première phase : Le monitoring (La surveillance)

La mise en place d’une stratégie de garantie de qualité de service et de prédiction de panne doit se baser sur un ensemble de mesures des attributs de la QdS pouvant être exploitées pour une prise de décision ultérieure. Cette phase correspond à la supervision de l’application. En effet, elle consiste à collecter les informations concernant les ressources existantes afin de les fournir aux étapes qui suivent dans le processus d’auto-adaptabilité.

#### 1.2.4.2 Deuxième phase : L’analyse

Cette phase présente une stratégie d’exploitation des mesures des QdS tirées à partir de la phase précédente dans l’objectif d’une prise de décision efficace et correcte. Elle consiste à analyser l’information produite par la phase de monitoring dans le but d’identifier le statut du système.

#### 1.2.4.3 Troisième phase : La planification

Cette phase est déclenchée lorsque la phase d’analyse détecte une panne, elle consiste à étudier les rapports de diagnostic générés lors de cette dernière phase afin de prendre les décisions de reconfiguration nécessaires. Ainsi, la phase de planification consiste à générer un plan d’actions nécessaire pour l’ajustement du comportement du système en se basant sur l’analyse réalisée lors de la phase précédente. Cette génération est guidée par des règles permettant l’identification du mécanisme d’adaptation le plus adéquat au contexte d’exécution de l’application.

#### 1.2.4.4 Quatrième phase : L’exécution

C’est la dernière étape du processus d’auto-adaptabilité, elle correspond à l’exécution des actions d’adaptation pour maintenir ou rétablir une QdS acceptable.

## 1.3 Problématique et objectifs

Comme déjà décrit, la solution de la mobilité forte des services Web orchestrés offre plusieurs possibilités et mécanismes pour gérer la mobilité. Chacun de ces mécanismes possède ses propres points forts et points faibles. Cependant, l'utilisation de politiques figées ne garantit pas toujours un bon résultat. Nous décrivons dans ce qui suit deux exemples illustrant cette problématique.

Comme premier exemple, nous supposons que la MTBF<sup>20</sup> est inférieure au temps d'exécution total d'un processus d'orchestration. En plus, nous présumons que la politique d'auto-adaptation utilisée est la mobilité de code, c'est-à-dire suite à une décision de mobilité, tout le processus d'orchestration sera ré exécuté. Dans ce cas, nous notons qu'en adoptant le mécanisme précédemment décrit au sein d'un environnement caractérisé par une fréquence de pannes importante, la probabilité de fin d'exécution du processus d'orchestration est assez faible, ceci en raison de la ré exécution de blocs du processus déjà réalisés plusieurs fois.

Nous supposons dans un deuxième exemple que le mécanisme d'auto-adaptation adopté est celui de checkpoint planifié d'une façon périodique. Dans ce cas, ce mécanisme paraît intéressant puisqu'il permet de sauvegarder l'état du processus d'orchestration à chaque période du temps de son exécution. Cette solution est efficace dans le contexte de dégradation de QoS. En effet, nous tolérons une certaine durée, entre la décision de mobilité et la réalisation du checkpoint, suffisante pour maintenir l'état d'exécution courant du processus d'orchestration. Toutefois, si nous nous trouvons dans le contexte d'une panne brusque, nous devons alors migrer instantanément sans pouvoir attendre le prochain checkpoint. Par conséquent, tout (ou une partie) du processus d'orchestration sera ré exécuté. Ainsi, nous n'allons plus bénéficier du coût d'ajout du bloc de checkpoint dans le même processus.

A travers ces exemples, nous venons de montrer que le choix de politiques d'auto-adaptation fixées d'avance est insuffisant pour garantir un bon ajustement du comportement des applications distribuées, en particulier pour les processus d'orchestration. En outre, nous mettons l'accent sur le fait que le choix de l'action de configuration doit dépendre de la nature de l'application, de l'état de l'environnement et de la nature du problème.

Dans ce contexte, la problématique qui se pose est comment choisir le mécanisme approprié pour chaque contexte d'exécution. Cette question s'inscrit dans la phase de planification du processus d'auto-adaptation et nécessite la définition d'un ensemble de règles aidant à décider à propos du choix de l'action de reconfiguration adéquate.

Ainsi, l'objectif fondamental de notre étude consiste à définir et à valider un ensemble de politiques d'auto-adaptabilité opérant dans la phase de planification, ceci

---

<sup>20</sup>Mean Time Between Failures

en suivant une démarche d'évaluation qui tient en compte différents contextes d'exécution dans le cas des processus d'orchestration. En effet, cette démarche consiste à définir en premier lieu un ensemble d'actions de reconfiguration possibles ainsi que des paramètres de mobilité. Ceux-ci dépendent du processus d'orchestration, de l'environnement d'exécution et de la nature de la panne. Ensuite, nous procédons à élaborer la correspondance entre ces actions et ces paramètres en réalisant plusieurs expérimentations dans différentes situations. Par conséquent, nous définissons des politiques de checkpoint et de mobilité nécessaires pour décider à propos des actions de reconfiguration appropriées afin de s'adapter au contexte d'exécution.

Pour ce faire, ce présent projet vise en premier lieu la mise en œuvre d'un ensemble de mécanismes appliqués pour une étude de cas de l'agence de voyage qui sera détaillée dans le troisième chapitre de ce mémoire. Ensuite, nous pointons la réalisation d'une étude expérimentale afin d'évaluer l'apport de l'utilisation de ces mécanismes sous plusieurs conditions, telles qu'un abaissement des performances au niveau du nœud, du réseau ou de l'application. Par conséquent, les résultats de ces expérimentations nous permettront de proposer un ensemble de politiques d'adaptation opérant dans la phase de planification. Il serait ainsi possible de générer pour chaque contexte d'exécution le plan d'actions le plus adéquat.

## 1.4 Etat de l'art

Dans cette section, nous allons étudier l'ensemble des travaux de recherche traitant le problème d'auto-adaptabilité au contexte d'exécution de l'environnement de déploiement des applications distribuées d'une part et des services Web d'autre part.

### 1.4.1 Cas des applications distribuées

Plusieurs travaux traitent la mobilité forte des applications distribuées et emploient différentes techniques de checkpoint. En outre, ces travaux visent plusieurs types d'applications, par exemple celles basées sur MPI<sup>21</sup> et RMI<sup>22</sup>.

En effet, le mécanisme de checkpoint peut être réalisé périodiquement et d'une façon coordonnée [10, 22]. Dans un tel cas, le rétablissement de l'application, après sa migration vers un nouveau nœud, nécessite le retour (rollback) au dernier checkpoint capturé afin de pouvoir poursuivre l'exécution suspendue. Cependant, cette solution n'est plus rentable lorsque l'application est déployée dans un environnement ayant une fréquence de pannes importante [28].

---

<sup>21</sup>Multi Protocol Interface

<sup>22</sup>Remote Method Invocation

D'autres solutions emploient aussi le mécanisme de checkpoint coordonné mais seulement au moment de la migration [36, 20, 21, 27]. Elles permettent la réduction du surcoût de checkpoint et assurent le rétablissement de l'application à partir du point d'interruption. Ces solutions sont efficaces seulement au sein d'environnements stables dont les pannes sont prédictibles d'avance.

Le mécanisme de checkpoint aux barrières de synchronisation naturelles est employé par [17], il offre un surcoût de checkpoint réduit mais l'intervalle de checkpoint est fixé. Cependant, cet intervalle doit être inférieur au temps moyen entre deux pannes consécutives noté MTBF afin d'assurer le progrès de l'application.

D'autres solutions emploient le mécanisme de checkpoint non coordonné grâce à l'exploitation des messages (eng. Message logging) [15]. De telles solutions s'avèrent les plus efficaces dans le cas d'un environnement caractérisé par une fréquence de pannes importante [28], mais possèdent le plus grand surcoût en termes de bande passante et de capacité de stockage.

Ainsi, chaque action de checkpoint est valable pour un contexte d'exécution bien déterminé. Ces conditions changent fréquemment dans les environnements à large échelle, ce qui rend la prise de décision à propos du bon choix du mécanisme de checkpoint avant l'exécution très difficile, en particulier dans le cas des applications possédant de longues durées d'exécution.

### 1.4.2 Cas des services Web

Le problème de l'auto-adaptation et de l'auto-réparation des services Web orchestrés a été largement étudié. La plupart des travaux existants s'intéressent à la résolution de ces problèmes lorsque les services Web partenaires sont à l'origine de la panne. Un tel problème est principalement résolu via la ré invocation ou la substitution [11, 25, 18, 24, 23, 30, 34, 32, 35] du service indisponible par un autre équivalent offrant les mêmes fonctionnalités.

Ezenwoye et Sadjadi offrent dans [18] une solution basée Proxy pour WS-BPEL comme étant une plate-forme garantissant une adaptation dynamique pour les services Web composés. Cette plate-forme définit un Proxy générique qui permet d'identifier ou de remplacer un service Web composant. A cette fin, un service Web composé émet des appels au Proxy pour invoquer un service Web composant au lieu de l'appeler directement. Cette solution basée Proxy augmente la disponibilité du service Web tant que le Proxy est disponible pour le service Web composé.

Ben Halima et al. proposent dans [25, 24] un middleware capable de fournir des propriétés d'auto-réparation concernant la gestion de la QoS des applications distribuées basées sur les services Web. La solution couvre tout le cycle de la gestion d'adaptation, y compris la surveillance et l'analyse des valeurs de la QoS, en plus de

la reconfiguration à base de substitution. Pour des fins de surveillance, les messages observés sont étendus avec des paramètres de QdS. Ceux-ci sont définis comme étant des métas données qui étendent les entêtes des messages SOAP des deux côtés du demandeur et du fournisseur. Dans le cas de violation de QdS, un algorithme de reconfiguration se fondant sur des « bindings » dynamiques et offrant différentes politiques de substitution est appliqué.

Mostefaoui et al. présentent dans [32] une solution basée sur les Aspects pour la conception et le développement de services autoréparables. Cette solution utilise un modèle de surveillance à base de tuples distribués dans l'espace. Ensuite, une phase de diagnostic est employée en détectant une violation de dépendance au sein des données de l'application et des flux de contrôle. L'adaptation est traitée au niveau du code à travers l'activation des Aspects qui permet le choix de la bonne stratégie d'adaptation fondée sur l'historique des exceptions et sur le contexte. Dans [35], Subramanian et al. intègrent la solution précédente au sein du moteur WS-BPEL au lieu d'utiliser l'approche basée sur les Aspects. Ainsi, cette approche permet de prédire les potentielles pannes inattendues, évaluer leur impact, décider des actions correctives et reprendre les opérations.

Autres travaux [13, 31, 12] proposent des solutions basées sur la réorganisation dynamique de la composition. Ainsi, si un service ne satisfait pas la QdS requise, toute la composition est réexaminée et peut être changée.

Pour des finalités d'autoréparation, Baresi et al. [13] proposent qu'un environnement d'exécution de services Web doit être capable d'identifier un service Web alternatif, si il existe. Dans des situations complexes, il devrait même être en mesure de réorganiser l'ensemble du processus de sélection dans lequel seuls les services Web disponibles sont invoqués. Cette solution est mise en œuvre au niveau du code et exploite des sondes qui surveillent l'exécution de la composition et proposent des activités de recouvrement afin de rendre le système capable de continuer son exécution. Dans [12], Baresi et al. étendent leur proposition en présentant une approche pour superviser les processus BPEL en exploitant les politiques et les Aspects. L'approche utilise le WSCoL<sup>23</sup> et le WSR<sub>e</sub>L<sup>24</sup> pour définir les pré- et les post-conditions aux niveaux composite et composante, et les actions de recouvrement, respectivement.

Momotko et al présentent, dans [31], une architecture fonctionnelle pour la gestion adaptative de la QdS concernant les compositions de services. Cette solution prend en charge plusieurs stratégies d'exécution basée sur la sélection dynamique et la négociation des services inclus dans une composition de service. Ainsi, en cas de violation de QdS, trois étapes sont effectuées : (1) renégocier le contrat avec le fournisseur courant de services atomiques (et, éventuellement, avec d'autres fournisseurs

---

<sup>23</sup>Web Service Constraint Language

<sup>24</sup>Web Service Recovery Language

de services affectés par l'exception); si ce n'est pas possible, (2) resélectionner un remplacement à partir d'autres fournisseurs de services atomiques qui répondent à la spécification des services et aux nouvelles exigences, sinon (3) replanifier l'ensemble de la composition des services.

### 1.4.3 Synthèse

L'étude de l'existant nous permet de conclure que la planification des actions dans le cadre des applications distribuées se caractérise par des scénarios de mobilité figés. En effet, un seul mécanisme est utilisé quelque soit le contexte d'exécution de ces applications.

En outre, et bien que les résultats des projets, dans le cadre des services Web, soient prometteurs, ils n'ont pas envisagé le cas où le processus d'orchestration constitue la source du problème. Dans une telle situation, le processus d'orchestration sera relancé, ce qui implique la ré invocation de tous les services partenaires. Une telle solution semble être coûteuse provoquant un retard considérable de la réponse.

C'est pour ces raisons que nous avons proposé une approche de planification qui comporte plusieurs politiques d'auto-adaptabilité des processus d'orchestration dépendant des différents contextes de leur exécution. En outre, notre approche recourt à maintes mécanismes de checkpoint et de mobilité afin de résoudre le cas où les processus d'orchestration constituent la source du problème, et éviter ainsi la ré exécution, généralement assez coûteuse, de ces processus.

## 1.5 Conclusion

Dans le domaine des systèmes distribués, il est intéressant de s'attarder sur les nouvelles orientations. Parmi ces technologies nous avons présenté différents concepts en relation avec les services Web. En outre, une étude détaillée de l'existant a été réalisée dans ce chapitre. Les deux classes de solutions précitées ne résolvent pas adéquatement la problématique mentionnée. Dans ce cadre, nous proposons une approche remédiant aux problèmes posés par ces solutions. Notre solution assure l'adaptation des processus d'orchestration aux changements fréquents du contexte de leur exécution. L'approche ainsi proposée fait l'objet du deuxième chapitre.

# Chapitre 2

## Démarche d'évaluation des actions de reconfiguration

### 2.1 Introduction

Notre travail s'inscrit dans le cadre de la gestion d'applications distribuées adaptatives. Il s'intéresse particulièrement à l'opération de reconfiguration des processus d'orchestration en réponse à la prédiction, à la détection d'une panne ou à la dégradation de la QoS.

Notre contribution dans ce projet consiste à définir et à valider un ensemble de politiques d'auto-adaptabilité des processus d'orchestration aux changements dynamiques du contexte de leur exécution.

Dans ce chapitre, nous présentons d'une manière détaillée notre solution en décrivant une démarche d'évaluation de politiques de reconfiguration opérant dans la phase de planification du cycle d'auto-adaptation relatif aux processus d'orchestration.

### 2.2 Principe de notre démarche

Notre démarche vise à définir et à valider un ensemble de politiques de checkpoint et de mobilité opérant dans la phase de planification, ceci en tenant compte de différents contextes d'exécution des processus d'orchestration. Pour ce faire, nous commençons par définir les actions de checkpoint et de mobilité possibles ainsi que les paramètres décrivant le contexte d'exécution. Ceux-ci dépendent de la structure du processus d'orchestration, de la mobilité, des caractéristiques de l'environnement d'exécution et de la nature du problème. Ensuite, nous procédons à faire correspondre ces actions et ces paramètres à travers la réalisation de plusieurs expérimentations sous différentes conditions. Par conséquent et en se basant sur les

observations de ces évaluations, nous définissons des politiques de mobilité et de checkpoint composées d'un ensemble de règles permettant de décider à propos des actions de reconfiguration appropriées en cas de problème. Ces décisions concernent la méthode et la fréquence de checkpoint nécessaires ainsi que le scénario de mobilité adéquat.

## 2.3 Définition des actions d'adaptation

Afin d'élaborer des politiques de reconfiguration des services Web orchestrés, nous présentons dans ce qui suit un ensemble de mécanismes de checkpoint et de mobilité.

### 2.3.1 Principe de la mobilité

Notre objectif étant de définir et de valider des politiques de checkpoint et de mobilité, il convient alors de s'attarder sur ces deux termes importants.

En effet, la mobilité forte des processus d'orchestration consiste à arrêter toute (ou un sous-ensemble de) l'exécution de ses instances, à les migrer vers un autre nœud, et à reprendre leur exécution à partir du dernier point de reprise (checkpoint). Le principe de cette solution consiste à ajouter à un processus d'orchestration des positions de checkpoint où l'état d'avancement de son exécution est sauvegardé afin d'être probablement utilisé dans le futur. Il est possible de varier les instants et la méthode du checkpoint. Aussi, plusieurs variétés de la mobilité sont envisagées. Dans la figure 2.1, nous présentons une de ces variétés qui consiste à attendre le prochain checkpoint avant de migrer, puis le processus poursuit son exécution à partir de ce dernier checkpoint.

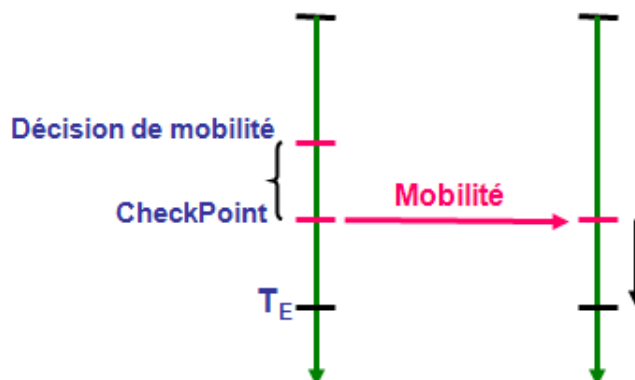


FIG. 2.1 – Illustration graphique du principe de la mobilité

Il existe d'autres scénarios de mobilité, tels que la ré exécution de tout le code du processus dans le noeud destinataire quelque soit la position de la décision de migration. En plus, la mobilité peut consister à la migration instantanée puis le retour au dernier checkpoint (rollback). Il est aussi possible de faire un checkpoint instantané juste avant de migrer et de poursuivre l'exécution à partir de celui-ci.

### 2.3.2 Actions de checkpoint

Le checkpointing est considéré comme étant une technique qui consiste à sauvegarder dans un support de stockage stable l'état d'un processus durant son exécution. En cas de panne, le processus est redémarré depuis son dernier checkpoint.

Dans ce contexte, plusieurs actions de checkpoint sont envisagées. En effet, nous pouvons varier les instants et les méthodes de checkpoint. En premier lieu, il se peut que l'action de reconfiguration ne recoure à aucun checkpoint. En second lieu, nous pouvons définir un mécanisme de checkpoint comme étant périodique, dans ce cas les différents checkpoints sont réalisés d'une façon périodique. Comme les positions de ces checkpoints sont figées et préalablement déterminées, nous exposons en troisième lieu un autre type de checkpoint que nous notons un checkpoint adaptatif. Ce mécanisme offre une meilleure flexibilité en particulier au niveau du choix à propos de l'instant de checkpoint. En dernier lieu, nous pouvons recourir à réaliser un checkpoint instantané, ceci dépend du besoin de migrer à un instant donné en supposant qu'il est possible d'ajouter un checkpoint à cet instant de mobilité. Il est à noter que les checkpoints peuvent avoir lieu aux barrières naturelles d'un processus d'orchestration (c'est-à-dire au début ou à la fin d'un bloc « Flow », ou encore dans un bloc élémentaire au début d'une structure séquentielle), sinon les checkpoints seront qualifiés de forcés.

### 2.3.3 Actions de mobilité

Afin de mettre en évidence la mobilité des processus d'orchestration, nous définissons un ensemble de scénarios correspondants qui peuvent être appliqués à une ou plusieurs instances à migrer selon la gravité du problème. Dans ce cas, il faut déterminer le nombre des instances à migrer et les choisir d'une façon pertinente.

#### 2.3.3.1 Premier scénario de mobilité S1 : Mobilité de code

Ce scénario consiste à la mobilité de code d'un processus d'orchestration quelconque. En d'autres termes, en cas de problème, tout ce code sera ré exécuté quelque soit la position de la décision de mobilité. En effet, nous n'envisageons pas de checkpoint dans ce cas de scénario.

### **2.3.3.2 Deuxième scénario de mobilité S2 : Rollback**

Le principe de ce scénario se résume dans la mobilité immédiate suivie d'un rollback (retour en arrière) au niveau du dernier checkpoint réalisé. Dans ce cas, il y aura ré invocation de tous les services Web partenaires. Pour mettre en évidence ce scénario, nous pouvons avoir recours au mécanisme de checkpoint périodique ou à celui adaptatif. C'est pourquoi le choix des positions de checkpoint doit être pertinent pour éviter un long rollback.

### **2.3.3.3 Troisième scénario de mobilité S3 : Migration au prochain checkpoint**

Dans ce scénario, il y aura une attente jusqu'au prochain checkpoint puis une migration vers le nouvel hôte. De même que le scénario précédent, nous pouvons utiliser soit le mécanisme de checkpoint périodique soit celui adaptatif.

### **2.3.3.4 Quatrième scénario de mobilité S4 : Checkpoint avant migration**

Ce dernier scénario se caractérise par la réalisation d'un checkpoint immédiat suivie d'une migration vers le nœud destinataire. Dans ce cas, les checkpoints sont envisagés instantanément. Ce scénario de mobilité ne comporte ni un surcoût de rollback ni un surcoût correspondant à celui de checkpoint périodique.

Nous illustrons dans la figure 2.2 les différentes actions de mobilité que nous venons de décrire.

