AMSELEM Jonathan
BALI Rami
FAYOLLE Samuel
GALEA Nicolas

Master 1 Informatique

Virtualisation d'orchestration de services

Rapport final du Travail d'Etude et de Recherche

26 Mai 2008

Encadrant : Philippe COLLET

SOMMAIRE

Sommaire 2

1. Introduction 3

1.1. Contexte 3
1.2. Objectif et démarche 3

2. Cahier des charges 5

2.1. Fonctionnalités 5
2.2. Contraintes non fonctionnelles 6

3. Organisation Prévisionnelle du projet 7

3.1. Priorités 7
3.2. Planning prévisionnel 7

4. Travail réalisé 8

4.1. Etude des différentes technologies 8

4.1.1. WS-BPEL 8
4.1.2. Fractal 8
4.1.3. SCA 9
4.1.4. ActiveBPEL 9
4.1.5. Projet Astro 9
4.1.6. Dynamo (dynamic monitoring) 10

4.2. Construction des ponts 10
4.3. Surveillance à la demande 11
4.4. Surveillance par notification 11
4.5. Architecture résultante 12
4.6. Tests et Validation 14

5. Bilan 15

5.1. Problèmes rencontrés 15

5.1.1. Génération des ponts 15
5.1.2. Déploiement 15
5.1.3. Surveillance par interrogation du moteur 15
5.1.4. Surveillance par notification 16
5.1.5. Contrôleur et intercepteur 16

5.2. Changements dans le planning 17
5.3. Etat d'avancement du projet 19
5.4. Perspectives 20

Annexes 21

Bibliographie : 29

1. INTRODUCTION

1.1. CONTEXTE

Face à la complexité grandissante des systèmes logiciels, les chercheurs et ingénieurs continuent à imaginer de nouveaux paradigmes. L'approche basée sur la construction d'architectures orientées Services est l'une des plus prometteuses pour gérer cette complexité. Il est communément reconnu que le concept de « Service » facilite l'intégration des systèmes logiciels en masquant la complexité des technologies sous-jacentes et en fournissant une vue globale du système.

Les [1]Web Services[5] définissent une manière standard d'interagir avec des applications distantes en utilisant les technologies du Web. La complexité augmente encore lorsqu'une tâche est basée sur des Web Services dépendant les uns des autres, une application doit donc définir une coordination de ces services.

WS-BPEL[3] , (Business Process Execution Language for Web Services) est une spécification du groupe Oasis qui définit un modèle de coordination et d'orchestration, c'est-à-dire l'enchaînement automatisé, de web-services entre eux. Ainsi, il est possible de définir entièrement un processus métier qui fait interagir des services issus de systèmes différents. Le moteur d'exécution ActiveBPEL[4] assure l'évaluation et l'orchestration des processus Web.

Parallèlement, les architectures orientées service se rapprochent du monde des composants logiciels. Un exemple est la spécification SCA (Service Component Architecture), qui propose un modèle de programmation pour la construction d'applications à base de composants. SCA intègre les orchestrations WS-BPEL comme des composants qui peuvent être assemblés avec d'autres composants SCA.

L'INRIA en collaboration avec France Télécom R&D, a développé en 2002, un modèle de composants hiérarchiques et dynamiquement reconfigurables appelé FRACTAL[1]. Un ancien groupe de TER a étendu ce modèle en implémentant une boîte à outils, FRACTAL-WS, qui permet de réaliser des ponts entre nos composants FRACTAL et des Web Services.

1.2. OBJECTIF ET DEMARCHE

L'objectif de ce TER est d'améliorer FRACTAL-WS en permettant la virtualisation d'orchestration de Web Services, ce qui permettrait par la suite d'orchestrer des composants fractal avec d'autres. Nous avons pour cela commencé à étudier WS2Fractal et Fractal2WS qui font partie de notre boîte à outils, ainsi que le code expérimental mis à notre disposition. Cette prise en main s'est effectuée en virtualisant une orchestration simple récupérant la version du serveur d'application. En parallèle, nous avons étudié les travaux existants avec SCA pour avoir une vision sur la concurrence. Il nous est apparu assez rapidement que l'étude du moteur d'orchestration ActiveBPEL serait un élément qui allait grandement influencer notre architecture finale, nous avons donc consacré une part d'étude importante à celui-ci.