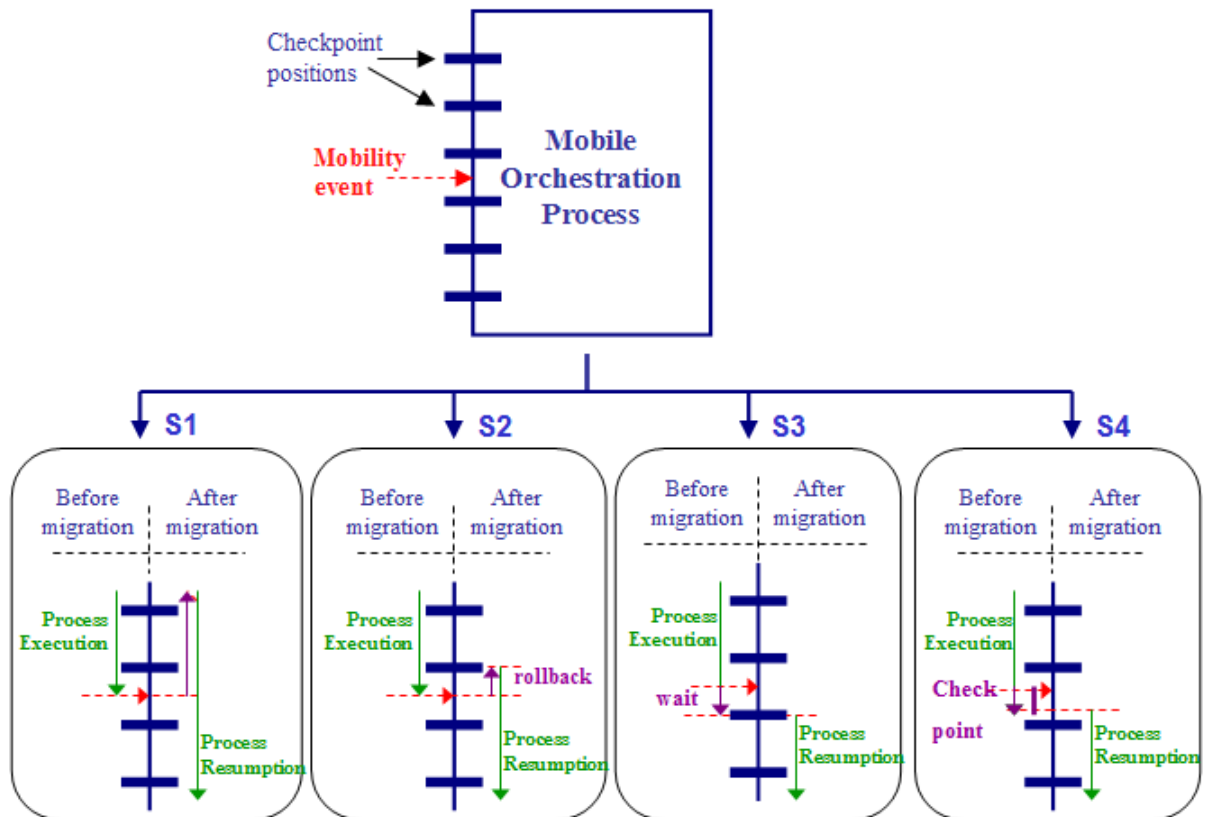


FIG. 2.2 – Illustration graphique des scénarios de mobilité

### 2.3.4 Synthèse

Dans le tableau 2.1, nous marions les quatre scénarios de mobilité précédemment décrits avec les différentes actions de checkpoint. En plus, nous présentons les résultats de la comparaison de ces scénarios de mobilité et nous mettons l'accent sur leurs forces et faiblesses.

TAB. 2.1 – Comparaison des scénarios de mobilité

| Identifiant du scénario | Description du scénario   | Nature du checkpoint       | Point(s) faible(s)   | Point(s) fort(s)   |
|-------------------------|---|----------------------------|--|--|
| <b>S1</b>               | Mobilité de code (ré exécution du code)   | Pas de checkpoint          | <ul style="list-style-type: none"> <li>– Perte de temps pour les processus longs</li> <li>– Probabilité faible d'arriver à la fin d'exécution</li> </ul> | Pas de perte de temps au cas où la durée du checkpoint est supérieure à la taille de l'exécution                                     |
| <b>S2</b>               | Mobilité immédiate et rollback (ré invocation de tous les services Web partenaires) | Périodique<br>Ou Adaptatif | Le choix des positions du checkpoint doit être pertinent pour éviter un long rollback  | Cas de détection de panne  |
| <b>S3</b>               | Attente jusqu'au prochain checkpoint puis migration                                 | Périodique<br>Ou Adaptatif | Cas de détection de panne  | Pas de rollback  |
| <b>S4</b>               | checkpoint immédiat et migration sans rollback                                      | Instantané                 | Cas de détection de panne  | <ul style="list-style-type: none"> <li>– Pas de rollback</li> <li>– Pas de surcoût correspondant au checkpoint périodique</li> </ul> |

## 2.4 Définition des paramètres décrivant le contexte d'exécution

Les paramètres, menant à une bonne prise de décision à propos de l'action d'auto-adaptation à appliquer, dépendent du processus d'orchestration (de l'application), de l'environnement de l'exécution et de la nature du problème. En outre, il convient de préciser qu'une partie de ces paramètres est statique, donc peut être déterminée au moment de déploiement du processus d'orchestration, tandis que l'autre partie est caractérisée par sa dynamique et varie ainsi au cours de l'exécution du processus en question. Dans ce qui suit, nous allons présenter l'ensemble de ces paramètres.

### 2.4.1 Paramètres relatifs au processus d'orchestration

Concernant ce type de paramètres, nous définissons ceux caractérisant la structure du processus d'orchestration, tels que le nombre de séquences « NbSeq », le nombre de structures « flow » noté « NbFlow », le nombre de branches en parallèle « NbBranch » et la taille des variables du processus d'orchestration. En outre, nous distinguons des paramètres caractérisant la mobilité du processus, en l'occurrence le temps nécessaire pour réaliser un checkpoint « Tc », le temps écoulé depuis le dernier checkpoint « TSLC », le temps entre deux checkpoints consécutifs « TE2C » ou encore noté intervalle de checkpoint I, le temps restant pour le prochain checkpoint « TRPC » et la latence de mobilité ML. Enfin, nous nous intéressons à récupérer le temps total d'exécution du processus sans checkpoint « TE », le temps d'exécution restant « TER », le nombre de migration(s) subie(s) et le nombre d'instances en cours.

### 2.4.2 Paramètres relatifs à l'environnement d'exécution

Nous caractérisons l'environnement d'exécution par deux métriques qui sont la MTBF et son écart type. En effet, nous supposons dans ce qui suit qu'un environnement est stable si l'écart type est petit par rapport à la MTBF supposée constante, non stable dans le cas contraire. Pour plus de clarification, nous définissons le coefficient de variation CV comme étant le rapport de l'écart type sur la MTBF. Ainsi, nous supposons qu'un environnement d'exécution est dit de niveau de stabilité 1 (stable) si  $CV < \text{seuil}$  et de niveau de stabilité 2 (non stable) si  $CV > \text{seuil}$ . Nous allons définir dans une section ultérieure ce seuil de stabilité.

### 2.4.3 Paramètres relatifs à la nature du problème rencontré

Le problème pouvant être rencontré peut s'agir de la détection ou de la prédiction d'une panne. Nous définissons dans le cadre de cette dernière variété deux niveaux de gravité du problème. En effet, nous supposons qu'un problème est grave dans le cas où la phase d'analyse nous informe qu'il y a une panne qui va se produire très prochainement, ou bien si la QdS offerte est inférieure à celle requise et ne cesse de se dégrader au cours du temps. Cependant, un problème est dit non grave lorsque la QdS offerte est inférieure à celle requise mais elle est constante au cours de l'exécution. Ces deux types de problèmes peuvent être localisés au niveau du noeud, du réseau ou de l'application elle même.

### 2.4.4 Synthèse

Nous récapitulons l'étude de l'ensemble des paramètres précédents dans le tableau 2.2 en les classifiant d'une façon plus claire :

TAB. 2.2 – Table de l'ensemble de paramètres décrivant le contexte d'exécution

| Classification                   |                               | Paramètres                                   | Statique / Dynamique | Moment de prise de décision           |
|----------------------------------|-------------------------------|--|----------------------|---------------------------------------|
| <b>Processus d'orchestration</b> | <b>Structure du processus</b> | Nombre de séquences "Nb-Seq"                 | Statique             | Au moment de déploiement du processus |
|                                  |                               | Nombre de flow "NbFlow"                      | Statique             | Au moment de déploiement du processus |
|                                  |                               | Nombre de branches en parallèle "Nb-Branche" | Statique             | Au moment de déploiement du processus |
|                                  |                               | Taille des variables                         | Statique             | Au moment de déploiement du processus |
|                                  |                               | Temps de checkpoint "Tc"                     | Statique             | Au moment de déploiement du processus |

| Classification |               | Paramètres                                       | Statique / Dynamique                         | Moment de prise de décision          |                                       |
|----------------|---------------|--|--|--------------------------------------|---------------------------------------|
|                | Mobilité      | Temps Depuis le Dernier checkpoint "TSLC"        | Dynamique                                    | Au cours de l'exécution du processus |                                       |
|                |               | Temps Entre Deux checkpoints consécutifs "TE2C"  | Dynamique                                    | Au cours de l'exécution du processus |                                       |
|                |               | Temps Restant pour le Prochain checkpoint "TRPC" | Dynamique                                    | Au cours de l'exécution du processus |                                       |
|                |               | Latence de mobilité "ML"                         | Dynamique                                    | Au cours de l'exécution du processus |                                       |
|                |               |  | Temps total d'exécution sans checkpoint "TE" | Statique                             | Au moment de déploiement du processus |
|                |               |  | Temps d'exécution restant "TER"              | Dynamique                            | Au cours de l'exécution du processus  |
|                |               |  | Nombre de migration(s) subie(s)              | Dynamique                            | Au cours de l'exécution du processus  |
|                |               |  | Nombre d'instances en cours                  | Dynamique                            | Au cours de l'exécution du processus  |
|                | Environnement |  | Temps moyen entre deux pannes "MTBF"         | Dynamique                            | Au cours de l'exécution du processus  |
|                |               |  | Ecart type entre deux pannes                 | Dynamique                            | Au cours de l'exécution du processus  |
| Problème       |               | Application                                      | Dynamique                                    | Au cours de l'exécution du processus |                                       |

| Classification |                            | Paramètres | Statique / Dynamique | Moment de prise de décision          |
|----------------|----------------------------|------------|----------------------|--------------------------------------|
|                | <b>Violation de la QdS</b> | Nœud       | Dynamique            | Au cours de l'exécution du processus |
|                |                            | Réseau     | Dynamique            | Au cours de l'exécution du processus |
|                | <b>Panne</b>               | Prédiction | Dynamique            | Au cours de l'exécution du processus |
|                |                            | Détection  | Dynamique            | Au cours de l'exécution du processus |

## 2.5 Formulation du problème

Selon le contexte d'exécution, il faut prendre les bonnes décisions en ce qui concerne l'instant de mobilité, les instances à migrer (il faut déterminer leur nombre et comment les choisir), la nature du checkpoint, la nature de mobilité et si la décision est faite avant ou au cours de l'exécution.

Pour ce faire, nous devons répondre à un ensemble de questions déterminantes du bon choix du plan d'actions à réaliser : Comment devrions-nous faire les checkpoints, en d'autres termes quel est le mécanisme de checkpoint le plus adéquat dans ce cas ? Quel scénario de migration devrions-nous adopter, c'est-à-dire devrions nous migrer instantanément ou bien attendre le prochain checkpoint ? Quand est ce que nous pourrions ajouter un checkpoint avant de migrer ? Quel est le nombre d'instances à migrer et comment les choisir ?

Afin de répondre à ces questions, nous nous trouvons face à une autre classification des paramètres de décision présentés dans la section précédente. En effet, chaque action lui correspond un ensemble de paramètres déterminants. Ainsi, le tableau 2.3 illustre la correspondance entre les actions de checkpoint et de mobilité d'une part et les paramètres décrivant le contexte d'exécution d'autre part.

TAB. 2.3 – Formulation du problème

| Questions  | Réponses à fournir  | Paramètres intervenants  |
|--|---|--|
| 1) Nature du checkpoint :<br>Comment faire les checkpoints ? | <b>Mécanismes de checkpoint :</b><br>1. sans checkpoint<br>2. périodique<br>3. adaptatif<br>4. instantané | MTBF et écart type correspondant, taille des variables, $T_c$ , TSLC, TE2C, TE, TER et ML. |
|  | Barrières naturelles Ou Forcé ?   | NbSeq, NbFlow et Nb-Branche.   |
| 2) Attendre ou migrer ?                                      | Scénario de migration (instantané ou j'attends ?)   | MTBF et écart type correspondant, TSLC, TE2C, TER et nature de la panne.                   |
|  | Ajouter un checkpoint avant la migration ou non ?   | Nature de la panne, TRPC, TSLC et TE2C.  |
| 3) Quelles sont les instances à migrer ?                     | Nombre qu'il faut migrer ?  | Nature de la panne et nombre d'instances en cours.   |
|  | Comment les choisir ?   | TSLC, TE2C, TER, nombre de migrations subies et ML.  |

A ce niveau, nous avons bien préparé le terrain pour pouvoir définir un ensemble de règles de reconfiguration des processus d'orchestration. Dans ce qui suit, nous allons décrire la démarche à suivre afin de construire le plan d'actions approprié à adopter pour chaque contexte d'exécution. Pour cela, nous allons répondre dans la section suivante aux questions précédemment posées en se basant sur une étude expérimentale.

## 2.6 Démarche d'évaluation de politiques d'auto-adaptabilité

Cette démarche vise à faire correspondre les paramètres de décision formant le contexte d'exécution d'un processus d'orchestration quelconque d'une part et le plan

d'actions qu'il faut adopter d'autre part.

Pour commencer, il convient de préciser que nous sommes aptes de décider à propos de certaines actions au moment de déploiement d'un processus d'orchestration. En effet, le planificateur peut déterminer d'une façon aisée les valeurs des paramètres statiques avant l'exécution du processus. Ces paramètres sont le « NbSeq », le « NbFlow », le « NbBranch », la taille des variables du processus, les « Tc », le « TE » et la « ML ». Une fois déterminés, nous procédons à varier quelques paramètres en fixant les autres pour déterminer le surcoût de cette variation.

## 2.6.1 Première question : Comment faire les checkpoints ?

Dans ce qui suit, nous allons mener une étude expérimentale afin de déterminer la fréquence et la méthode de checkpoint appropriées pour chaque contexte d'exécution des processus d'orchestration.

### 2.6.1.1 Quand utiliser le mécanisme sans checkpoint ?

D'une façon naturelle, nous pouvons confirmer que si la somme du temps de checkpoint  $T_c$  et celui de rétablissement  $TR_{\text{ét}}$  de l'état sauvegardé notée  $(T_c + TR_{\text{ét}})$  est supérieure ou égale au temps d'exécution  $TE$  d'un processus d'orchestration, alors il n'est pas nécessaire de faire un checkpoint. Cependant, ceci n'est pas suffisant. En effet, si en plus nous supposons que  $TE$  est supérieur à la MTBF, alors il se peut que plusieurs pannes se produisent sans que l'exécution du processus ne soit achevée comme illustré dans la figure 2.3.

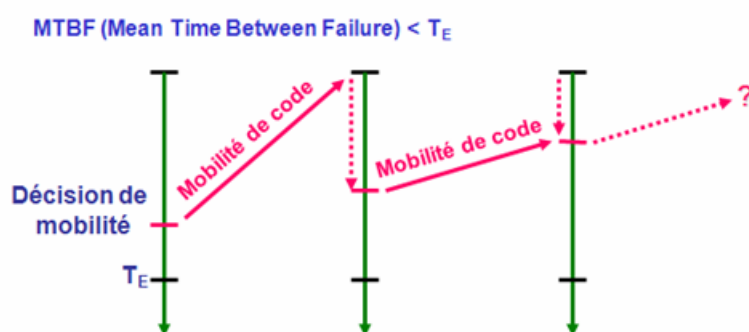


FIG. 2.3 – Exemple d'utilisation d'un mécanisme sans checkpoint

Ainsi, la probabilité de fin d'exécution du processus d'orchestration est assez faible, ceci en raison de la ré exécution de blocs du processus déjà réalisés plusieurs fois. D'où, il s'avère intéressant de faire des checkpoints afin d'atteindre le plutôt possible la fin d'exécution du processus.

Dans un autre cas, nous supposons que TE est inférieur à la MTBF, tels que Te est égal à environ 2 jours et la MTBF est égale à 3 jours. Encore une fois, ces conditions ne sont pas suffisantes pour se contenter de la valeur de la MTBF. Car il se peut qu'une panne se produit ce qui engendre une grande perte de temps d'exécution. Ainsi, il est important de faire des checkpoints dans ce cas aussi.

D'après ces exemples, nous pouvons confirmer que le mécanisme de checkpoint est intéressant dans le cas où la somme ( $T_c + TR_{\text{ét}}$ ) est supérieure ou égale à TE qui est inférieur à la MTBF.

### 2.6.1.2 Quand utiliser le mécanisme de checkpoint périodique ?

Dans ce qui suit, nous avons fixé les positions de checkpoint à quatre, deux parmi lesquels sont au sein d'une structure « flow » et les deux autres font partie d'une structure séquentielle. L'intervalle de checkpoint utilisé est environ 25% par rapport au temps d'exécution total. Ensuite, nous avons varié la MTBF d'une façon croissante. Le temps d'exécution du processus BPEL transformé et sans déclenchement de la mobilité est égal à environ 96 secondes. Dans la figure 2.4, nous avons détecté plusieurs pannes à différents niveaux par rapport aux différentes positions des checkpoints au sein du processus BPEL.

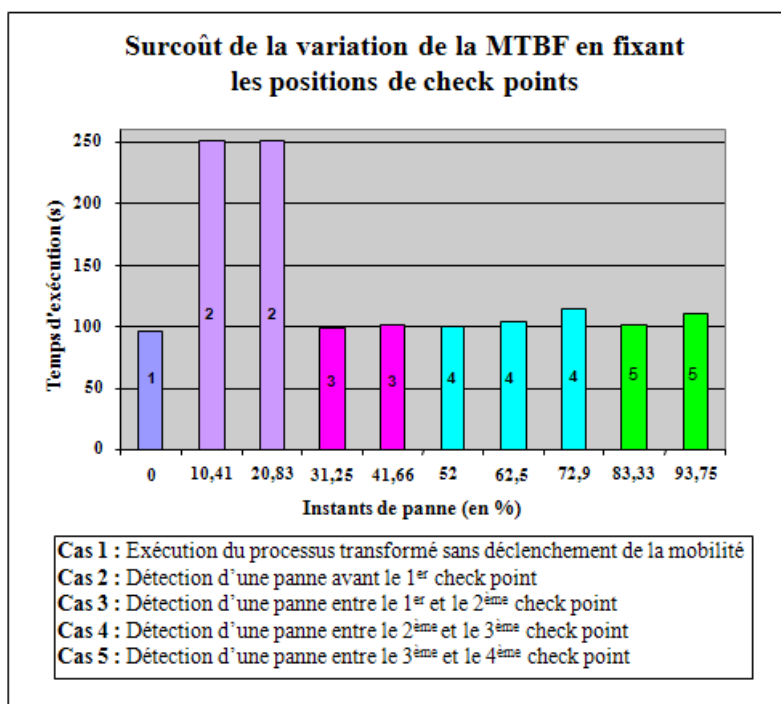


FIG. 2.4 – Surcoût de la variation de la MTBF en fixant les positions de checkpoint

La figure 2.4 témoigne de la variation du temps d'exécution du processus BPEL

pour chaque MTBF. En effet, pour le cas numéro 2, nous notons que la probabilité d'arriver à la fin de l'exécution est faible, ceci s'explique par le fait que la panne se produit toujours avant le premier checkpoint. Pour les cas 3, 4 et 5, nous supposons que le nombre de migration de l'instance BPEL est fixé à un. Dans ce cas, la mobilité se fait après un certain checkpoint. Nous remarquons pour chaque cas que plus la date de déclenchement de la mobilité s'éloigne du dernier checkpoint, plus le temps d'exécution du processus en question augmente, ceci est du à l'augmentation du temps de rollback dans le nœud destinataire. En d'autres termes, plus le temps écoulé depuis le dernier checkpoint TSLC dans le nœud initial est petit, plus le temps d'exécution total du processus BPEL est petit. Nous détaillerons ce point davantage en répondant à la troisième question.

Inversement à ce que nous venons d'expérimenter, nous fixons dans ce qui suit la MTBF à environ 15s pour un processus BPEL consommant en moyenne 40s sans réalisation de checkpoint. Puis, nous varions à chaque fois l'intervalle de checkpoint en variant les positions des points de reprise. En effet, nous fixons la première borne de cet intervalle à une position avant l'instant de mobilité. Après, nous varions à chaque fois la deuxième borne tout en se positionnant avant le déroulement de la migration tant que c'est possible. Il est à noter qu'une borne correspond à une action de checkpoint.

Nous présentons dans la figure 2.5 sur l'axe des abscisses le rapport entre les intervalles de checkpoint et la MTBF, nous mesurons ensuite pour chaque pourcentage le temps d'exécution du processus BPEL correspondant. La première valeur concerne le cas du processus initial sans checkpoint (ainsi il n'y a pas d'intervalle de checkpoint), quant à la dernière valeur, elle correspond au plus long intervalle de checkpoint possible avant l'instant de mobilité du processus BPEL.

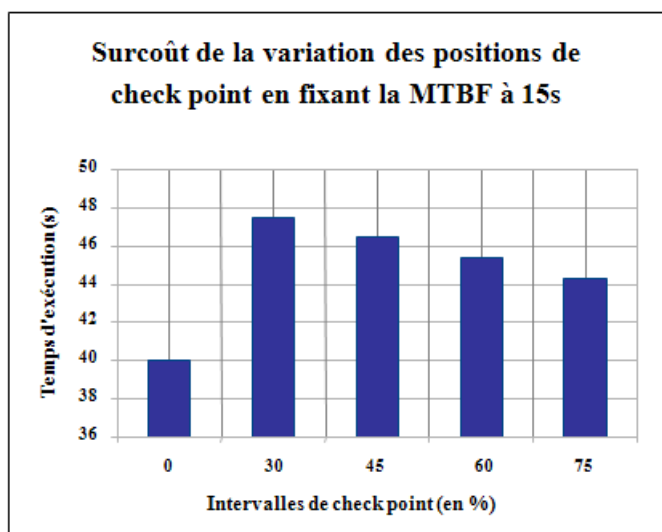


FIG. 2.5 – Surcoût de la variation des positions de checkpoint en fixant la MTBF

A partir de la figure 2.5, il est clair que plus l'intervalle de checkpoint est grand par rapport à la MTBF tout en fixant la première borne de cet intervalle, plus le temps d'exécution du processus BPEL est réduit. En effet, moins de blocs de code sont exécutés dans le noeud destinataire étant donné que leur état d'exécution est déjà sauvegardé dans la base de données distante.