---

[1] [1], [2], [3], [4], [5] : voir annexe page 21.

Notre analyse préliminaire du sujet nous avait confronté à deux visions d'architectures : soit un seul composant d'orchestration par fichier BPEL (comme le fait SCA), soit plusieurs composants pour un fichier BPEL. La seconde architecture avait plus d'avantages que la première car dans l'hypothèse de faisabilité nous aurions pu connaître le déroulement d'une orchestration avec exactitude. L'étude d'ActiveBPEL nous a permis d'avoir une nouvelle vision.

Deux stratégies se sont distinguées pour surveiller les orchestrations lancées. La première consiste à interroger le moteur d'orchestration afin d'obtenir des informations sur l'orchestration et les processus en cours. La seconde est basée sur l'enregistrement d'un client comme observateur afin d'être averti automatiquement.

Dans un premier temps nous allons présenter notre cahier des charges, puis notre organisation du projet, notamment nos priorités ainsi que notre planning.

Ensuite, nous décrirons le travail réalisé, avec le détail de chaque fonctionnalité.

Et enfin, sous forme de bilan, nous reviendrons sur les problèmes rencontrés et leurs solutions, puis nous présenterons les suites prévues de ce TER.

2. CAHIER DES CHARGES

Afin de définir clairement les fonctionnalités à ajouter dans la boite à outils FractalWS et pouvoir suivre l'avancement de leur réalisation, un cahier des charges a été établi et présenté lors de la soutenance intermédiaire.

2.1. FONCTIONNALITES

Le principal objectif est de représenter une orchestration (un fichier BPEL) par un composant Fractal contenant, pour chaque Web Service partenaire, un autre composant (proxy) permettant les communications avec ce Web Service (Figure 1). Ainsi une application à base de composants Fractal peut lancer des orchestrations BPEL qui orchestrent, par exemple, d'autres composants déployés comme des Web Services.

Les proxys crées nous permettent de détecter les appels que peut faire le moteur d'orchestration sur les Web Services partenaires mais pas de savoir précisément quelle exécution de l'orchestration l'utilise. Nous devons donc identifier chaque instance de l'orchestration présente sur le moteur, à travers nos composants. Une piste est de représenter chaque exécution de l'orchestration par un composant dédié (Figure 2).

Ensuite, comme les orchestrations s'effectuent sur un moteur distant, nous n'avons qu'une vision très limitée de leur avancement. Les composants les représentants doivent permettre d'observer l'état d'avancement.

Enfin, nous devons utiliser des orchestrations dans un moteur de messagerie instantanée (Amui), également développé par les équipes précédentes, pour piloter ses différentes entités.