Ces résultats nous donne l'idée d'approximer une valeur d'un intervalle de checkpoint acceptable équivalente à une période de checkpoint pour un processus d'orchestration. En effet, nous approximations l'instant de panne par la valeur de la MTBF plus au moins son écart type noté  $E$ . Ensuite, nous considérons le temps de checkpoint nécessaire avant la migration. Ainsi, nous élaborons la règle suivante notée (R) et déclarée comme suit :  $I = (1 - E) * MTBF - T_c$ , avec  $I$  est l'intervalle de checkpoint. Par conséquent, sachant à peu près l'instant de mobilité à partir de la MTBF et sachant aussi  $T_c$ , nous pouvons définir la longueur d'une période de checkpoint et fixer ainsi les positions correspondantes.

Dans une section précédente, nous avons fait allusion à un seuil de stabilité dépendant de  $CV$ . Pour cela, nous avons réalisé une expérimentation en fixant la MTBF et en variant l'écart type, en d'autres termes nous avons varié le  $CV$ . En effet, plus ce coefficient est grand, plus l'environnement est moins stable. Ensuite, nous avons calculé l'intervalle de checkpoint d'après la règle (R) afin de fixer les positions des points de reprise. Puis, nous avons mesuré pour chaque  $CV$  le temps d'exécution du processus BPEL correspondant. Les résultats sont illustrés dans la figure 2.6.

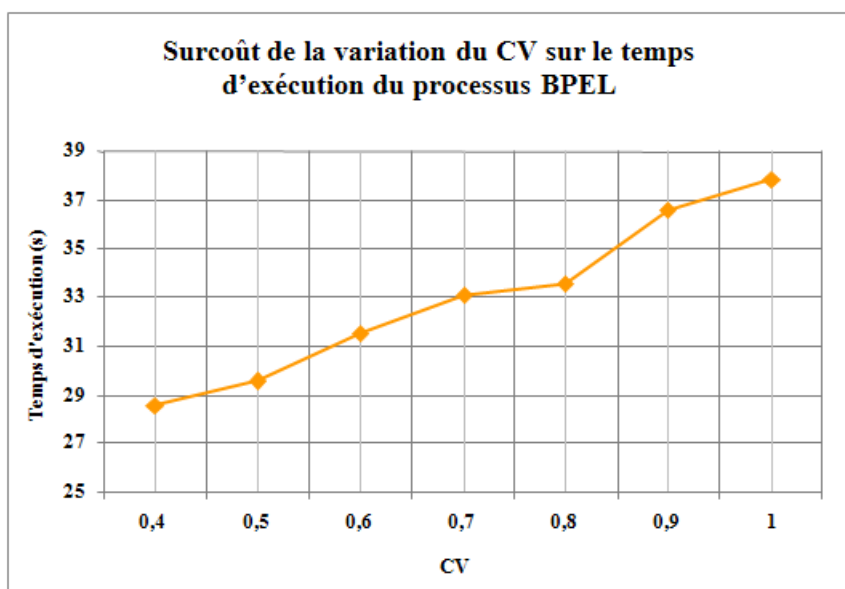


FIG. 2.6 – Surcoût de la variation du CV sur le temps d'exécution du processus BPEL

Nous remarquons d'après la figure 2.6 que plus le CV augmente, plus le temps d'exécution du processus BPEL augmente. Outre cela, nous notons que le surcoût de la variation du CV par rapport au temps d'exécution initial sans migration du processus ne dépasse pas en moyenne 5% pour des valeurs de CV inférieures à 0.8, tandis que ce surcoût est le plus important pour  $CV = 0.9$  et ne cesse de croître pour des CV plus grands, il est en fait égal à 12%. Ainsi, à partir d'un  $CV = 0.8$ , il n'est plus intéressant de choisir l'intervalle de checkpoint selon (R). Par conséquent, la limite de l'intérêt du mécanisme de checkpoint périodique est déterminée par la valeur de CV.

### 2.6.1.3 Quand utiliser le mécanisme de checkpoint adaptatif ?

D'après la figure 2.6, pour des CV supérieurs strictement à 0.8, l'environnement d'exécution tend vers une instabilité dont la période de checkpoint obtenue à partir de (R) devient incapable de conserver un surcoût de variation raisonnable et faisable par rapport au temps d'exécution initial.

De ce fait, il s'avère intéressant de recourir au mécanisme de checkpoint adaptatif en définissant un nouvel intervalle de checkpoint dépendant de la QdS. Effectivement, l'intervalle de checkpoint optimal a pour but de minimiser le temps d'exécution du processus d'orchestration en minimisant le nombre de checkpoint. Mais, si le contrat relatif à la QdS tolère un temps d'exécution plus large, le nombre de checkpoint peut augmenter afin de faire face à des pannes imprévues. Cette solution est intéressante

surtout dans le cas des environnements très dynamiques où la MTBF ne reflète plus le temps réel entre les pannes. En plus, l'utilisation d'un intervalle de checkpoint optimal peut ne pas assurer la QoS requise. Dans un tel cas, la prise en considération de la QoS pour déterminer cet intervalle nous permet de prédire le cas où le contrat de la QoS ne peut pas être satisfait, ainsi une action de reconfiguration doit se produire. La détermination de l'intervalle de checkpoint correspondant fait partie d'une perspective de ce travail à court terme.

#### 2.6.1.4 Quand utiliser le mécanisme de checkpoint aux barrières naturelles ?

Nous avons implémenté un processus d'orchestration BPEL dont la taille de départ de ses variables est fixée à 53 octets et le nombre de branches en parallèle au niveau d'une structure « flow » est fixé à deux. Maintenant, nous fixons tous les paramètres et nous varions le nombre de branches en parallèle d'une manière croissante. Le but est de déterminer l'impact de cette variation sur le « TC » au niveau d'une structure « flow ». La taille des variables du processus BPEL étant fixée à 53 octets. Dans la figure 2.7 suivante nous présentons le surcoût de la variation du nombre de branches en parallèle.

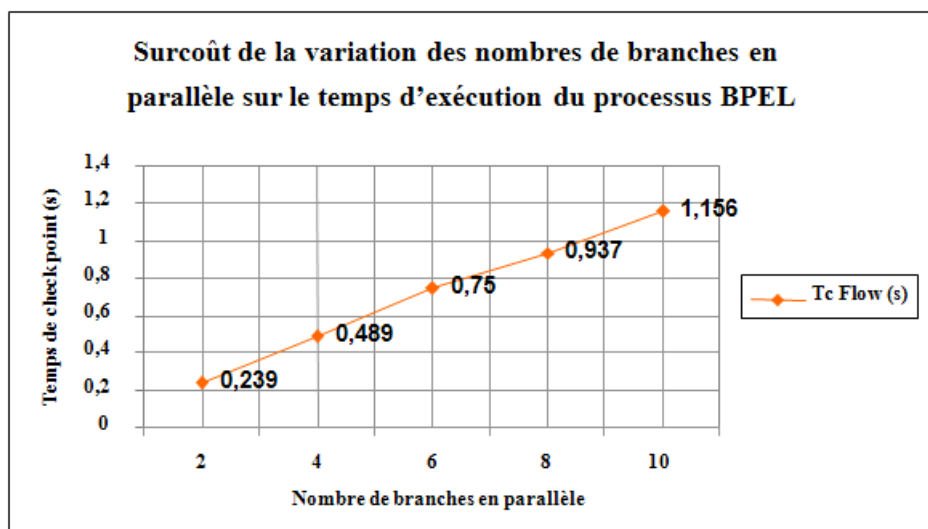


FIG. 2.7 – Surcoût de la variation du nombre de branches en parallèle du processus BPEL

D'après les évaluations précédentes, il est clair que plus le nombre de branches en parallèle est grand, plus le temps de checkpoint au niveau d'une structure « flow » est important. Ceci est dû à l'augmentation du temps de synchronisation de ces

branches. En effet, chaque ajout d'une branche en parallèle dans une structure « flow » coûte en moyenne 17% du  $T_c$ .

Maintenant, nous présumons que la taille de départ des variables de notre processus est 53 octets et que le nombre de branches en parallèle au niveau d'une structure « flow » est fixé à deux. Nous avons ensuite varié la taille de ces variables d'une façon croissante pour le même nombre de branches. Nous avons mesuré pour chaque variation le « TC » au niveau d'une structure « flow », le « TC » au niveau d'une structure séquentielle et le temps de rétablissement dans le nouvel hôte, celui-ci correspond au chargement du checkpoint et au rétablissement de l'instance BPEL. Les mesures de départ de ces temps en secondes sont respectivement : 0.239 ; 0.098 et 0.234.

La figure 2.8 illustre le surcoût de la variation de la taille des variables du processus BPEL en considérant deux branches en parallèle figurant dans une structure "flow" et en augmentant à chaque fois la taille des variables de 53 octets.

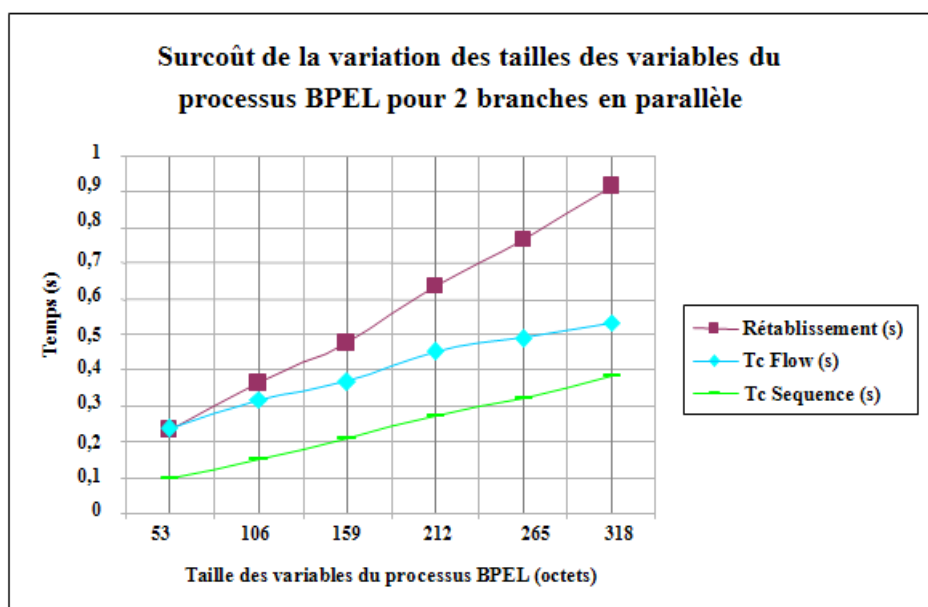


FIG. 2.8 – Surcoût de la variation de la taille des variables du processus BPEL pour 2 branches en parallèle

Nous remarquons que plus la taille des variables augmente, plus la valeur des temps précédemment cités augmente. Ceci s'explique par l'augmentation du nombre d'octets à écrire dans la base de données distante, d'où l'augmentation des temps de checkpoint  $T_c$  au niveau d'une structure « flow » et au niveau d'une structure séquentielle. De même la croissance du nombre d'octets à lire depuis la base de données fait que le temps de rétablissement croît.

D'après la figure 2.8, nous déduisons que chaque ajout de 53 octets coûte en moyenne 32%, 30% et 29% respectivement par rapport au temps de rétablissement, au Tc Flow et au Tc Sequence. Ainsi, le surcoût d'ajout d'un octet dans la taille des variables ne dépasse pas en moyenne 0.6%, 0.566% et 0,547% respectivement par rapport au temps de rétablissement, au Tc Flow et au Tc Sequence. Nous remarquons ainsi que les valeurs de ces surcoûts sont très proches. En effet, il s'agit à chaque fois de la lecture ou de l'écriture de la même taille des variables depuis la base de données distante. Nous notons aussi d'après la figure 2.8 que la valeur moyenne de la différence entre le temps de checkpoint Tc au niveau d'une structure « flow » et au niveau d'une structure séquentielle est 0.158s, cette moyenne correspond en fait au temps nécessaire pour l'ajout d'une branche et au temps de synchronisation des deux branches en parallèle. Or, nous avons déduit d'après l'expérimentation précédente que le surcoût moyen d'ajout d'une branche est 17% du Tc, ce qui correspond en moyenne à 0.057s. Ainsi, le temps de synchronisation de deux branches en parallèle est de l'ordre de 0.1s, ce qui présente en moyenne 25% du Tc au niveau d'une structure « flow ».

D'après les évaluations précédentes, nous pouvons déduire le temps de checkpoint Tc au sein d'une structure "flow". Nous notons dans ce qui suit TcNoc/NB : le temps de checkpoint de N octets dans N branches en parallèle, tel que N un entier strictement positif. En effet, le temps de checkpoint de N octets dans une branche noté TcNoc/1B est calculé comme suit :  $TcNoc/1B = Tc1oc/1B + Tc1oc/1B * 0.566 * 10^{-3} * Noc$  (1). En plus, nous pouvons calculer le temps de checkpoint de N octets dans 2 branches. En effet,  $TcNoc/2B = TcNoc/1B + TcNoc/1B * 0.25$  (2). Enfin,  $TcNoc/NB = TcNoc/2B + TcNoc/2B * 0.17 * (NB-2)$  (3).

A partir de ces équations, nous déduisons une équation générale de calcul du temps de checkpoint notée (R1) en remplaçant TcNoc/2B dans la dernière équation (3) par sa valeur déterminée dans (2). Ainsi, nous obtenons (R1) :  $Tc = TcNoc/1B + TcNoc/1B * 0.25 + [TcNoc/1B + TcNoc/1B * 0.25] * 0.17 * (NB-2)$ . Nous pouvons développer davantage cette équation en substituant la valeur de TcNoc/1B dans (2) par son équivalent d'après (1).

Dans la figure 2.9 , nous avons fixé un même bloc "flow" et nous avons varié le nombre de clients (instances) afin de calculer le surcoût de cette variation

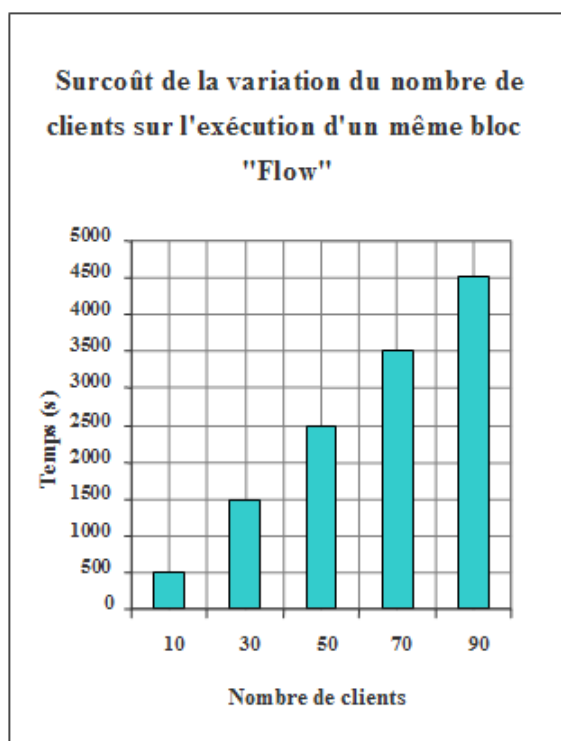


FIG. 2.9 – Surcoût de la variation du nombre de clients sur l'exécution d'un même bloc "Flow"

Nous remarquons d'après cette figure que l'ajout de 20 clients lui correspond en moyenne un surcoût de 2% du temps d'exécution du bloc "flow". Ainsi, le surcoût d'ajout d'un client s'évalue à 0.1% de ce temps. Nous définissons dans un processus d'orchestration la longueur d'exécution d'un bloc "flow" `longFlow`. Ainsi, nous décidons que si  $\text{Max}(\text{longFlow})$  multiplié par le nombre d'instances en cours et par le surcoût correspondant 0.1 est inférieur ou égal à l'intervalle de checkpoint déterminé dans les sections précédentes, alors il n'est pas nécessaire de faire un checkpoint dans ces blocs "flow". Dans le cas contraire, ceci est envisageable sachant que le  $T_c$  est égal à la valeur calculée d'après l'équation (R1).

### 2.6.2 Deuxième question : Attendre ou migrer ?

La réponse à cette question nous permet de déterminer comment choisir entre les scénarios de mobilité définis dans une section précédente. Cependant, il convient de préciser que le choix de l'action de reconfiguration dans ce cas est fortement lié à la nature de la panne (détection ou prédiction), au niveau de gravité du problème et aux sources de la violation de la QoS (application ou/et noeud ou/et réseau), nous supposons que les valeurs de ces paramètres sont établies dans la phase d'analyse

du processus d'auto-réparation.

Pour débiter, nous notons que le premier scénario S1 qui est la mobilité de code est équivalent à un scénario sans checkpoint. Dans ce cas, tout le processus sera ré exécuté après sa migration. Ainsi, l'action de mobilité S1 est utilisée dans les mêmes conditions de décision à propos du mécanisme sans checkpoint présenté dans la section précédente. Par conséquent, nous choisissons S1 si  $[TE < (TRét + Tc)]$  et  $[TE < MTBF]$ .

Nous passons dans ce qui suit à déterminer les conditions de décision à propos des scénarios de mobilité S2, S3 et S4. Il convient alors de préciser que dans le cas d'une détection de panne, il n'est pas possible de faire un checkpoint, ainsi l'action d'auto-adaptabilité consiste à migrer toutes les instances vers un noeud plus performant puis réaliser un rollback : il s'agit ainsi du scénario de mobilité S2.

Dans ce qui suit, nous nous positionnons dans le cas d'une prédiction de panne ou encore dans le cas de dégradation de QdS, nous supposons que le niveau de stabilité de l'environnement d'exécution est grave quelque soit la source de la panne, nous rappelons que c'est-à-dire il y a une panne proche ou bien la QdS est entrain de se dégrader au cours du temps. Dans un tel cas, le scénario adopté est S2. En effet, le niveau de stabilité précédemment décrit ne rend pas possible de réaliser un checkpoint instantané ou bien d'attendre le prochain checkpoint.

Toujours dans le cas de dégradation de QdS, nous supposons à présent que le niveau de stabilité de l'environnement d'exécution est non grave, c'est-à-dire il n'y a pas une panne proche et la QdS est constante au cours du temps tout en étant acceptable par rapport à celle requise.

Dans une première expérimentation, nous supposons que le problème vient du réseau et non pas du noeud (en d'autres termes le temps de communication entre les noeuds est assez important en raison du faible débit du réseau) et que la dégradation de QdS est non grave. Nous avons mesuré dans ces conditions le temps d'exécution du processus BPEL en variant les instants de mobilité par rapport au dernier checkpoint et en adoptant pour chaque instant un des trois scénarios de mobilité S2, S3 et S4. Les résultats de cette expérimentation sont présentés dans la figure 2.10.

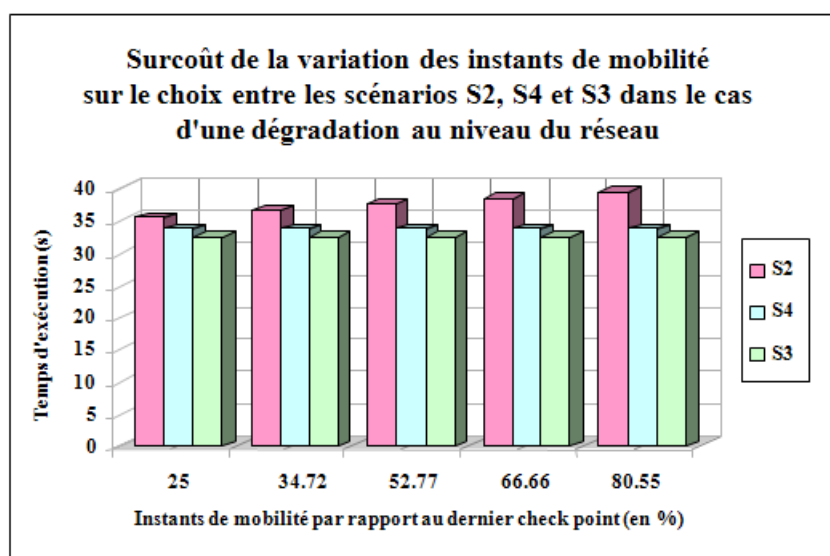


FIG. 2.10 – Cas d'une dégradation non grave au niveau du réseau

Cette expérimentation témoigne du fait qu'en adoptant le scénario S2, plus l'instant de mobilité est loin du dernier checkpoint, plus le temps d'exécution du processus BPEL est long, ceci est dû au temps de rollback long dans le noeud destinataire. En effet, plus l'intervalle de checkpoint augmente de 10%, plus le temps d'exécution total du processus BPEL augmente d'environ 2.86%. Quant aux valeurs obtenues pour le cas du scénario S4, nous remarquons que les temps d'exécution du processus d'orchestration sont presque les mêmes quelque soit l'instant de mobilité. En effet, ces mesures ne dépendent pas de cet instant puisque à chaque fois il y aura une action de checkpoint avant de migrer puis le processus va reprendre son exécution à partir de ce dernier checkpoint. La même remarque est notée pour le cas du scénario S3, quelque soit l'instant de mobilité, nous allons attendre le prochain checkpoint figurant dans le processus BPEL puis migrer. En comparant les résultats obtenus pour S3 et S4, nous remarquons que la différence entre les temps d'exécution pour chaque cas de ces scénarios correspond au surcoût de checkpoint ajouté suite à l'exécution du scénario S4, ce surcoût est de l'ordre de 4%. D'après ces évaluations, nous décidons de choisir S2 lorsque le temps d'exécution total TE est inférieur ou égal au temps de checkpoint Tc, car il n'est pas intéressant de faire un checkpoint dans ce cas. Sinon, nous adoptons pour le scénario S3 car il correspond à celui de moindre coût.

Maintenant, et dans les mêmes conditions que l'expérimentation précédente, nous nous intéressons au cas d'une dégradation uniquement au niveau du noeud. Dans ce cas, nous rejetons dès le départ l'utilisation du scénario S3 car il n'est pas possible d'attendre le prochain checkpoint. Nous présentons dans la figure 2.11 les résultats

de cette expérimentation.