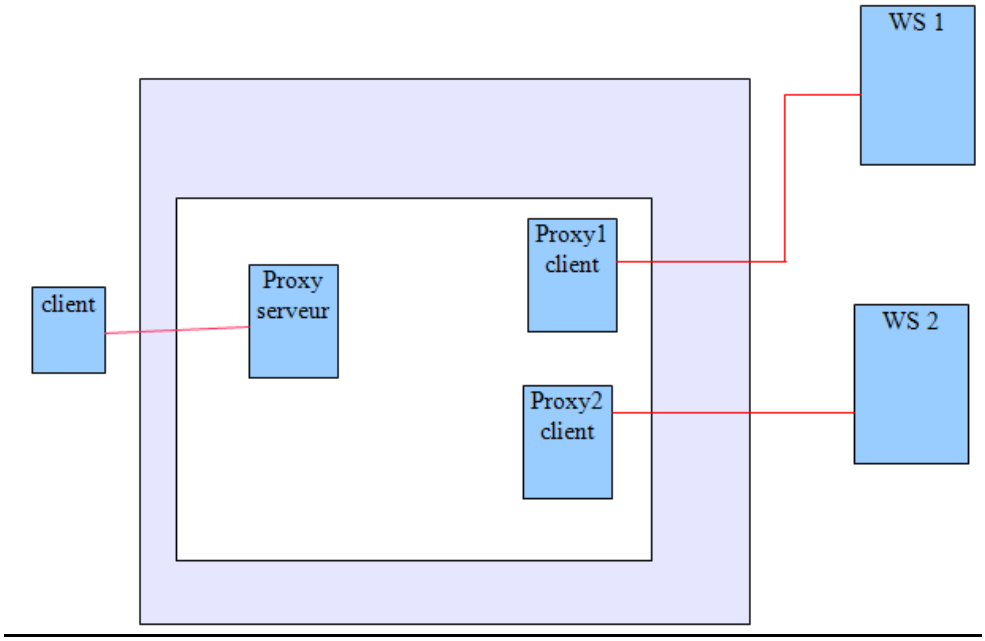


Figure 1 : Architecture 1-1

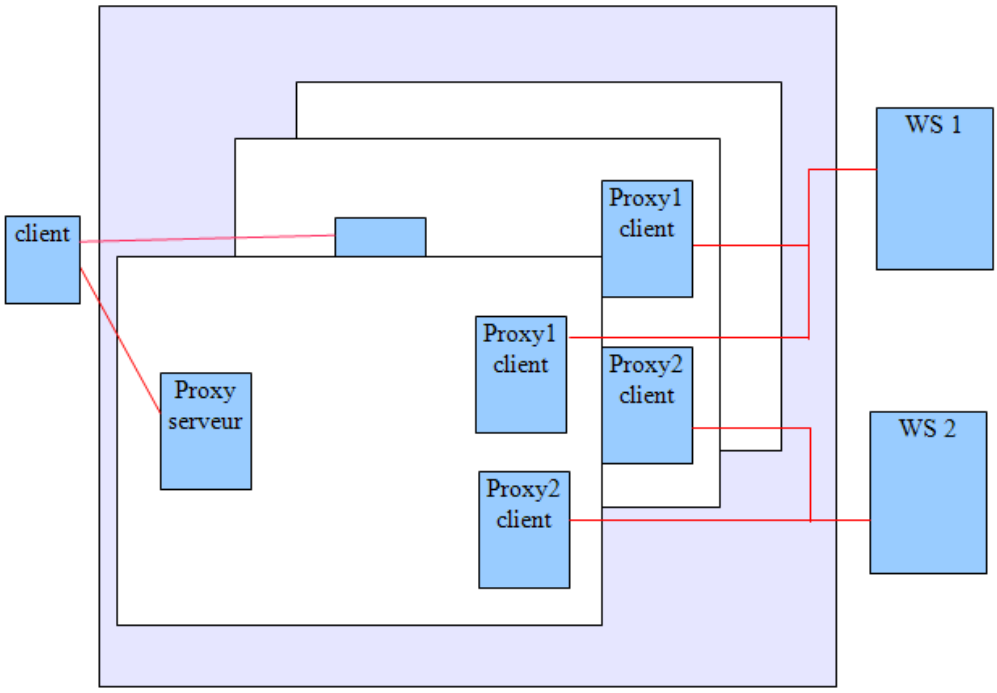


Figure 2 : Architecture 1-N

2.2. CONTRAINTES NON FONCTIONNELLES

Du fait du grand nombre de technologies utilisées, les contraintes liées aux plates-formes logicielles sont importantes.

- Linux, Windows XP
- Java : Sun JDK 5
- Apache Tomcat 5.x : Conteneur de Servlet.
- Apache Axis 1.4 : Une implémentation du protocole SOAP utilisé pour la transmission de messages.
- FractalWS 1.0 : Boite à outils réalisant des ponts entre le monde des Web Services et les composants Fractal.
- Fractal Julia 2.5 : Implémentation en Java du modèle Fractal.
- ActiveBpel 3.1 : Moteur d'orchestration fonctionnant dans un serveur Tomcat.
- WSBPEL 2.0 : Langage de définition d'orchestrations de Web Services.