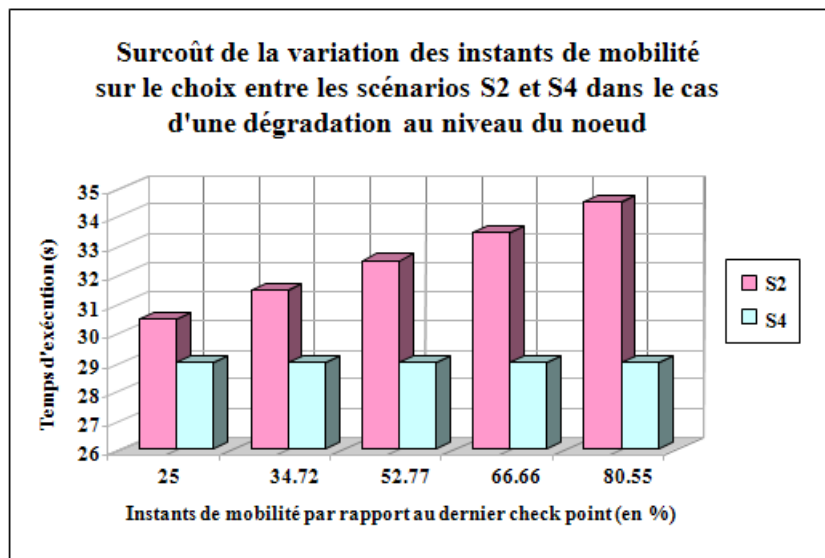


FIG. 2.11 – Cas d'une dégradation non grave au niveau du noeud

D'après la figure 2.11 et par équivalence à l'expérimentation précédente, il est clair qu'en adoptant le scénario S2, plus l'instant de mobilité est loin du dernier checkpoint, plus le temps d'exécution du processus BPEL est important. En plus, nous remarquons, en cas d'utilisation du scénario S4, que les mesures du temps d'exécution du processus d'orchestration sont presque les mêmes quelque soit l'instant de mobilité pour la même raison déjà citée dans l'évaluation précédente. D'après ces évaluations, nous décidons encore une fois de choisir S2 lorsque TE est inférieur à Tc. Sinon, nous choisissons le scénario S4 car il possède le moindre coût.

Enfin, nous procédons à une expérimentation dont la dégradation touche à la fois le réseau et le noeud de déploiement. Ainsi, le niveau de gravité de l'environnement d'exécution devient plus important. En étant dans les mêmes conditions que les évaluations précédentes, nous déduisons les mêmes résultats et les mêmes décisions que ceux relatifs à l'expérimentation précédente. La figure 2.12 témoigne de ces résultats pour les scénarios S2 et S4. Nous avons omis le scénario S3 pour la raison que le noeud est dégradé dans ce cas.

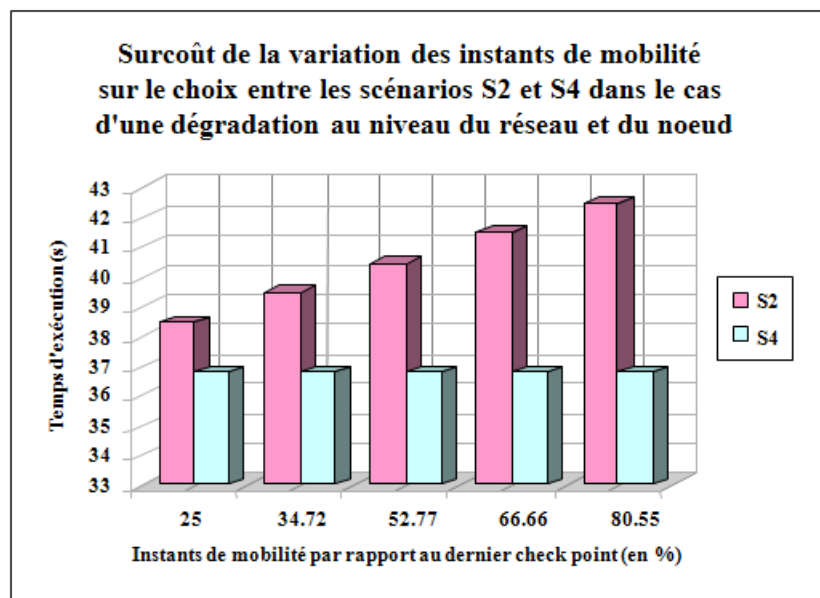


FIG. 2.12 – Cas d'une dégradation non grave au niveau du réseau et du noeud

D'après les évaluations réalisées, nous attribuons dans ce qui suit des poids aux scénarios de mobilité S2, S3 et S4. En effet, S2 possède le poids le plus lourd, S4 lui correspond un poids moyen et S3 a le plus faible. De ce fait, en cas de combinaison de problèmes de différentes sources (application, noeud, réseau), le scénario de mobilité à adopter correspond à celui dont le poids est le plus lourd. Par exemple, dans le cas où  $TE$  est inférieur à  $Tc$ , si nous supposons qu'il y a une dégradation non grave au niveau du noeud, donc le scénario approprié est S4. Maintenant, nous présumons qu'il y a une dégradation non grave au niveau du réseau, ainsi le scénario choisi est S3. Si en cas de présence de ces deux types de dégradation à la fois, alors l'action de mobilité à adopter est S4. En effet, le poids de S4 est supérieur à celui de S3. Ceci s'explique par le fait qu'il y a plus de contraintes au niveau de la gravité du problème posé.

Toujours dans le cas de dégradation de QoS non grave, nous présumons dans ce qui suit qu'il y a un problème de scalabilité lié à l'application et donc celle-ci ne peut plus répondre d'une façon faisable à toutes les requêtes clientes. Dans ce cas, l'action de mobilité à adopter est S2 si  $TE$  est inférieur ou égal à  $Tc$  et S4 sinon. En effet, le choix du scénario S3 augmente davantage la charge du noeud hébergeant les instances en cours en augmentant le temps d'attente des prochains checkpoints. Ainsi, il est intéressant de migrer une partie de ces instances vers un autre noeud afin de reprendre leur exécution. Le choix du nombre et des instances à migrer sera mieux expliqué en répondant à la troisième question dans la section suivante.

### 2.6.3 Troisième question : Quelles sont les instances à migrer ?

La réponse à cette question nous permet de déterminer quel est le nombre d'instances à migrer vers un noeud plus performant et comment les choisir.

#### 2.6.3.1 Quel est le nombre d'instances qu'il faut migrer ?

Pour commencer, il convient de préciser que s'il s'agit d'une détection de la panne, alors l'action à prendre est évidemment la migration de toutes les instances, ce qui correspond au choix du scénario de mobilité S2. Maintenant, nous nous positionnons dans le cas d'une prédiction de panne et nous allons essayer d'expliquer comment décider à propos du nombre d'instances à migrer. En effet, nous présumons que nous disposons d'une courbe de références fournie par la phase d'analyse du processus d'auto-réparation comme illustré dans la figure 2.13.

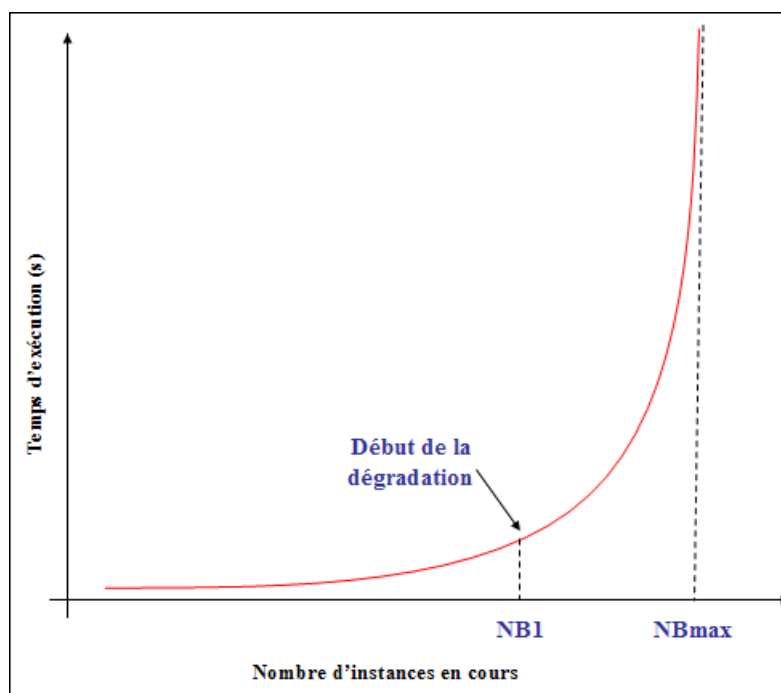


FIG. 2.13 – Variation du nombre d'instances en fonction du temps d'exécution

En effet, cette courbe mesure la variation du nombre d'instances en cours de traitement en fonction du temps de leur exécution. Nous déduisons à partir la figure 2.13 que plus le nombre d'instances augmente, plus ce temps croît. Ceci est envisageable comme la charge du noeud correspondant devient de plus en plus importante. En outre, nous remarquons que la forme de la courbe de références est équivalente à celle d'une fonction exponentielle et qu'à partir d'un certain nombre

d'instances noté "NB1", le temps d'exécution croît rapidement. En fait, nous supposons que NB1 est la dernière bonne valeur avant l'occurrence d'une dégradation. En d'autres termes, NB1 représente le nombre d'instances limite avant la croissance importante du temps d'exécution. En plus, la figure 2.13 montre que la dernière valeur mesurée notée "NBmax" correspond au plus grand nombre d'instances pouvant être traitées simultanément après la dégradation. Ainsi, le nombre d'instances à migrer afin d'équilibrer la charge du noeud noté "NBM" est égal la différence entre NBmax et NB1. Ces deux dernières valeurs étant fournies par la phase d'analyse.

### 2.6.3.2 Comment choisir les instances à migrer ?

Maintenant, nous allons réaliser deux expérimentations pour essayer d'établir une équation de choix des instances à migrer. La première consiste à varier le TSLC et à voir son impact sur le temps d'exécution total du processus BPEL. Nous illustrons les résultats en question dans la figure 2.14.

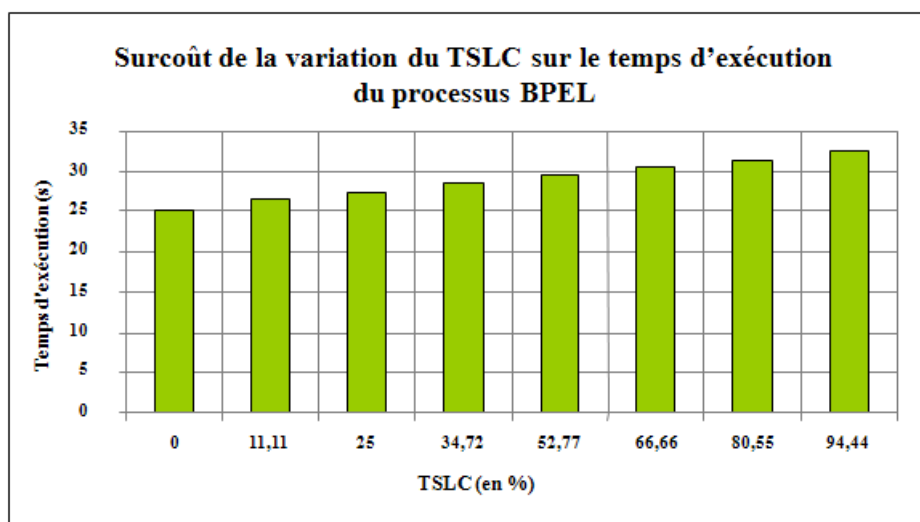


FIG. 2.14 – Surcoût de la variation du TSLC sur le temps d'exécution du processus BPEL

La première valeur mesurée dans la figure 2.14 correspond au temps d'exécution du processus BPEL sans migration. Nous remarquons que plus le TSLC est grand, plus le temps d'exécution est grand. En effet, le temps de rollback dans le noeud destinataire est aussi plus important. Nous déduisons que chaque augmentation du TSLC de 7% lui correspond une augmentation de 2% au niveau du temps d'exécution du processus BPEL.

Nous passons maintenant à la deuxième expérimentation qui consiste à varier le nombre de migrations par instance et à mesurer le temps d'exécution observé à

chaque fois. Les résultats de cette variation font l'objet de la figure 2.15.

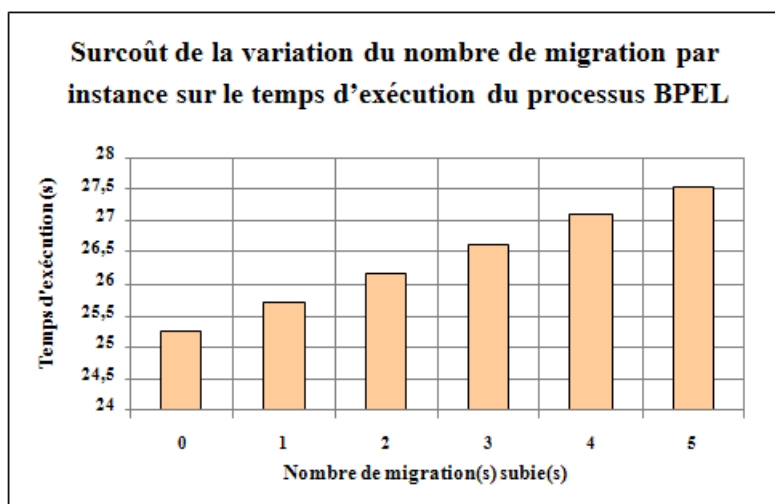


FIG. 2.15 – Surcoût de la variation du nombre de migration par instance sur le temps d'exécution du processus BPEL

D'après la figure 2.15, nous remarquons que plus le nombre de migrations subies par une instance croît, plus le temps d'exécution augmente. En réalité, chaque migration lui correspond l'ajout d'un surcoût de migration, de rétablissement et de communication entre les différents noeuds. Ce surcoût correspond à environ 2%.

En se basant sur les deux expérimentations réalisées, nous rappelons que l'augmentation du TSLC de 1% coûte 2/7% et que chaque migration coûte 2%. Nous pouvons alors déduire une équation de calcul de surcoût total notée (R2) :  $CS2 + (NBMig * 2) + CS4$ . En effet, étant donné que le choix entre les actions de mobilité pour chaque instance à migrer se fait en répondant à la deuxième question posée, CS2 est le coefficient de surcoût de l'utilisation du scénario S2, il est égal à  $(2/7 * TSLC)$ . NBMig correspond au nombre de migrations subies pour l'instance en cours. Quant à CS4, il s'agit d'un coefficient de surcoût de l'utilisation du scénario S4 qui correspond au surcoût d'ajout d'un checkpoint dans ce cas. Ces coefficients sont omis en cas d'absence de l'utilisation des scénarios correspondants. Ainsi, les instances ayant le surcoût minimum calculé selon (R2) seront choisies. Nous rappelons qu'il ne faut pas dépasser le nombre NBM déterminé dans la section précédente.

#### 2.6.4 Synthèse : Politiques d'auto-adaptabilité des processus d'orchestration

Dans ce qui suit, nous allons récapituler l'ensemble des politiques d'auto-adaptabilité des processus d'orchestration en exposant les différentes règles établies. En fait, nous

distinguons une politique de checkpoint et une autre de mobilité.

La première politique est le fruit des évaluations réalisées en répondant à la première question portant sur la nature de checkpoint. L'ensemble de règles correspondantes est présenté dans le tableau 2.4 :

TAB. 2.4 – Politique de checkpoint

| Actions de checkpoint | Conditions  |
|-----------------------|---|
| Sans checkpoint       | $[TE < (TR_{\text{ét}} + T_c)]$ et $[TE < MTBF]$              |
| Checkpoint Périodique | $CV \leq 0.8$ ET $I = (1 - \text{écart type}) * MTBF - T_c$   |
| Checkpoint Adaptatif  | $CV > 0.8$ ET $I = f(\text{paramètres de l'environnement})$   |
| Barrières Naturelles  | $\text{NbClients} * 0.1 * \text{Max}(\text{longFlow}) \leq I$ |

Quant à la deuxième politique, elle englobe les décisions à propos du choix du scénario de migration le plus adéquat et des instances à migrer. Cette politique est le résultat de la réponse aux deux dernières questions. Le tableau 2.5 illustre les actions adéquates pour déterminer le scénario de mobilité approprié :

TAB. 2.5 – Politique de mobilité par instance

|            | Panne | Dégradation QdS |      |        |             |      |        |
|------------|-------|-----------------|------|--------|-------------|------|--------|
|            |       | Non grave       |      |        | grave       |      |        |
|            |       | Application     | Nœud | Réseau | Application | Nœud | Réseau |
| $TE < T_c$ | S2    | S2              | S2   | S2     | S2          | S2   | S2     |
| $TE > T_c$ | S2    | S4              | S4   | S3     | S2          | S2   | S2     |

Il convient aussi de rappeler qu'en cas de combinaison des problèmes de dégradation de QdS, le scénario de mobilité approprié correspond à celui de poids le plus fort.

Enfin, en cas de problème au niveau de l'application (que ce soit grave ou non grave), le nombre d'instances à migrer, noté NBM précédemment, est la différence entre la dernière valeur mesurée du temps d'exécution du processus (NBmax) et la dernière bonne valeur correspondante juste avant la dégradation de ce temps (NB1). Nous supposons que ces deux dernières valeurs sont fournies par la phase d'analyse. En plus, les instances à migrer correspondent à celles dont le surcoût est le minimum. Celui-ci étant calculé selon l'équation  $(R1) = (2/7 * TSLC) + (NBMig * 2) + CS4$  où CS4 est le surcoût d'ajout d'un checkpoint en cas d'utilisation du scénario de mobilité S4.

## 2.7 Conclusion

Dans ce chapitre, nous avons fourni une description détaillée de l'approche que nous avons proposée. Notre solution permet en premier lieu de générer le meilleur plan d'actions nécessaire pour ajuster le comportement des processus d'orchestration en identifiant le mécanisme d'adaptation le plus adéquat. En plus, elle garantit l'amélioration des performances globales de ce type de processus. Durant notre étude, nous remédions ainsi aux insuffisances présentées par les différentes solutions mentionnées dans la littérature.

Dans le chapitre suivant, nous allons détailler davantage le principe de la mobilité forte employée pour assurer l'auto-adaptabilité des services Web orchestrés en tenant compte de leur état d'exécution. Outre cela, nous allons automatiser notre approche. D'une part, nous allons fournir les détails d'implémentation correspondante. D'autre part, nous allons exposer les résultats d'évaluation de la performance de la solution de la mobilité forte.

# Chapitre 3

## Mise en œuvre de notre solution

### 3.1 Introduction

Afin de mettre en évidence la mobilité forte des processus d'orchestration, il est intéressant d'une part de s'attarder sur les architectures basées sur les mécanismes de checkpoint et de mobilité, d'autre part de comprendre l'ensemble des règles de transformation de code nécessaires pour assurer le maintien, la capture et le rétablissement de l'état d'exécution des processus d'orchestration. Ainsi, il est possible d'entamer la phase d'implémentation de notre approche appliquée pour une étude de cas qui sera décrite dans ce qui suit.

Dans la première section de ce chapitre, nous présentons en détail la solution de la mobilité forte inscrite dans le cadre de thèse de Mme. Soumaya Marzouk [29] à travers une étude des différents mécanismes de checkpoint et de mobilité. Ensuite, la deuxième section est réservée pour exposer les détails d'implémentation appliqués pour une étude de cas d'une agence de voyage. Enfin, la dernière section est consacrée pour présenter les résultats de l'évaluation des mécanismes précédemment décrits.

### 3.2 Principe de la mobilité forte appliquée aux processus d'orchestration

Dans cette section, nous allons décrire avec détails le principe de la solution de la mobilité forte appliquée dans le cas des processus d'orchestration.

#### 3.2.1 Architecture d'un processus d'orchestration mobile

Afin d'assurer la mobilité forte d'un processus d'orchestration, nous ajoutons principalement deux services de gestion appelés le Gestionnaire d'Invocation des

Services Web (WSIM) et le Gestionnaire de checkpoint des Services Web (WSCM).

Le WSIM est un service déployé entre le client et le service orchestré. Il traite le routage de messages entre le processus d'orchestration et ses clients. Ce service est essentiel dans le cas de migration des processus d'orchestration, car il assure le routage de la réponse résultante du nouvel hôte du processus d'orchestration vers le client initial.

Le WSCM est un service qui peut être déployé sur le même hôte du processus d'orchestration. Il est responsable de la gestion des checkpoints des instances des processus d'orchestration. En effet, lorsqu'une action de checkpoint est lancée au moment de l'exécution, une instance appelle le WSCM pour enregistrer une copie de son état de progrès (checkpoint). Ainsi, le WSCM sauvegarde tous les checkpoints capturés dans une base de données distante afin de permettre leur utilisation probable pour la reprise des instances interrompues.

### **3.2.1.1 Mécanisme de checkpoint périodique**

Comme l'illustre la figure 3.1, toutes les invocations des processus d'orchestration (OP) doivent passer à travers le WSIM (1). En règle générale, chaque invocation implique la création et l'exécution d'une nouvelle instance du processus d'orchestration. Chaque fois que cette instance atteint une position de checkpoint, elle envoie son état actuel au WSCM (2) qui le stocke, à son tour, dans une base de données distante (3). Quand la mobilité se produit, une copie du processus d'orchestration sera déployée sur un nouveau nœud et toutes (ou une partie de) ses instances en cours d'exécution seront arrêtées. Ensuite, une erreur (4) sera propagée au WSIM. Ce dernier (5) ré-envoie toutes les invocations interrompues au nouveau processus d'orchestration en vue de reprendre les instances suspendues. À la réception de ces requêtes, le nouveau processus d'orchestration crée, pour chacune, une nouvelle instance qui contacte le WSCM pour obtenir le dernier checkpoint capturé (6) et stocké dans la base de données distante (7), puis reprend son exécution correspondante.

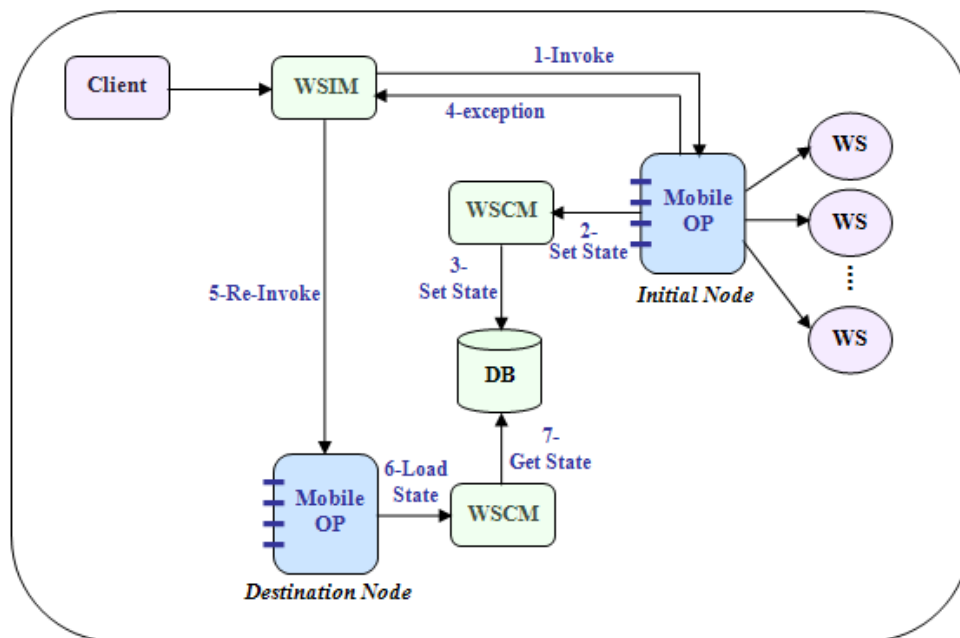


FIG. 3.1 – Architecture basée sur le mécanisme de checkpoint périodique

**Limitation du mécanisme de checkpoint périodique.** Comme illustré dans la figure 3.1, il est clair que les positions des checkpoints sont d'une part figées et d'autre part prédéfinies avant l'exécution du processus d'orchestration. Ainsi, cette architecture n'offre pas une certaine flexibilité au moment de l'exécution.

### 3.2.1.2 Mécanisme de checkpoint adaptatif

Pour palier à la contrainte précédemment citée, nous avons intégré les Aspects dans l'architecture des processus d'orchestration mobiles.

#### 3.2.1.2.1 Définition d'un Aspect

L'Aspect est l'unité de modularisation en POA (Programmation Orientée Aspect), tout comme la classe est l'implémentation d'un besoin commun en POO (Programmation Orientée Objet), [19] d'où une simplification du développement et diminution du risque d'erreurs, puisque les développeurs peuvent se focaliser sur un seul problème à la fois.

Une fois les différents Aspects définis, ils sont assemblés pour construire l'application. Ce processus d'intégration, essentiellement automatique, est appelé "tissage" (eng. Weaving).

#### 3.2.1.2.2 Principe

Au lieu d'avoir un appel direct à un module technique depuis un module métier, ou entre deux modules techniques différents, en programmation par Aspect, le code du module en cours de développement est concentré sur le but poursuivi, tandis qu'un Aspect est spécifié de façon autonome, implémentant un Aspect technique particulier, par exemple la persistance ou encore la génération de trace. Un ensemble de points d'insertions (eng. *joinpoint*) sont ensuite définis pour établir la liaison entre l'Aspect et le code métier ou un autre Aspect. Ces définitions de *joinpoint* sont déterminées dans le cadre de la POA. Selon les Framework ou les langages d'Aspects, la fusion du code technique avec le code métier est alors soit réalisée à la compilation, soit à l'exécution.

Aussi, l'astuce particulière de la programmation par Aspect consiste à utiliser un système d'expressions rationnelles pour préciser à quels points d'exécution (eng. *joinpoint*) du système l'Aspect spécifié devra être activé. Un Aspect permet donc de spécifier :

- les points d'action (eng. *pointcut*), qui définissent les points de jonction satisfaisants aux conditions d'activation de l'Aspect, donc le ou les moments où l'interaction va avoir lieu,
- les greffons, c'est-à-dire les programmes (eng. *advice*) qui seront activés avant, autour de ou après les points d'action définis.

### 3.2.1.2.3 Lexique d'un Aspect

La Programmation Orientée Aspect, vu qu'elle propose un paradigme de programmation et de nouveaux concepts, a développé un jargon bien spécifique de ses concepts qui sont, en définitive, simples mais puissants.

**Aspect.** Un module définissant des greffons et leurs points d'activation ;

**Greffon (eng. *advice*).** Un programme qui sera activé à un certain point d'exécution du système, précisé par un point de jonction ;

**Tissage ou tramage (eng. *weaving*).** Insertion statique ou dynamique dans le système logiciel de l'appel aux greffons ;

**Point d'action, de coupure ou de greffe (eng. *pointcut*).** Endroit du logiciel où est inséré un greffon par le tisseur d'Aspect ;

**Point de jonction ou d'exécution (eng. *joinpoint*).** Endroit spécifique dans le flot d'exécution du système, où il est valide d'insérer un greffon. Pour clarifier le propos, il n'est pas possible, par exemple, d'insérer un greffon au milieu du code d'une fonction. Par contre il est possible de le faire avant, autour de, à la place ou après l'appel de la fonction ;

**Considérations entrecroisées ou préoccupations transversales (eng. *cross-cutting concerns*).** Mélange, au sein d'un même programme, de sous-programmes distincts couvrant des Aspects techniques séparés.

#### 3.2.1.2.4 Avantages des Aspects

Le couplage entre les modules gérant des Aspects techniques peut être réduit de façon très importante, ce qui présente de nombreux avantages :

- Maintenance aisée : les modules techniques, sous forme d'Aspects, peuvent être maintenus plus facilement du fait de leur détachement de leur utilisation,
- Meilleure réutilisation : tout module peut être réutilisé sans se préoccuper de son environnement et indépendamment du métier ou du domaine d'application. Chaque module implémentant une fonctionnalité technique précise, il n'y a pas besoin de se préoccuper des évolutions futures : de nouvelles fonctionnalités pourront être implémentées dans de nouveaux modules qui interagiront avec le système au travers des Aspects.
- Gain de productivité : le programmeur ne se préoccupe que de l'Aspect de l'application qui le concerne, ce qui simplifie son travail, et permet d'augmenter la parallélisation du développement.
- Amélioration de la qualité du code : la simplification du code qu'entraîne la programmation par Aspect permet de le rendre plus lisible et donc de meilleure qualité.

#### 3.2.1.2.5 Architecture basée sur le mécanisme de checkpoint adaptatif

Ainsi, il serait intéressant d'intégrer les Aspects afin de définir une architecture de mobilité des processus d'orchestration plus performante. La figure 3.2 illustre le mécanisme de checkpoint basé sur les Aspects. En effet, nous remarquons la présence de deux Aspects dans cette architecture. Le premier correspond à un Aspect de capture déployé entre le WSCM et le nœud initial. Il permet suite à son déploiement dynamique de sauvegarder l'état courant d'exécution du processus.

Le deuxième Aspect est responsable du ré établissement de l'état d'exécution correspondant au dernier checkpoint sauvegardé. Il est déployé entre le WSCM et le nouveau nœud vers lequel a migré le processus d'orchestration.

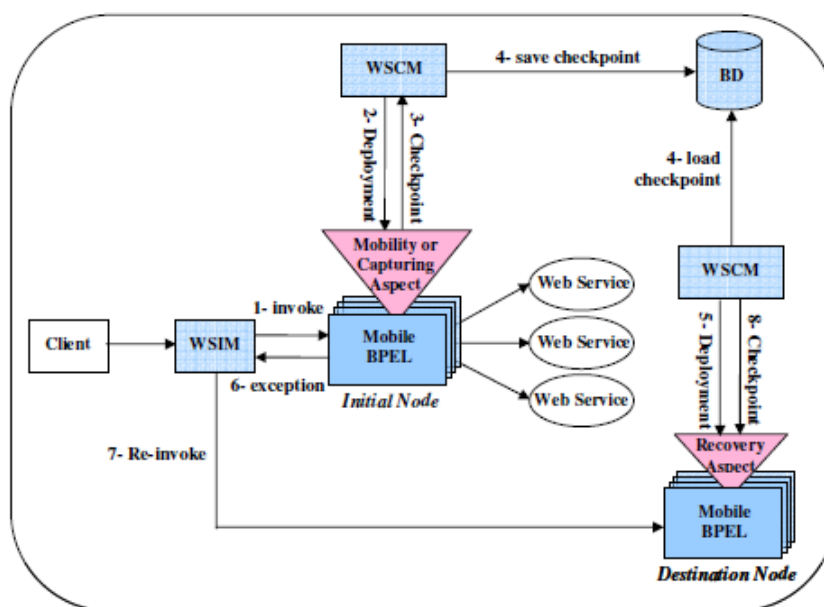


FIG. 3.2 – Architecture basée sur le mécanisme de checkpoint adaptatif

### 3.2.1.2.6 Exemples d'Aspects

Nous présentons dans ce qui suit deux exemples d'Aspect de capture d'état d'exécution défini dans le cadre du mécanisme de checkpoint adaptatif précédemment décrit.

Comme présenté dans la figure 3.3, le code du premier exemple d'Aspect consiste à sauvegarder l'état courant de l'exécution par la biais de la méthode « `setState()` ». Cet Aspect est déployé au niveau des barrières naturelles du processus d'orchestration ( nous rappelons que c'est-à-dire au début ou à la fin d'un bloc « Flow » ou encore dans un bloc élémentaire au début d'une structure séquentielle). Il est à noter qu'il n'y a pas de coût de synchronisation à ce niveau.

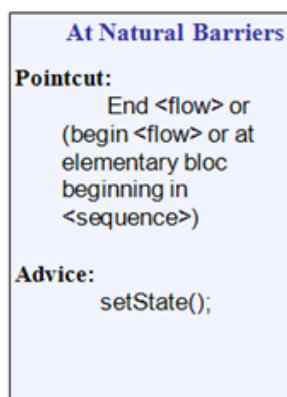


FIG. 3.3 – Exemple d'Aspect déployé aux barrières naturelles

D'après la figure 3.4 , la différence entre le premier et le deuxième exemple d'Aspect est que celui-ci est déployé au début d'un bloc élémentaire, ce qui implique la nécessité de synchroniser toutes les branches en parallèle, si elles existent, avant de faire un checkpoint.

```
Forced Checkpoint  
Pointcut:  
Elementary bloc  
beginning  
Advice:  
Si (NbBranche==NBmax)  
    setState();  
Else  
    NbBranche++;  
    wait until State Save  
FinSi
```

FIG. 3.4 – Exemple d'Aspect d'un checkpoint forcé

### 3.2.2 Migration du processus d'orchestration

Un processus d'orchestration est un programme permettant la composition d'un ensemble de services Web, dans une structure d'un workflow, afin d'assurer une certaine fonctionnalité. Plusieurs langages permettent la description de ces processus. Dans ce projet, nous avons employé la spécification WS-BPEL comme étant un langage de description des processus d'orchestration. En conséquence, nous présentons des règles de transformation permettant la génération d'un processus d'orchestration basé WS-BPEL fortement mobile.

Néanmoins, notre approche n'est pas spécifique à WS-BPEL, mais elle peut être appliquée à d'autres langages.

#### 3.2.2.1 Règles de transformation du code source

Assurer une forte mobilité d'un processus BPEL consiste à intégrer la capacité à capter l'état de son exécution (checkpoint), ainsi que la possibilité de charger un point de reprise (rétablissement) afin d'en reprendre l'exécution.

Dans notre approche, ces fonctionnalités sont réalisées grâce à la transformation du code du processus BPEL initial. En effet, le code du processus est instrumenté dans le but, pour chaque instance en cours d'exécution, (1) de maintenir son perpétuel état mis à jour, (2) de capturer et de sauvegarder l'état courant d'exécution

lorsqu'une position de checkpoint est atteinte, et enfin (3) de charger un checkpoint et de poursuivre l'exécution à partir de celui-ci.

### 3.2.2.1.1 Maintien de l'état d'exécution du processus BPEL

En vue de maintenir l'état d'exécution d'un processus d'orchestration, nous transformons son code par l'ajout d'un ensemble de variables correspondant à son état actuel d'exécution. Ce dernier est composé principalement de l'identifiant de l'instance du processus d'orchestration, de l'ensemble de ses variables locales et de la position atteinte dans l'exécution du processus. Celle-ci représente la position de la prochaine activité à réaliser. Il est à noter qu'il y a tant de variables de position que de branches (le nombre de variables de position est égal au nombre maximal de branches en parallèle). L'ensemble de ces données représente le checkpoint propre à une instance du processus d'orchestration.

Le maintien de cet état mis à jour requiert l'instrumentation du code du processus BPEL. Les transformations correspondent principalement à la gestion des positions des prochaines activités qui seront effectuées. Pour ce faire, nous définissons une règle de transformation pour chaque type d'activité BPEL. Nous distinguons des règles spécifiques pour les activités de base et d'autres pour les activités structurées. Pour les activités de base, nous définissons une règle de transformation dans la figure 3.5 qui vise à référencer un ensemble d'activités avec un numéro de position. Nous appelons cet ensemble un bloc élémentaire. Cette règle de transformation consiste en (1) la mise à jour de la position de la prochaine activité à effectuer et (2) le test si l'activité en cours correspond à celle qui devrait être réalisée. Ainsi, cette règle sera appliquée aux activités `invoke`, `receive`, `reply`, `wait`, et `assign`.

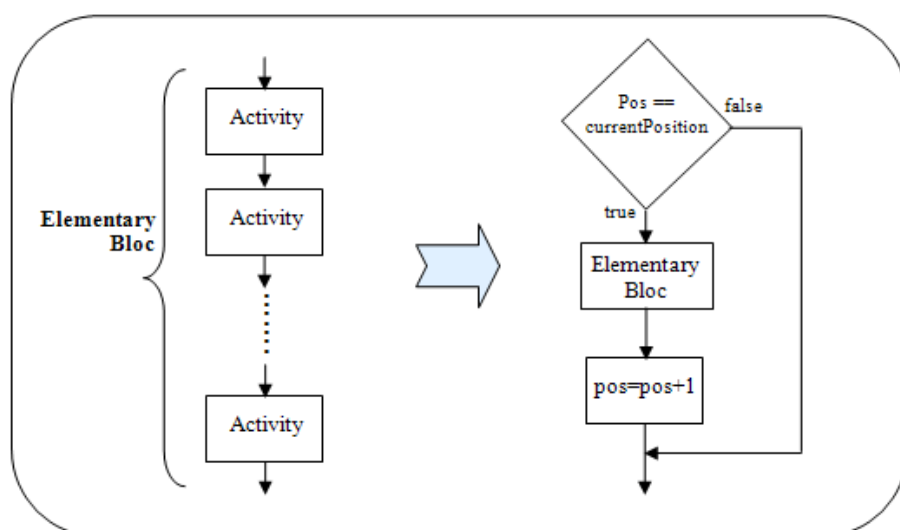


FIG. 3.5 – Règle de transformation des activités de base

En outre, les activités link doivent être instrumentées en vue de maintenir la cohérence du processus après la migration. En effet, l'activité link permet d'interdire l'exécution d'une activité jusqu'à la fin d'une autre. Cette information sera perdue au moment de la migration car elle est maintenue par le moteur d'orchestration. Pour éviter ce problème, nous remplaçons cette activité, comme indiqué dans la figure 3.6, par une activité wait qui retarde l'exécution de l'activité en question jusqu'à ce que la variable booléenne link1 devienne "true".

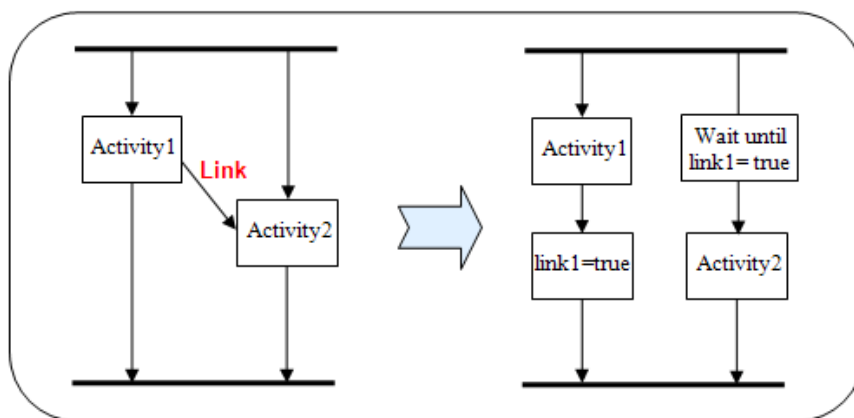


FIG. 3.6 – Règle de transformation de l'activité link

Les règles de transformation des activités structurées sont plus complexes. En particulier, pour les activités if, switch, while, forEach, repeatUntil et Pick. La figure 3.7 présente la règle de transformation de l'activité while.

Cette règle garantit que l'accès à la structure while est fait uniquement si la position de l'activité en cours, qui doit être exécutée, correspond à l'une des positions des activités situées dans la boucle. En outre, cette transformation assure que, lors de la fin d'exécution d'une boucle, la position de la prochaine instruction correspond à la position de l'activité suivante. De même, nous avons défini, une règle de transformation pour chaque type d'activité structurée.

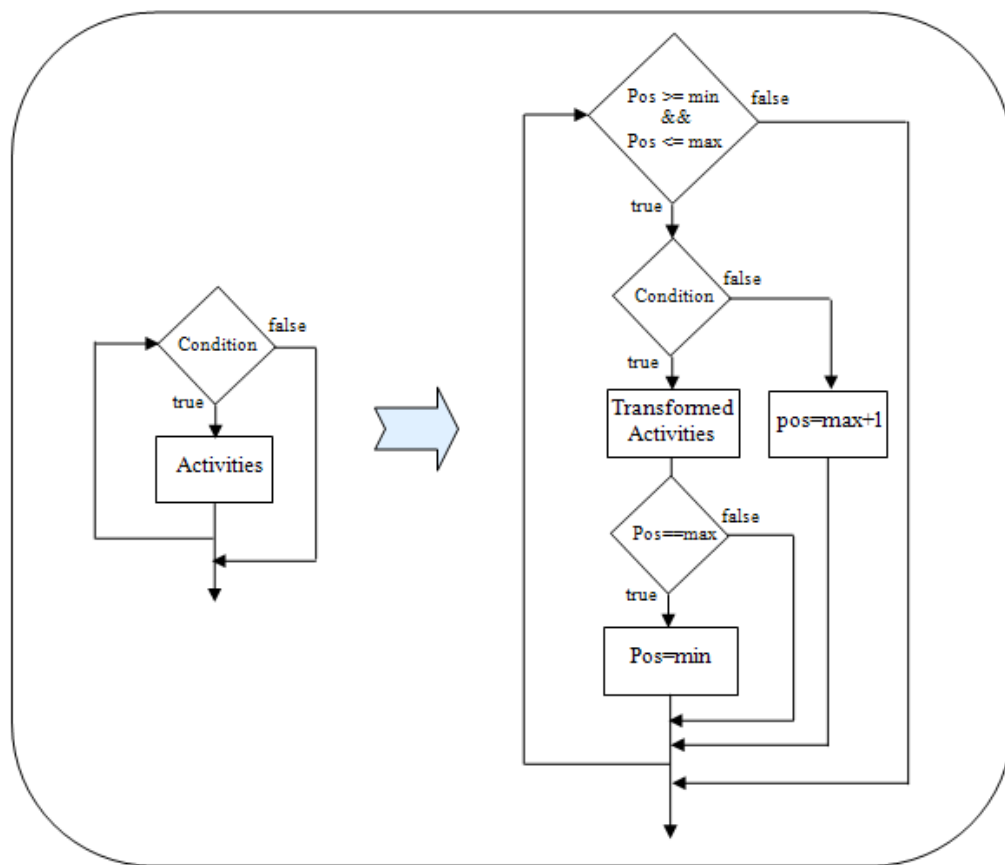


FIG. 3.7 – Règle de transformation de l'activité while

### 3.2.2.1.2 Capture de l'état du processus BPEL

La capture d'un checkpoint d'une instance d'un processus BPEL correspond à sauvegarder son état d'exécution dans le WSCM. Cette opération est très simple lorsque le checkpoint de l'instance est nécessaire lors de l'exécution d'une activité incluse dans une séquence. Dans un tel cas et afin d'assurer la cohérence du checkpoint, celui-ci sera réalisé après la fin du bloc élémentaire actuel. Tel que présenté dans la figure 3.8, l'opération de checkpoint correspond à l'invocation du WSCM dans l'ordre de sauvegarder l'ensemble des valeurs constituant un état d'une instance BPEL.

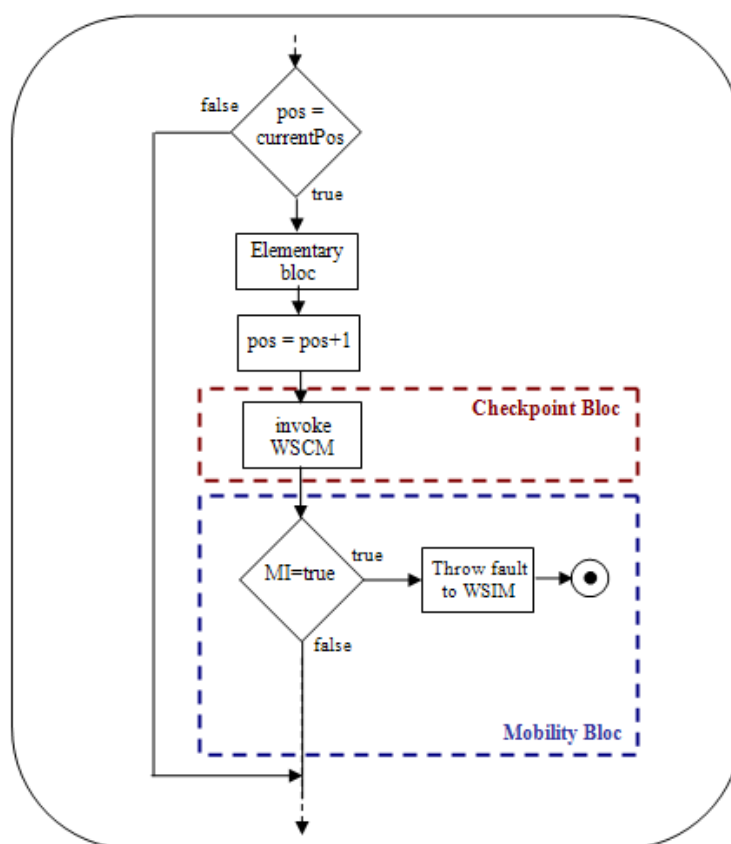


FIG. 3.8 – Capture de l'état d'exécution du processus BPEL dans une structure séquentielle

Toutefois, si le checkpoint est requis lors de l'exécution d'une activité qui est incluse dans une structure flow, le problème de la cohérence du checkpoint devient plus délicat. En effet, la capture d'un tel checkpoint ne peut pas être faite directement en raison de la structure du processus d'orchestration qui admet plusieurs branches d'activités exécutées en parallèle. Une telle situation est similaire au checkpoint d'applications distribuées constituées d'un ensemble de composants en parallèle. Pour de telles applications, tous les processus doivent être synchronisés (actions de checkpoint coordonnées) avant de capturer l'état d'exécution afin d'assurer sa cohérence. Nous adoptons l'approche des checkpoints coordonnés par la synchronisation de toutes les branches exécutées en parallèle de l'instance du processus BPEL, avant de procéder à un checkpoint. Cette transformation est illustrée dans la figure 3.9 (bloc de checkpoint) et doit être appliquée, à chaque fois qu'un checkpoint est requis, à la fin d'un bloc élémentaire de chaque branche d'une structure flow ou de toute autre structure parallèle telle que la structure forEach.

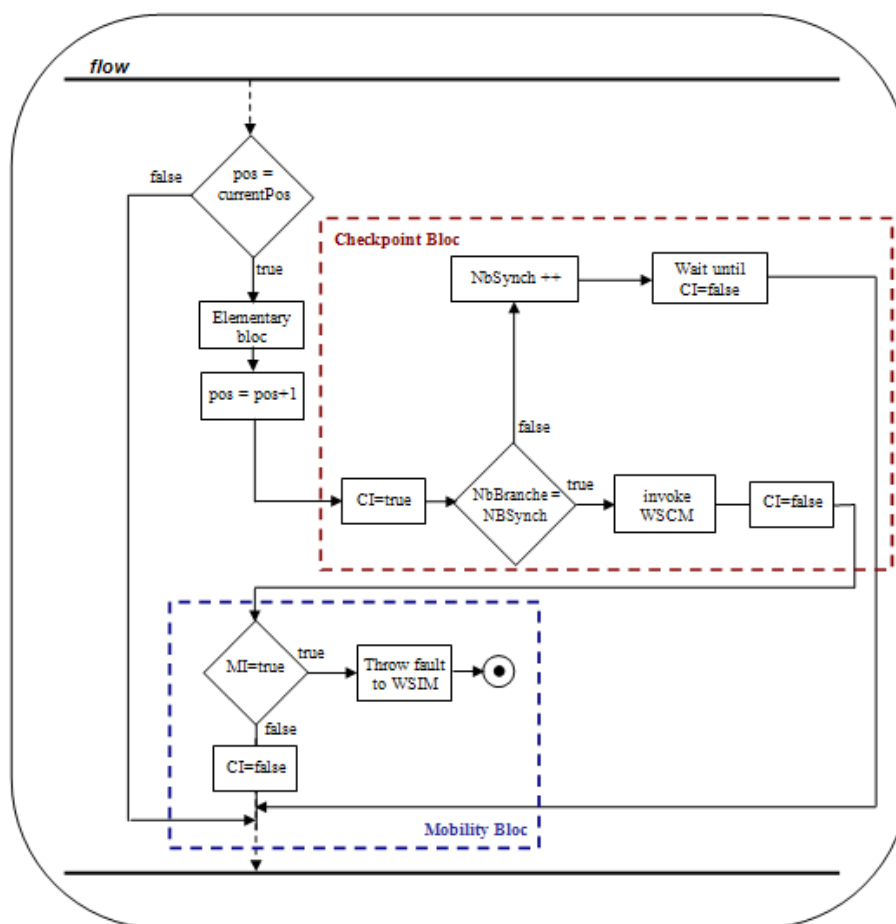


FIG. 3.9 – Synchronisation des activités dans une structure « flow »

Ainsi, deux autres variables doivent être ajoutées à l'état du processus BPEL. La première notée *NbBranche* représente le nombre de branches courant. La deuxième variable *NbSynch* correspond au nombre de branches déjà synchronisées. De cette façon, lorsque toutes les branches sont synchronisées, l'état du processus est envoyé au WSCM. En outre, nous ajoutons d'autres variables au processus d'orchestration qui sont l'indicateur de checkpoint (CI) montrant que l'action d'un checkpoint est déclenchée, et l'indicateur de la mobilité (MI) informant qu'une mobilité forcée est requise.

Toutefois, les positions de checkpoint, nécessitant une synchronisation, doivent être choisies avec soin pour éviter les blocages. Par exemple, si la synchronisation bloque une activité *a1*, qui devrait être exécutée avant une activité *a2* (c'est-à-dire en utilisant l'activité *link*), et si l'activité *a2* est située avant la barrière de synchronisation, dans un bloc appartenant à une autre branche du processus d'orchestration. Ainsi, toute l'instance sera bloquée. Afin d'éviter une telle situation, la transformation d'une activité *link* devrait inclure un bloc de checkpoint avant l'exécution de

l'activité en question. Ce bloc ne sera exécuté que si une action de checkpoint est déclenchée. Ainsi, la condition d'entrée à l'activité wait sera  $link1=true$  ou  $CI=true$  comme illustré dans la figure 3.10.

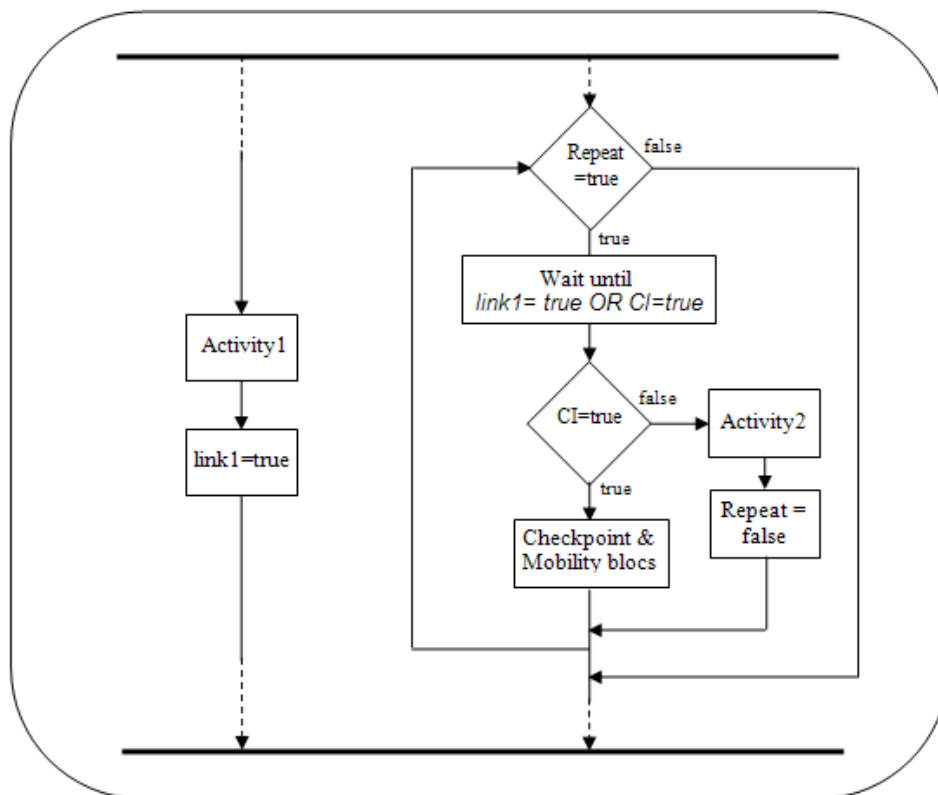


FIG. 3.10 – Transformation des liens des activités au sein d'une structure « flow »

De plus, chaque lien de l'activité ciblée doit être situé au début d'un bloc élémentaire, afin de maintenir la cohérence de l'exécution après la migration. Sinon, toutes les activités situées avant l'activité en question dans le bloc élémentaire seront ré-exécutées.

### 3.2.2.1.3 Migration du processus BPEL

Comme indiqué précédemment, la migration du processus BPEL peut avoir lieu dans deux cas. Le premier correspond à une indisponibilité du nœud BPEL. Dans un tel cas, toutes les instances en cours d'exécution seront interrompues et une exception sera automatiquement propagée au WSIM. Dans ce cas, une solution consiste à déployer un nouveau processus BPEL et à reprendre l'exécution de toutes les instances à partir du dernier checkpoint capturé.

En cas de mobilité forcée, la migration d'un processus BPEL est lancée par l'envoi d'un ordre de migration (avec  $MI = true$  en entrée) à un sous-ensemble de ses

instances. Cet ordre est reçu par l'instance à travers une activité `receive` ajoutée en parallèle au processus originel comme illustré dans la figure 3.11. En conséquence, la variable `MI` sera affectée à `true`, ce qui permet l'exécution du bloc de la mobilité (c'est-à-dire interruption et mobilité), après la capture du prochain checkpoint. Dans ce bloc, une exception sera artificiellement jetée au `WSIM` pour l'informer de l'interruption de l'instance. Pour éviter de bloquer le processus d'orchestration en l'absence de mobilité, l'ordre de mobilité sera envoyé à la fin du processus d'orchestration (avec `MI = false`) afin de permettre la terminaison du processus d'orchestration.

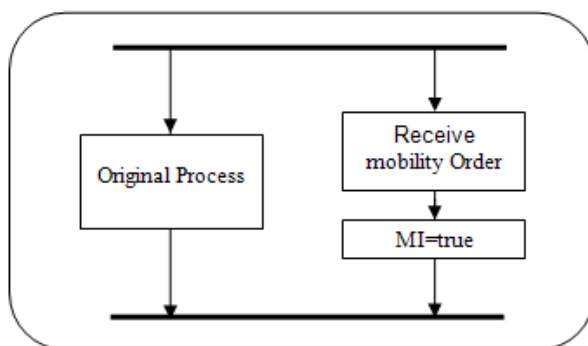


FIG. 3.11 – Transformation assurant la réception d'un ordre de checkpoint/migration

#### 3.2.2.1.4 Rétablissement de l'exécution du processus BPEL

Après le déploiement du processus BPEL sur le nouvel hôte, le `WSIM` reprend l'exécution de cette instance en utilisant les paramètres de la requête initiale. Dès la réception de cette requête, le processus BPEL contacte le `WSCM` pour obtenir le dernier checkpoint capturé de l'instance arrêtée, puis copie le résultat pour l'initialisation de l'état de l'instance en cours et, enfin, reprend son exécution. Ceci est réalisé grâce à un bloc de reprise ajouté au début du processus BPEL. Ce bloc est exécuté seulement si l'invocation requiert la reprise d'une instance arrêtée. L'identification d'une telle situation se fait par la corrélation des messages. C'est pourquoi, si les messages BPEL initiaux assurant la création d'instance ne sont pas corrélés, le `WSIM` y ajoute un identificateur avant d'invoquer le processus BPEL. En recevant le message, ce dernier vérifie l'existence d'un checkpoint correspondant à l'instance identifiée. Si un tel checkpoint existe, le bloc de récupération sera exécuté.

Toutefois, ces transformations ne sont pas toujours suffisantes pour assurer une reprise correcte des instances du processus BPEL mobile. Tel est le cas si le processus BPEL mobile utilise des invocations asynchrones (`invoke receive`), et si le processus est arrêté avant la réception de la réponse des partenaires. En effet, celle-ci sera perdue, car le service partenaire enverra la réponse à l'ancien hôte du processus BPEL.

Dans la finalité d'éviter ce problème, une invocation qui ne reçoit pas de réponse avant la migration devrait être réémise. Une autre solution peut consister en choisissant avec soin les positions de checkpoint afin d'éviter de telles situations. Ainsi, mettre des positions de checkpoint entre un invoke et son receive correspondant devrait être évité.

### 3.3 Etude de cas d'une agence de voyage

Dans cette section, nous présentons notre étude de cas utilisée afin de montrer la faisabilité de notre solution. Pour cela, nous réalisons une implémentation d'un processus BPEL appliquée dans le cas d'une agence de voyage. Par la suite, nous décrivons comment le transformer en un processus fortement mobile, dans l'ordre de pouvoir l'exploiter ensuite dans l'élaboration d'une démarche de validation des actions de mobilité opérant dans la phase de planification.

#### 3.3.1 Réalisation d'une étude de cas : Processus d'orchestration BPEL d'une agence de voyage

L'application consiste en un processus BPEL orchestrant l'ensemble des services de réservation d'un vol et d'une chambre d'un hôtel pour une agence de voyage. Nous considérons que les paramètres d'entrée relatifs au client de ce processus sont : la date de départ, la date d'arrivée, la ville de départ et celle d'arrivée. Quant au paramètre de sortie de ce processus est sous la forme d'une offre finale informant le client sur les détails de son voyage.

En assumant que les compagnies partenaires de cette agence fournissent leurs services en utilisant des services Web, il est alors possible de composer les services Web des compagnies aériennes et des chaînes hôtelières. En effet, le processus BPEL invoque les services de réservation d'un vol en parallèle pour obtenir leurs offres, il les compare de façon à choisir celle la moins chère. Il refait les mêmes opérations avec les services de réservation d'une chambre d'un hôtel. Enfin, ce processus retourne au client l'offre globale la moins chère.

La figure 3.12 montre le code de base du processus de l'agence de voyage, qui implémente l'opération « getTravelPackage ». En fait, nous allons juste présenter des simples invocations élémentaires aux services Web partenaires. Pour ce faire, le processus déclare trois liens des partenaires qui connectent la composition respectivement au client, au service Web de réservation d'un vol et au service Web de réservation d'une chambre d'un hôtel. Il déclare aussi six variables pour supporter les messages requête et réponse du client, de même ceux relatifs aux invocations des

opérations « findAFlight » et « findARoom » des services Web correspondants. Ensuite, nous trouvons l'activité « receive » qui relie l'opération « getTravelPackage » à une activité « assign » pour copier les informations de la variable « clientrequest » aux variables « flightrequest » et « hotelrequest ». En plus, les deux activités « invoke » suivantes appellent les services de réservation d'un vol et d'une chambre d'un hôtel. Outre cela, une activité « assign » est utilisée pour copier les informations de vol et d'hôtel depuis les variables « flightresponse » et « hotelresponse » dans la variable « clientresponse ». L'activité « reply » envoie l'offre finale au client.

```

<process name="travelPackage">
  <partnerLinks>
    <partnerLink name="client" partnerLinkType="clientPLT" .../>
    <partnerLink name="flight" partnerLinkType="flightPLT" .../>
    <partnerLink name="hotel" partnerLinkType="hotelPLT" .../>
  </partnerLinks>
  <variables>
    <variable name="clientrequest" messageType="findPackageRequest"/>
    <variable name="clientresponse" messageType="findPackageResponse"/>
    <variable name="flightrequest" messageType="findAFlightRequest"/>
    <variable name="flightresponse" messageType="findAFlightResponse"/>
    <variable name="hotelrequest" messageType="findARoomRequest"/>
    <variable name="hotelresponse" messageType="findARoomResponse"/>
  </variables>
  <receive name="receiveClientRequest" partnerLink="client"
    portType="travelServicePT" operation="getTravelPackage"
    variable="clientrequest" createInstance="yes"/>
  <assign>
  <copy>
    <from variable="clientrequest" part="deptDate"/>
    <to variable="flightrequest" part="DepartOn"/>
  </copy>
  ...
  <assign>
  <invoke name="invokeFlightServiceIP"
    partnerLink="flight" portType="flightPT" operation="findAFlight"
    inputVariable="flightrequest" outputVariable="flightresponse"/>
  <invoke name="invokeHotelServiceIP"
    partnerLink="hotel" portType="HotePT" operation="findARoom"
    inputVariable="hotelrequest" outputVariable="hotelresponse"/>
  <assign>
  <copy>
    <from variable="flightresponse" part="flightDetails"/>
    <to variable="clientresponse" part="flightInfo"/>
  </copy>
  <copy>
    <from variable="hotelresponse" part="roomDetails"/>
    <to variable="clientresponse" part="hotelInfo"/>
  </copy>
  <assign>
  <reply name="replyToClient" partnerLink="client" portType="travelServicePT"
    operation="getTravelPackage" variable="clientresponse" />
</process>

```

FIG. 3.12 – Extrait du code de base du processus BPEL de l'agence de voyage

En se penchant sur le code de base présenté précédemment, nous avons procédé à l'exécuter en parallèle en supposant que l'agence de voyage contacte à la fois deux compagnies aériennes puis deux chaînes hôtelières. En plus, nous avons dupliqué ce bloc résultat pour être exécuté en séquence afin de pouvoir intégrer aisément dans ce qui suit les positions de checkpoint au niveau d'une structure « flow » ou d'une

structure séquentielle. La figure 3.13 illustre graphiquement cette description plus clairement. Ainsi, notre processus prototype final contient une séquence englobant deux structures « flow ».

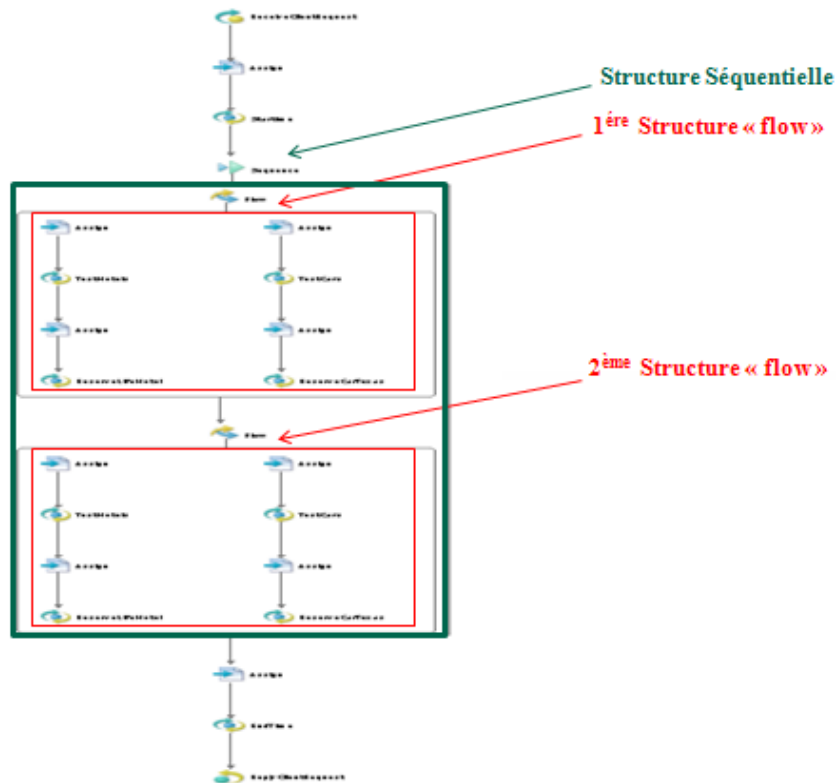


FIG. 3.13 – Illustration graphique du processus prototype BPEL

### 3.3.2 Transformation du processus réalisé en un processus fortement mobile

Nous avons instrumenté notre application dans le but d'ajouter un support de mobilité au processus BPEL. Ainsi, nous avons introduit un ensemble de positions de checkpoint et nous avons appliqué les différentes règles de transformation afin de générer un service d'agence de voyage mobile.

#### 3.3.2.1 Implémentation du WSIM

Nous avons développé le WSIM sous forme d'un processus BPEL. Il est déployé entre le client et le processus d'orchestration pour traiter le routage de messages entre eux. Ce service est essentiel dans le cas de migration des processus d'orchestration, car il assure le routage de la réponse résultante du nouvel hôte du processus

d'orchestration vers le client initial. Nous présentons dans la figure 3.14 un extrait du code BPEL du WSIM dans le cas d'une seule migration.

```
<bpel:invoke inputVariable="invokeDPin" name="InvokeDP"  
  operation="clientRequest" outputVariable="invokeDPout"  
  partnerLink="invokeDP">  
  
<bpel:condition>contains($invokeDPout/DefaultNamespace:clientRequestReturn, "exception")</bpel:condition>  
  
<bpel:invoke inputVariable="invokeFPin" name="InvokeFP"  
  operation="clientRequest" outputVariable="invokeFPout"  
  partnerLink="invokeFP">
```

FIG. 3.14 – Extrait du code du WSIM

Dans cet exemple d'extrait du code BPEL, nous supposons qu'il y a un problème suite à l'invocation « InvokeDP » d'un processus d'orchestration initial, ainsi une erreur est propagée au WSIM. Une fois détectée, cette entité ré envoie toutes les invocations interrompues au nouveau processus d'orchestration en vue de reprendre les instances suspendues, ceci est illustré dans la figure 3.14 par l'invocation nommée « InvokeFP » de ce processus d'orchestration.

### 3.3.2.2 Implémentation du WSCM

Le WSCM est un service sans état pouvant être déployé sur le même hôte du processus d'orchestration. Il est responsable de la gestion des checkpoints des instances du processus d'orchestration. En effet, lorsqu'une action de checkpoint est lancée au moment de l'exécution, une instance appelle le WSCM pour enregistrer une copie de son état de progrès. Ainsi, le WSCM enregistre tous les checkpoints capturés dans une base de données distante afin de permettre leur utilisation pour la reprise des instances interrompues.

#### 3.3.2.2.1 Implémentation du « WSCM.wsdl »

La figure 3.15 correspond à un extrait du code de l'interface du service WSCM. En effet, celui-ci possède deux méthodes. La première est « saveState », elle a comme rôle de sauvegarder l'état de l'invocation interrompue dans la base de données distante. La deuxième méthode est « getState », elle permet de récupérer l'état d'avancement de l'invocation interrompue à partir de la base de données grâce à l'identifiant de l'instance correspondante.

```
<wsdl:definitions>
.....
  <wsdl:message name="getStateRequest">
    <wsdl:part name="OPIdentifier" type="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="getStateResponse">
    <wsdl:part name="getStateReturn" type="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="setStateRequest">
    <wsdl:part name="opidentifier" type="xsd:string"/>
    <wsdl:part name="deptcity" type="xsd:string"/>
    <wsdl:part name="destcity" type="xsd:string"/>
    <wsdl:part name="nbbranch" type="xsd:string"/>
    <wsdl:part name="pos1" type="xsd:string"/>
    <wsdl:part name="pos2" type="xsd:string"/>
    <wsdl:part name="deptdate" type="xsd:string"/>
    <wsdl:part name="retdate" type="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="setStateResponse">
    <wsdl:part name="setStateReturn"/>
  </wsdl:message>
.....
</wsdl:definitions>
```

FIG. 3.15 – Implémentation du « WSCM.wsdl »

### 3.3.2.2.2 Implémentation du « WSCM.java »

La figure 3.16 présente le code des deux méthodes du WSCM, à savoir « setState » et « getState ».

```

////////////////////////////////////***** Set State *****/////////////////////////////////////
public void setState(String opidentifier, String deptcity, String destcity, String nbbranch,
                    String pos1, String pos2, String deptdate, String retdate) throws SQLException, ParseException {
    .....
    // Connection to the database
        con=new Connexion();
        stmt=con.connect_stmt();
        if (con == null)
            throw new RuntimeException("No connection to the database returned!");
    .....
    // Insertion request
    String theQuery = "INSERT INTO statedb VALUES (" + opidentifier + "," + deptcity + "," + destcity + "," +
nbbranch + "," + pos1 + "," + pos2 + "," + deptdate + "," + retdate + ")";
    .....
}

////////////////////////////////////***** Get State *****/////////////////////////////////////
public String getState(String opidentifier) throws SQLException, ParseException {
    .....
    // Connection to the database
        con=new Connexion();
        stmt=con.connect_stmt();
        if (con == null)
            throw new RuntimeException("No connection to the database returned!");
        String testQuery = "SELECT deptcity, destcity, deptdate, retdate, nbbranch, pos1, pos2 " +
"FROM statedb WHERE opidentifier=" + opidentifier + """;
    // Execution of the query
    try {
        rs = stmt.executeQuery(testQuery);
    .....
    while (rs.next()) {
    .....
        deptcity=rs.getString("deptcity"); destcity=rs.getString("destcity"); deptdate=rs.getString("deptdate");
        retdate=rs.getString("retdate"); nbbranch=rs.getString("nbbranch"); pos1=rs.getString("pos1");
        pos2=rs.getString("pos2");
    }
    } catch (SQLException se) {
    .....
    }
}

```

FIG. 3.16 – Implémentation des méthodes du « WSCM.java »

### 3.3.2.3 Transformation du code du processus BPEL réalisé

Assurer une forte mobilité d'un processus BPEL consiste à intégrer la capacité à capter l'état de son exécution (checkpoint), ainsi que la possibilité de charger un point de reprise (rétablissement) afin de reprendre l'exécution interrompue.

Dans notre approche, ces fonctionnalités sont réalisées grâce à la transformation du code du processus BPEL initial. En effet, le code du processus est instrumenté dans le but, pour chaque instance en cours d'exécution, (1) de maintenir son perpétuel état mis à jour, (2) de capturer et de sauvegarder l'état courant d'exécution lorsqu'une position de checkpoint est atteinte, et enfin (3) de charger un checkpoint et de poursuivre l'exécution à partir de celui-ci.

Comme nous avons vu dans la section précédente, il convient d'affecter un numéro de position pour chaque bloc élémentaire. Ainsi, celui-ci, accompagné de la mise à jour de sa position correspondante, forme une opération atomique au cours de

laquelle la migration n'est pas autorisée. Il est à noter que ces blocs ne doivent pas contenir des rubriques de structures de contrôle ou des boucles. En plus, la taille de chaque bloc doit être sélectionnée d'une façon judicieuse, elle ne doit pas être grande ce qui cause le retard de la migration, non plus trop petite ce qui engendre l'augmentation de la taille du code transformé par rapport à celui initial.

En suivant ces consignes, nous avons choisi les blocs élémentaires auxquels nous avons affecté des numéros de positions. Ensuite, nous avons appliqué la règle de transformation assurant la capture de l'état d'exécution du processus BPEL dans une structure séquentielle deux fois. Une fois juste après la première structure « flow » et une autre aussi juste après la deuxième structure « flow ». Outre cela, nous avons synchronisé chaque deux invocations en parallèle des services de réservation d'un vol se trouvant au milieu des deux structures « flow ». Enfin, nous avons ajouté un bloc de rétablissement au début du processus BPEL transformé. En fait, ce bloc est responsable de contacter le WSCM, grâce à la méthode « getState », afin de charger l'état capturé d'une instance interrompue.

## **3.4 Mise en oeuvre des différents mécanismes de checkpoint**

Afin d'évaluer la performance de notre solution, nous proposons de décrire notre architecture de déploiement utilisée pour mesurer le surcoût résultant des différents mécanismes de checkpoint.

### **3.4.1 Architecture de déploiement de notre solution**

Nous présentons dans la figure 3.17 l'architecture de déploiement que nous avons utilisée pour mettre en évidence la mobilité forte du processus d'orchestration BPEL associé à notre étude de cas.

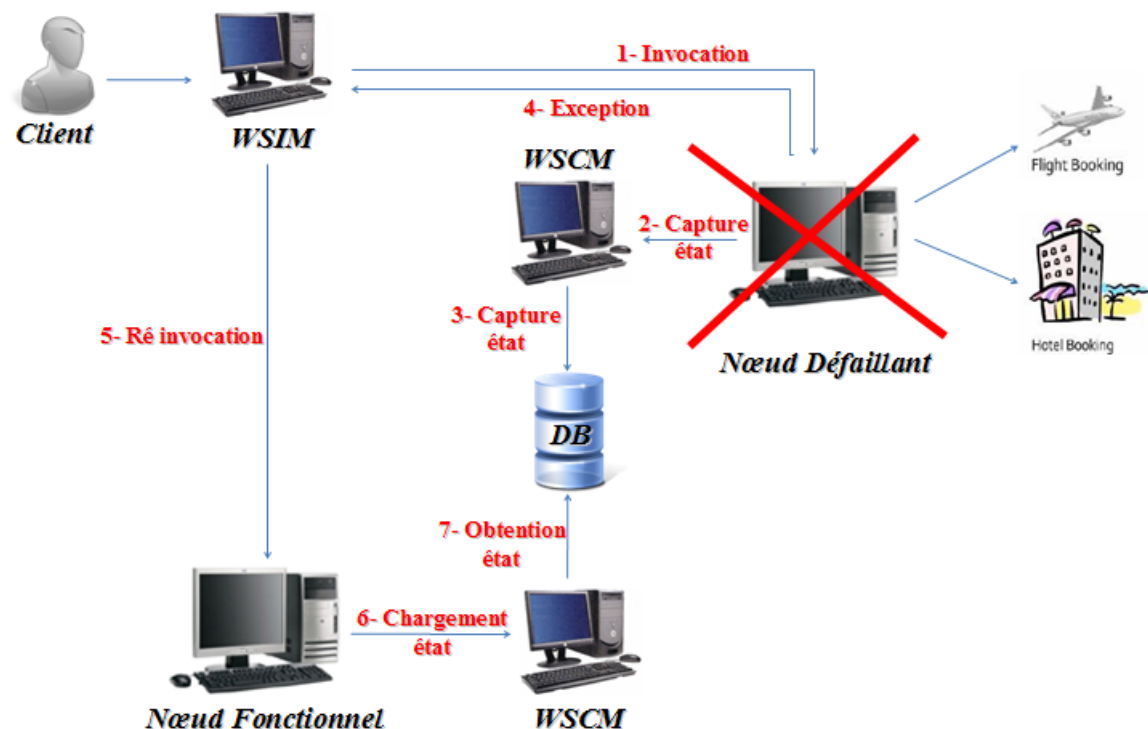


FIG. 3.17 – Architecture de déploiement de notre solution

Notre architecture est constituée du client, du WSIM, du WSCM, de l'hôte BPEL initial et de celui final. Nous avons ainsi utilisé cinq machines connectées via un réseau local à l'unité de recherche ReDCAD et caractérisées par un processeur 3GHz et une mémoire centrale 500Mo. Le WSIM étant implémenté sous forme d'un processus BPEL et le WSCM correspond à un service Web sans état communiquant avec une base de données distante.

Nous avons utilisé comme SGBD PostgreSQL version 8.3. En plus, nous avons commencé par choisir l'outil ActiveVOS Designer version 6.0.1 afin de mettre en œuvre le mécanisme de checkpoint périodique. Ce choix est justifié par le fait qu'avec ActiveVOS, il est facile de concevoir, construire, gérer, déployer et maintenir les applications à base de services. En outre, ce logiciel est léger et indépendant de la plateforme de travail[1]. Sauf que le moteur d'orchestration du produit ActiveVOS ne supporte pas le déploiement des Aspects nécessaire pour implémenter le mécanisme de checkpoint adaptatif. Cependant, nous avons choisi le moteur d'orchestration AO4BPEL. Il s'agit d'une extension du BPEL orientée Aspect qui soutient la définition de flux de travail pour les Aspects des processus BPEL [16].

### 3.4.2 Evaluation de la performance du mécanisme de checkpoint périodique

Nous considérons dans cette section que le processus BPEL mobile contient quatre positions de checkpoint dont deux se trouvent au sein de deux structures « flow » et les deux autres font partie d'une structure séquentielle. En plus, nous considérons dans les évaluations suivantes que la mobilité est déclenchée à 50% de l'exécution de notre processus. La figure 3.18 illustre graphiquement les positions de checkpoint au sein de notre processus BPEL.

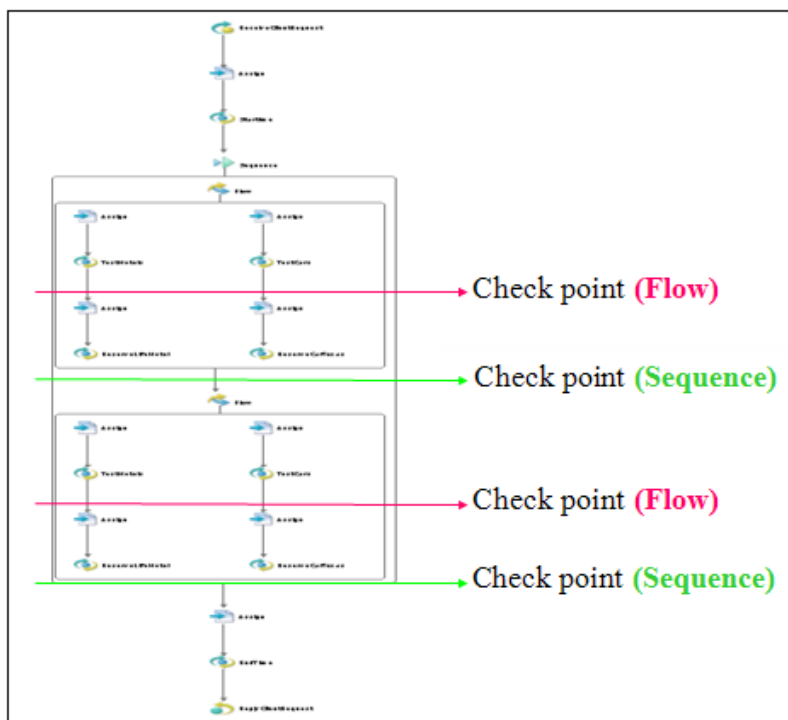


FIG. 3.18 – Illustration graphique des positions de checkpoint dans le processus BPEL transformé

#### 3.4.2.1 Evaluation à distance du mécanisme de checkpoint périodique

La figure 3.19 correspond à une évaluation à distance, selon l'architecture de déploiement décrite précédemment, de la mobilité forte appliquée en utilisant le mécanisme de checkpoint périodique pour plusieurs clients.

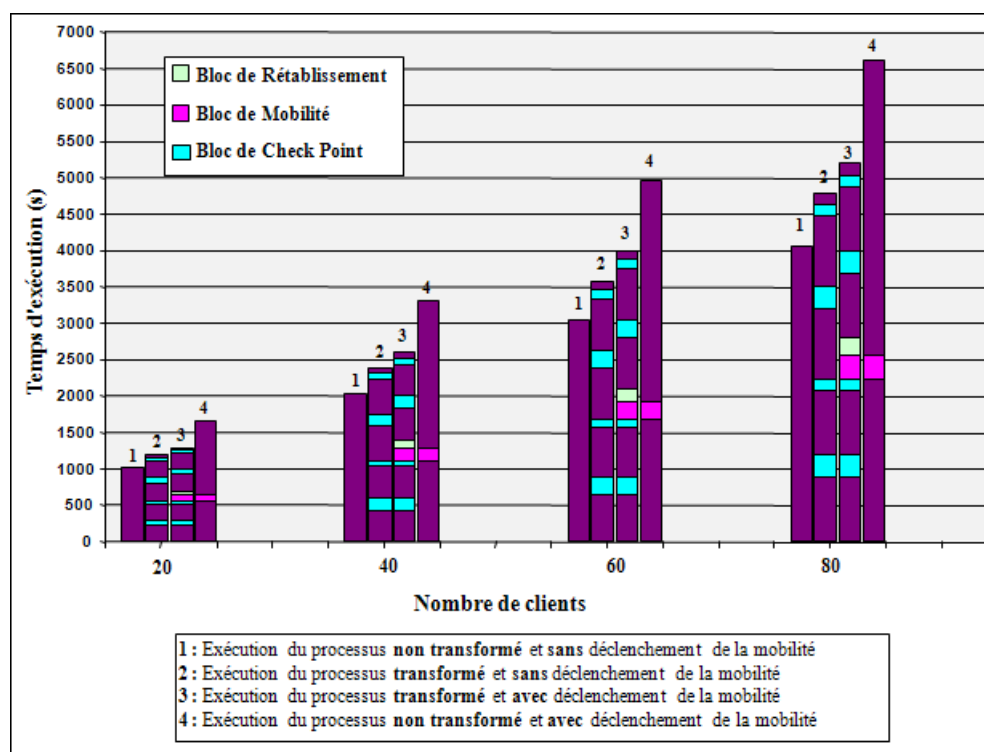


FIG. 3.19 – Evaluation à distance du mécanisme de checkpoint périodique

### 3.4.2.2 Interprétation des résultats des évaluations

La figure précédente présente les résultats d'évaluation de l'exécution du processus BPEL de l'agence de voyage pour chaque valeur du nombre de clients. Dans cette évaluation, nous avons varié le nombre d'instances exécutées en parallèle en variant le nombre de clients. Nous remarquons que plus le nombre de clients (instances) augmente, plus les temps de checkpoint, mobilité et rétablissement augmentent.

En effet, la première valeur (1) correspond au temps d'exécution de ce processus avant d'appliquer les règles de transformation de son code. La deuxième valeur (2) représente le temps d'exécution de la version mobile de notre processus BPEL lorsque la mobilité n'est pas déclenchée. Nous déduisons que le surcoût introduit par la transformation du processus BPEL ne dépasse pas 20% comparé au processus initial. Les expérimentations montrent en plus que le temps d'un checkpoint séquentiel est inférieur à celui d'un checkpoint parallèle. Ce résultat est prédictible étant donné que l'opération de synchronisation des branches parallèles augmente le surcoût du checkpoint parallèle. La troisième valeur (3) de l'expérimentation correspond au temps d'exécution du processus BPEL mobile en présence d'une violation de QoS et en considérant un scénario de reconfiguration de la mobilité forte. Dans ce cas, le surcoût de mobilité correspond au temps nécessaire pour migrer le processus BPEL

vers le nouvel hôte et pour rétablir son exécution, c'est-à-dire charger le checkpoint et rétablir l'instance BPEL. En effet, le surcoût de ces actions ne dépasse pas 2% comparé au temps d'exécution du processus initial. Ainsi, il est clair que le temps d'exécution du processus avec mobilité forte possède un surcoût acceptable comparé à celui du processus originel. Enfin, la quatrième valeur (4) correspond au temps d'exécution du processus BPEL initial en présence d'une violation de QdS et en utilisant une solution de reconfiguration classique, celle-ci consiste en la ré exécution de toute la composition des services Web. Ce temps d'exécution est considérablement important par rapport à celui relatif au processus BPEL transformé utilisant la mobilité forte.

### 3.4.2.3 Synthèse

D'après cette évaluation, nous faisons preuve de la faisabilité de la mobilité forte basée sur le mécanisme de checkpoint périodique. En outre, nous avons fait témoignage que ce mécanisme peut être utilisé comme étant un mécanisme d'auto-réparation, puisqu'il devient possible de rétablir l'exécution des instances suspendues suite à une panne au niveau du nœud hôte du processus d'orchestration. Ainsi, l'exécution de ces instances va être rétablie à partir du dernier checkpoint dans le nouveau nœud. Ce mécanisme peut aussi être employé dans le cas de violation de la QdS du processus d'orchestration lui même. Dans ce cas, un sous-ensemble des instances du processus d'orchestration peut migrer au prochain checkpoint vers une réplique du processus d'orchestration. Ensuite, l'exécution de ces instances sera reprise à partir du point d'interruption.

## 3.4.3 Evaluation de la performance du mécanisme de checkpoint adaptatif

Dans cette expérimentation, nous avons utilisé des Aspects de checkpoint forcé et de checkpoint aux barrières naturelles synchronisées. Dans ce cas ces barrières se localisent à la fin d'une structure « flow ». Nous avons ensuite déployé ces Aspects alternativement 4 fois durant l'exécution de notre processus BPEL transformé. Enfin, nous avons déployé un Aspect de mobilité pour migrer toutes les instances en cours.

### 3.4.3.1 Aspects déployés pour la mise en œuvre de la mobilité forte

Dans le but de mettre en œuvre un mécanisme de checkpoint dynamique, nous avons inclus le code de checkpoint au sein de programmes orientés Aspect. Les Aspects de capture et de recouvrement sont déployés au moment de l'exécution du processus d'orchestration lorsqu'un checkpoint ou une mobilité sont requises,

ou bien lorsque cette exécution est reprise après une action de mobilité. L'Aspect de capture est déployé chaque fois qu'un checkpoint est requis. Une fois déployé, il sauvegarde l'état d'exécution du processus d'orchestration et le transmet au WSCM. Il peut aussi être déployé à tout moment d'exécution et peut agir de plusieurs façons, par exemple, il peut jouer le rôle d'un checkpoint forcé, ou bien d'un checkpoint immédiat et migration ou encore d'un checkpoint au niveau de la prochaine barrière de synchronisation naturelle, etc.

Le choix de l'un de ces Aspects à déployer dépend des conditions de l'exécution courante. Comme présenté dans la figure 3.20, le checkpoint à une barrière de synchronisation naturelle correspond à attendre jusqu'à atteindre une exécution séquentielle (exécution sans branches parallèles) pour effectuer le checkpoint. En effet, à cette position, capturer le checkpoint est simple et requiert seulement l'envoi de l'état courant au WSCM après la mise à jour du compteur de l'activité.

```

<aspect name="CheckpointAtSynchronizationBarriers">
  <partners>
    <partner name="wscm" serviceLinkType="tns:wscmSLT"/>
  </partners>
  <variables>
    <variable name="setrequest" messageType="tns:setStateRequest"/>
    <variable name="setresponse" messageType="tns:setStateResponse"/>
  </variables>
  <pointcut>
    //process[@name="travelPackage"]//assign[@name="ActivityCounterUpdateInSequence"]
  </pointcut>
  <advice type="after">
    <bpws:assign>
      <!-- copy instance identifier to the setrequest variable -->
      <!-- copy activity counters identifier to the setrequest variable -->
      <!-- copy original variables to the setrequest variable -->
    </bpws:assign>
    <bpws:invoke name="invokeWSCMService" partner="wscm"
      portType="tns:WSCM" operation="setState"
      inputVariable="setrequest"
      outputVariable="setresponse">
    </bpws:invoke>
    <!-- End of the Checkpoint Bloc -->
  </advice>
</aspect>

```

FIG. 3.20 – Aspects implémentant un checkpoint aux barrières de synchronisation naturelles

Par contre, comme le montre la figure 3.21, un checkpoint forcé inclut la synchronisation des branches parallèles, avant la capture de l'état d'exécution, afin d'assurer un checkpoint consistant. Le checkpoint au moment de la migration est similaire au scénario de checkpoint forcé, mais il requiert en plus un forçage de la migration en jetant une notification au WSIM et en stoppant l'exécution du processus d'orches-

tration juste après la capture du checkpoint. Toutes ces techniques de checkpoint requièrent un rollback du processus d'orchestration après migration, ce qui implique la ré invocation de tous les services partenaires qui ont été invoqués après le dernier checkpoint et l'instant de mobilité. Pour cela, les services partenaires doivent être sans état afin d'assurer la consistance de l'application. Dans le cas de services partenaires avec état, un Aspect de checkpoint non coordonné peut être créé via le checkpoint forcé en ajoutant des Aspects assurant la fonctionnalité de l'exploitation de messages. En effet, ceci correspond simplement à créer un Aspect qui est activé après les activités `invoke` et `receive` et qui sauvegarde les messages dans le WSCM.

```

<aspect name="ForcedCheckpoint">
  <partners>
    <partner name="wscm" serviceLinkType="tns:wscmSLT"/>
  </partners>
  <variables>
    <variable name="setrequest" messageType="tns:setStateRequest"/>
    <variable name="setresponse" messageType="tns:setStateResponse"/>
  </variables>

  <pointcut>
    //process[@name="travelPackage"]//assign[@name="ActivityCounterUpdate"]
  </pointcut>
  <advice type="after">

    <!-- Beginning of the forced checkpoint Bloc -->
    <bpws:assign>
      <!-- put checkpointIndicator to 1 -->
    </bpws:assign>

    <!-- if synchronization is done -->
    <bpws:switch>
      <bpws:case condition=
        "bpws:getVariableData('NbBranches')= bpws:getVariableData('NbSynch') ">

        <bpws:assign>
          <!-- copy instance identifier to the setrequest variable -->
          <!-- copy NBbranches to the setrequest variable -->
          <!-- copy activity counters identifier to the setrequest
            variable -->
          <!-- copy original variables to the setrequest variable -->
        </bpws:assign>

        <bpws:invoke name="invokeWSCMService" partner="wscm"
          portType="tns:WSCM" operation="setState"
          inputVariable="setrequest"
          outputVariable="setresponse">
        </bpws:invoke>

        <bpws:assign>
          <!-- put the CheckpointIndicator to 0 -->
        </bpws:assign>

      <!-- if synchronization is not yet done -->
      <bpws:otherwise>
        <!-- increment NBSynch -->
        <bpws:while condition="bpws:getVariableData('CheckPointIndicator')
          = 1">
          </empty>
        </while>
      </bpws:otherwise>
    </bpws:switch>
    <!-- End of the Checkpoint Bloc -->
  </advice>
</aspect>

```

FIG. 3.21 – Aspects implémentant un checkpoint forcé

L'Aspect de recouvrement présenté dans la figure 3.22 est déployé après la migration d'un processus d'orchestration. Dans un tel cas, cet Aspect va être déployé dans le nouveau serveur afin de charger le dernier checkpoint capturé pour chaque instance interrompue, l'intégrer dans l'instance du processus d'orchestration et reprendre son exécution. Dans le cas d'un checkpoint non coordonné avec exploitation de messages, l'Aspect de recouvrement doit remplacer les activités invoke et receive par un appel au WSCM pour obtenir les résultats. Pour une communication asynchrone, un problème peut survenir si un checkpoint a lieu après une activité invoke et avant

son receive correspondant. Dans ce cas, le service partenaire invoqué va envoyer la réponse au nœud d'orchestration initial et le processus d'orchestration courant va attendre indéfiniment la réponse. Pour résoudre ce problème, nous proposons de ré invoquer, après migration, toutes les invocations asynchrones non encore accomplies.

```

<aspect name="Recovery">
  <partners>
    <partner name="wscm" serviceLinkType="tns:wscmSLT"/>
  </partners>
  <variables>
    <variable name="getrequest" messageType="tns:getStateRequest"/>
    <variable name="getresponse" messageType="tns:getStateResponse"/>
  </variables>

  <pointcut name="recovery">
    //process[@name="travelPackage"]//receive[@partner="client"]
  </pointcut>
  <advice type="after">

    <bpws:assign>
      <!-- copy instance identifier to the getrequest variable -->
    </bpws:assign>

    <bpws:invoke name="invokeWSCMSERVICE" partner="wscm"
      portType="tns:WSCM" operation="getState"
      inputVariable="getrequest"
      outputVariable="getresponse">
    </bpws:invoke>

    <bpws:switch>
      <bpws:case condition=
        "bpws:getVariableData('instanceID')= bpws:getVariableData
        'getresponse', 'instanceID'">

        <bpws:assign>
          <!-- copy instance identifier from the getresponse variable -->
          <!-- copy activity counters identifier from the getresponse
          variable -->
          <!-- copy NbBranches from the getresponse variable -->
          <!-- copy original variables from the getresponse variable -->
        </bpws:assign>

      </bpws:switch>
      <!-- End of the Recovery Bloc -->
    </advice>
  </aspect>

```

FIG. 3.22 – Aspect de recouvrement

La mobilité d'un processus d'orchestration peut être due à une panne au niveau du nœud hôte ou à une violation de QoS. Dans le premier cas, le processus est brutalement interrompu, ainsi la seule solution correspond à reprendre toutes les instances interrompues dans le nouveau nœud en les ré invoquant à travers le WSIM. Dans le deuxième cas, la mobilité correspond à une décision externe forcée en déployant l'Aspect de mobilité qui interrompt l'exécution de l'instance et propage une exception au WSIM. Cet Aspect peut avoir un pointcut général qui permet son exécution immédiate pour toutes les instances en cours (sans l'attribut condition). Ou bien, il peut inclure un pointcut filtrant les instances qui sélectionne celles concernées par

la mobilité. Dans la figure 3.23, nous avons employé un simple filtre qui sélectionne une instance, mais nous pouvons filtrer plusieurs paramètres comme le nombre de migrations, le temps d'exécution passé, etc. Dans un tel cas, le paramètre utilisé permet une mobilité immédiate en déployant l'Aspect de mobilité.

```
<aspect name="ForcedMobility">
<pointcut condition="getVariableData('instanceID') == '123'">
//process[@name="travelPackage"]//invoke |
//process[@name="travelPackage"]//receive |
//process[@name="travelPackage"]//assign |
//process[@name="travelPackage"]//wait |
//process[@name="travelPackage"]//switch |
//process[@name="travelPackage"]//while
</pointcut>
<advice type="before">
  <bpws:sequence>
    <bpws:throw name="throw" faultName="bpws:forcedTermination"/>
    <bpws:terminate/>
  </bpws:sequence>
</advice>
</aspect>
```

FIG. 3.23 – Aspect de mobilité

### 3.4.3.2 Évaluation à distance du mécanisme de checkpoint adaptatif

La figure 3.24 représente une évaluation à distance de la mobilité forte appliquée en utilisant le mécanisme de checkpoint adaptatif pour plusieurs clients. L'architecture de déploiement étant toujours celle décrite précédemment.

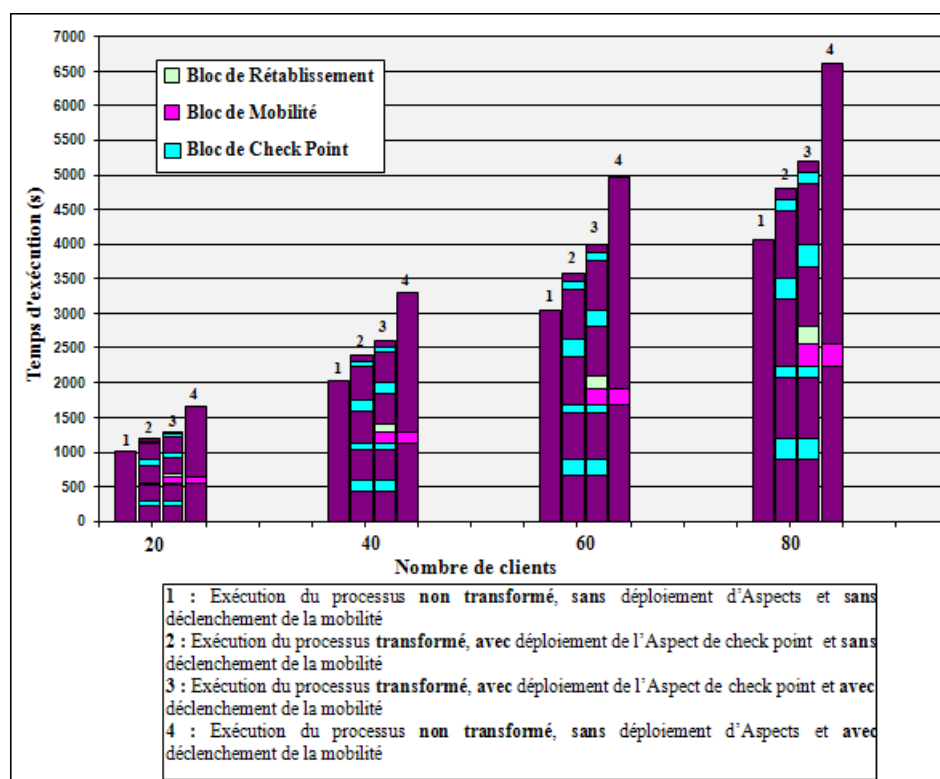


FIG. 3.24 – Evaluation à distance du mécanisme de checkpoint adaptatif

### 3.4.3.3 Interprétation des résultats des évaluations

La figure précédente correspond aux résultats d'évaluation de l'exécution de notre processus BPEL pour différents nombres de clients. Comme dans l'évaluation précédente, nous avons varié le nombre d'instances exécutées parallèlement en variant le nombre de clients. Nous notons que plus le nombre de clients (instances) augmente, plus les temps de checkpoint, mobilité et rétablissement augmentent. En effet, les Aspects déployés sont exécutés pour chaque instance.

La première valeur d'évaluation (1) correspond à l'exécution du processus BPEL initial non transformé et sans déploiement d'Aspect. La deuxième valeur (2) montre le temps d'exécution de ce processus transformé et avec déploiement des Aspects comme décrit précédemment. Nous déduisons que le surcoût introduit par la transformation sans déploiement des Aspects ne dépasse pas 17%, et avec déploiement des Aspects 20% comparé au processus initial. Nous notons aussi la différence dans les temps d'exécution de l'Aspect de checkpoint pour un même nombre d'instances. En fait, le temps nécessaire pour sauvegarder un checkpoint dépend du nombre de branches en cours lors de l'exécution de l'Aspect. Plus le nombre de branches est grand, plus la synchronisation est longue et plus le temps de checkpoint est grand. La troisième valeur (3) dans les résultats de l'évaluation correspond à l'évaluation de

la mobilité de toutes les instances au cours d'exécution. Ceci est réalisé en déployant les Aspects de mobilité dans le premier nœud puis ceux de rétablissement dans le nouveau nœud pour poursuivre l'exécution. Le surcoût des actions de mobilité et de rétablissement ne dépasse pas 2% comparé au temps d'exécution du processus initial. La dernière valeur (4) correspond à la mobilité du processus d'orchestration initial non transformé et sans déploiement d'Aspect de checkpoint ou de rétablissement, mais seulement celui de mobilité. Ainsi, après la mobilité, tout le processus est ré exécuté depuis son début. Cette approche offre de meilleures performances surtout lorsque la mobilité est déclenchée à un niveau plus avancé de l'exécution du processus. Dans cette évaluation, la mobilité est déclenchée à 50% de cette exécution. Nous notons que le surcoût moyen de la mobilité de l'approche est environ 22% et que celui relatif au processus initial est environ 60%. Il est à préciser que le surcoût de la mobilité forte du processus d'orchestration est pratiquement constant quelque soit l'instant de mobilité. Cependant, le surcoût de la mobilité du processus initial est d'une part faible quand la mobilité est déclenchée au début du processus (environ 2%) et d'autre part assez grand lorsque la mobilité a lieu à la fin du processus (100%).

#### 3.4.3.4 Synthèse

Cette évaluation prouve la faisabilité et l'efficacité de la mobilité forte basée sur le mécanisme de checkpoint adaptatif. En effet, grâce aux Aspects, cette solution offre un support de checkpoint des processus d'orchestration assez flexible. Ainsi et dépendant des différentes conditions d'exécution, telles que l'état de l'environnement de déploiement, la QdS requise, etc., le besoin de checkpoint peut être aperçu et immédiatement réalisé en déployant l'Aspect adéquat. De ce fait, toutes les instances (ou une partie) du processus d'orchestration peuvent migrer vers un autre nœud et leur exécution sera reprise à partir du dernier checkpoint.

## 3.5 Conclusion

Nous avons réussi à implémenter les fonctionnalités relatives à la mobilité forte des services Web orchestrés à travers une étude de cas illustrée par un processus BPEL d'une agence de voyage. En outre, nous avons évalué la performance des différents mécanismes de checkpoint, ceux-ci étant utilisés pour valider les politiques d'auto-adaptabilité élaborées selon une démarche d'évaluation dépendant des différents contextes d'exécution des processus d'orchestration. Ainsi, nous sommes parvenus à combler les différentes lacunes posées dans la littérature que ce soit dans le cadre d'applications distribuées ou de services Web.

# Conclusion générale et perspectives

En guise de conclusion, nous avons défini dans le cadre de nos travaux de maîtrise une approche de définition et de validation de politiques d'auto-adaptabilité des services Web orchestrés mobiles. L'approche que nous avons proposée se base principalement sur l'élaboration d'une démarche d'évaluation d'un ensemble de règles de checkpoint et de mobilité opérant dans la phase de planification.

Notre solution a été validée à travers un ensemble d'expérimentations. En effet, nous avons réussi dans un premier temps à implémenter et à évaluer la mobilité forte appliquée pour les processus d'orchestration BPEL. Dans un second temps, nous avons pu montrer la faisabilité et l'efficacité des différents mécanismes de checkpoint et de mobilité utilisés. Dans un troisième temps, nous avons défini et validé, à travers plusieurs scénarios de test, des règles de reconfiguration des processus d'orchestration en tenant compte de leurs contextes d'exécution.

La démarche d'évaluation des actions de reconfiguration élaborée dans le présent travail a été conçue principalement pour supporter la phase de planification du processus d'auto-réparation dans le contexte des services Web orchestrés.

Cependant, notre approche se distingue par rapport à ses précédentes par le fait qu'elle ne se base pas sur l'utilisation d'un mécanisme de mobilité figé. Ce point est primordial pour assurer l'adaptation des processus d'orchestration aux changements de leur environnement d'exécution. Il est à noter également que notre approche définit un ensemble de scénarios de migration ayant pour but essentiel d'éviter la ré exécution de code déjà exécuté, surtout pour les processus de longues durées d'exécution.

Certes la solution que nous avons proposée répond à la plupart des besoins définis au départ, néanmoins elle présente quelques limites notamment en ce qui concerne la définition d'un intervalle de checkpoint dynamique qui tient en compte le dynamisme important de l'environnement d'exécution. L'intérêt de cet intervalle s'aperçoit alors dans le cas de l'utilisation d'un mécanisme de checkpoint adaptatif.

Ainsi, afin de compléter nos travaux, nous visons, à court terme, améliorer notre démarche d'évaluation des politiques d'auto-adaptabilité. Nous souhaitons également dans cette même optique tester notre approche sur des réseaux à plus large

échelle ainsi que sur des environnements de grilles. Ce test permettra en fait de mettre notre solution dans des conditions plus réelles ce qui nous aidera par conséquent à raffiner davantage les règles établies.

Nous visons également, à moyen terme, automatiser notre approche de planification en développant un moteur de génération de règles de reconfiguration.

Finalement nous souhaitons, à long terme, intégrer cette approche de planification dans un processus d'auto-réparation complet afin de la valider.

# Bibliographie

- [1] Activevos designer. <http://www.activevos.com/products-activevos.php>.
- [2] Business process execution language for web services (bpel4ws). <http://www-128.ibm.com/developerworks/library/specification/ws-bpel>.
- [3] Organization for the advancement of structured standards uddi. OASIS. Version 3.0.2, UDDI Spec Technical Committee Draft, Dated 20041019.
- [4] Web service reliable messaging. <http://www.oasis-open.org/committees/wsrn>.
- [5] Ws-addressing. <http://www.w3.org/submission/ws-addressing>.
- [6] Ws-transaction specifications. <http://www-128.ibm.com/developerworks/library/specification/ws-tx>.
- [7] Web services description language (wsdl) 1.1. W3C - World Wide Web Consortium, 2001. W3C Note.
- [8] Soap version 1.2, part 1 : Messaging framework. W3C - World Wide Web Consortium, 2003. W3C Recommendation.
- [9] Requirements for the internationalization of web services. W3C - World Wide Web Consortium, 2004. W3C Working Group Note.
- [10] G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. The cactus worm : Experiments with dynamic resource discovery and allocation in a grid environment. *The International Journal of High Performance Computing Applications*, 15(4) :345–358, 2001.
- [11] L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi, , and M. Segnan. Towards self-diagnosing web services. In *Proceedings of IFIP/IEEE Int. Workshop on Self-Managed Systems et Services (SELFMAN 2005)*, 2005.
- [12] L. Baresi, C. Ghezzi, and S. Guinea. Towards self-healing service compositions. In *Proceedings of the First Conference on the Principles of Software Engineering*, 2004.
- [13] L. Baresi, C. Ghezzi, and S. Guinea. Towards self-healing composition of services. In *Contributions to Ubiquitous Computing*, volume 42, pages 27–46. Springer, 2007.

- [14] S. Bouchenak. *Mobilité et Persistance des Applications dans l'Environnement Java*. PhD thesis, Institut National Polytechnique de Grenoble, France, 2001.
- [15] F. Cappello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Nri, and O. Lodygensky. Computing on large-scale distributed systems : Xtrem web architecture, programming models, security, tests and convergence with grid. *Future Generation Computer Systems*, 21(3) :417–437, 2005.
- [16] Anis Charfi. *Aspect-Oriented Workflow Languages : AO4BPEL and Applications*. PhD thesis, Université de Darmstadt, Allemagne, Décembre 2006.
- [17] R. Y. de Camargo, A. Goldchleger, F. Kon, and A. Goldman. Checkpointing bsp parallel applications on the integrate grid middleware : Research articles. *Concurrent Computing : Practice and Experience*, 18(6) :567–579, 2006.
- [18] O. Ezenwoye and S. M. Sadjadi. Trap-bpel : A framework for dynamic adaptation of composite services. In *Proceedings of the International Conference on Web Information Systems and Technologies*, March 2007.
- [19] David Fauthoux. *Des grains aux aspects, proposition pour un modèle de programmation orientée aspect*. PhD thesis, Université de Paul Sabatier , Toulouse III, Mai 2004.
- [20] R. Fernandes, K. Pingali, and P. Stodghill. Mobile mpi programs in computational grids. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 22–31, New York, USA, 2006. ACM.
- [21] S. Frechette and D. R. Avresky. Method for task migration in grid environments. In *Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications*, pages 49–58, Washington, DC, USA, 2005.
- [22] P. Garbacki, B. Biskupski, and H. E. Bal. Transparent fault tolerance for grid applications. In *Proceedings of the European Grid Conference on Advances in Grid Computing*, volume 3470 of *Lecture Notes in Computer Science*, pages 671–680. Springer, 2005.
- [23] S. A. Gurguis and A. Zeid. Towards autonomic web services : achieving self-healing using web services. *SIGSOFT Softw. Eng. Notes*, 30(4) :1–5, 2005.
- [24] R. B. Halima, M. Jmaiel, and K. Drira. A qos-driven reconfiguration management system extending web services with self-healing properties. In *In Proceedings of the 16th IEEE International Workshops on Enabling Technologies Workshop on Information Systems et Web Services*, Paris, France, June 2007. IEEE Computer Society.

- [25] R. B. Halima, M. Jmaiel, and K. Drira. A qos-oriented reconfigurable middleware for self-healingweb services. Beijing, China, September 2008. IEEE Computer Society.
- [26] M. JURIC. *BPEL and JAVA*. April 2005.
- [27] J. Kovacs. Transparent parallel checkpointing and migration in clusters and clustergrids. *International Journal of Computational Science and Engineering*, 4(3) :171–181, 2007.
- [28] P. Lemarinier, A. Bouteiller, G. Krawezik, and F. Cappello. Coordinated checkpoint versus message log for fault tolerant mpi. *Int. J. High Perform. Comput. Netw.*, 2(2) :146–155, 2004.
- [29] S. Marzouk, A. Jmal Maâlej, I. Bouassida Rodriguez, and M. Jmaiel. Periodic checkpointing for strong mobility of orchestrated web services. In *Proceedings of the International Workshop on Self Healing Web Services (SHWS 2009) in conjunction of the 7th IEEE International Conference on Web Services (ICWS 2009)*, LosAngeles, California, USA,, 2009. ACM.
- [30] S. Modafferi, E. Mussi, and B. Pernici. Sh-bpel : a selfhealing plug-in for ws-bpel engines. In *Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, pages 48–53, New York, USA, 2006. ACM.
- [31] Mariusz Momotko, MichałGajewski, André Ludwig, Ryszard Kowalczyk, Marek Kowalkiewicz, and Jian Ying Zhang. Towards adaptive management of qos-aware service compositions. *Multiagent Grid Syst.*, 3(3) :299–312, 2007.
- [32] G. K. Mostéfaoui, N. C. Narendra, Z. Maamar, and P. Thiran. On modeling and developing self-healing web services using aspects. In *Proceedings of the Second International Conference on COMMunication System softWARE and MiddlewaRE (COMSWARE 2007)*. IEEE computer society, 2007.
- [33] C. Peltz. Web service orchestration and choreography, a look at wsci and bpel4ws. *Web Services Journal*, 3(7) :30–35, July 2003.
- [34] B. Pernici and A. M. Rosati. Automatic learning of repair strategies for web services. In *Proceedings of the Fifth European Conference on Web Services*, pages 119–128, Washington, DC, USA, 2007. IEEE Computer Society.
- [35] S. Sattanathan, P. Thiran, N. C. Narendra, G. K. Mostéfaoui, and Z. Maamar. On the enhancement of bpel engines for self-healing composite web services. In *Proceedings of the 2008 International Symposium on Applications and the Internet*, pages 33–39, Washington, DC, USA, 2008. IEEE Computer Society.
- [36] S. S. Vadhiyar and J. J. Dongarra. Self adaptivity in grid computing. *Concurrency Computation : Practice and Experience*, 17(2-4) :235–257, 2005.

---

# Évaluation de Politiques d'Auto-Adaptabilité basées sur la Mobilité des Services Web Orchestrés

---

**Afef JMAL épouse MAÂLEJ**

---

**الخلاصة:** تلعب خدمات الواب دورا رئيسيا في تنفيذ التطبيقات الموزعة التي غالبا ما تكون مستوعبة في بيئات على نطاق واسع. كما أن موارد هذه البيئات تتميز بالدينامية والتقلب، و هو ما يخلق مخاطر تدهور جودة الخدمات (QoS)، و في بعض الأحيان عدم توفر هذه الخدمات. في هذا السياق، يجب أن توضع تدابير لإصلاح الذات و للتكيف الذاتي في حيز التنفيذ لضمان استمرارية و جودة الخدمات. لتحقيق هذا المنال، قمنا بتطوير آليات تعتمد على نقاط الاستئناف والتنقل في إطار خدمات الواب المنسقة. من ثم، فإننا قيمنا مساهمة استخدام هذه الآليات في حالات مختلفة من خلال دراسة تجريبية. وأخيرا، اعتمدنا على هذا التقييم لتوليد سياسات التكيف الذاتي لتنسيق خدمات الواب في إطار دينامية بيئة تنفيذها. تتولد هذه السياسات في مرحلة التخطيط من عملية الإصلاح الذاتي.

**Résumé :** Les services Web demeurent aujourd'hui un acteur principal dans la mise en œuvre des applications distribuées. De telles applications sont souvent déployées sur des environnements à large échelle. Le dynamisme et la volatilité des ressources de ces environnements engendrent des risques de dégradation des QoS voire même des pannes des services déployés. Dans ce contexte, des actions d'auto-réparation et d'auto-adaptabilité doivent être mises en place afin d'assurer les QoS requises. Pour ce faire, nous avons commencé par implémenter des mécanismes de checkpoint et de mobilité dans le cas des services Web orchestrés. Ensuite, nous avons évalué l'apport de l'utilisation de ces mécanismes dans différentes situations à travers une étude expérimentale. Enfin, nous avons exploité ces évaluations pour définir des politiques d'auto-adaptabilité des processus d'orchestration face au dynamisme de leur environnement d'exécution. Ces politiques sont générées dans la phase de planification du processus d'auto-réparation.

**Abstract:** Web services are still a major player in the implementation of distributed applications. Such applications are often deployed on large-scale environments. The dynamism and volatility of the resources in these environments create risk of QoS degradation or even failure of deployed services. In this context, measures of self-healing and self-adaptability must be taken to ensure the required QoS. For that, we started by implementing various checkpoint and mobility mechanisms in the case of orchestrated Web services. Then, we evaluated the contribution of the use of these mechanisms in different situations through an experimental study. Finally, we used these evaluations to generate policies of self adaptability of orchestration processes face to the dynamism of their execution environment. These policies are generated in the planning phase of the self-healing process.

**المفاتيح:** عملية تنسيق خدمات الواب ، سياسة، التكيف الآلي، التخطيط، قوة التنقل، نقطة استئناف.

**Mots clés :** processus d'orchestration, politique, auto-adaptabilité, planification, mobilité forte, checkpoint.

**Key-words:** orchestration process, policy, self-adaptability, planning, strong mobility, checkpoint.