3. ORGANISATION PREVISIONNELLE DU PROJET

3.1. PRIORITES

En fonction de l'étude du sujet et des différents éléments mis à notre disposition, nous avons pu définir les priorités suivantes pour les fonctionnalités à fournir:

Niveau 1 :

Comprendre le code d'expérimentation fourni précédemment et savoir comment le compléter puis le déployer à plus grande échelle, afin d'utiliser des orchestrations à partir de composants Fractal ou de Web Services représentés par des composants.

Niveau 2 :

Il existe plusieurs niveaux de représentation possible pour les orchestrations :

- Un composant par fichier BPEL et donc par type d'orchestration.
- Un composant par exécution de l'orchestration BPEL.

Niveau 3 :

Permettre de consulter l'état d'avancement des orchestrations en cours d'exécution ou après.

Niveau 4 :

Intégration et validation du code créé dans le logiciel de communications instantanées AMUI.

3.2. PLANNING PREVISIONNEL

Vu le grand nombre de technologies utilisées, nous avons commencé ensemble par étudier Fractal et BPEL en lisant les spécifications fournies par les éditeurs et en s'exerçant avec les tutoriaux disponibles puis nous avons rédigé le cahier des charges.

Nous avons définis deux groupes :

- **Groupe A** : Rami et Nicolas doivent analyser, déployer et stabiliser le code fourni
- **Groupe B** : Samuel et Jonathan doivent étudier l'implémentation de BPEL dans SCA puis la documentation d'ActiveBpel et Fractal pour trouver une solution pour l'architecture 1-N et la surveillance de l'état du système.

A la fin de cette seconde phase, nous devons pouvoir représenter une orchestration par un seul composant Fractal composite contenant les proxys nécessaires.

Ensuite, l'équipe B devra implémenter la solution choisie au problème des composants par exécution d'une orchestration. L'équipe A procèdera en même temps à l'élaboration de tests pour cette partie.

Enfin, nous réaliserons ensemble l'intégration dans AMUI et la rédaction du rapport final.

4. TRAVAIL REALISE

4.1. ETUDE DES DIFFERENTES TECHNOLOGIES

Étant donné la complexité du contexte nous environnant, nous devions commencer par l'assimilation de technologies variées afin d'avoir le minimum de maîtrise pour pouvoir travailler efficacement sur le sujet proposé. Nous avons aussi dû nous pencher sur le code fournit par Annabelle Mercier qui s'occupait de la génération des proxies clients et serveurs de notre application.

4.1.1. WS-BPEL

Le langage BPEL4WS, (Business Process Execution Language for Web Services) ou tout simplement BPEL, est une spécification du consortium OASIS supportée par IBM, Microsoft, et BEA. Elle remplace les précédentes spécifications XLANG de Microsoft, et WSFL (Web Services Flow Language) d'IBM. Le modèle de procédé BPEL forme une couche au-dessus de WSDL(Web Service Definition Language). Il définit la coordination des interactions entre l'instance d'orchestration et ses Web Services partenaires.

Les procédés dans BPEL exportent et importent les fonctionnalités en utilisant des interfaces de services web uniquement. BPEL permet de modéliser des procédés exécutables : chacun spécifie l'ordre d'exécution des activités constituant le procédé, des partenaires impliqués dans le procédé, des messages échangés entre ces partenaires, et le traitement de fautes et d'exceptions spécifiant le comportement dans les cas d'erreurs ou d'exceptions. Le grand avantage de BPEL est la possibilité de décrire les interactions entre les logiques métiers des différentes entreprises à travers les services web. Les éléments du procédé BPEL sont : les liens de partenaires, les activités et les données (voir annexe).

Plus particulièrement nous nous sommes concentrés sur les ensembles de corrélations qui identifient de manière unique un processus (exécution d'une orchestration). Des données de corrélation sont ajoutées automatiquement aux messages transmis entre le moteur et les Web Services partenaires pour identifier le processus correspondant à ces messages.

Ces données auraient pu nous servir pour identifier le processus qui utiliserait nos proxies serveur mais nous avons trouvé une meilleure solution à l'aide d'un processus de surveillance (expliqué par la suite).

4.1.2. FRACTAL

Le modèle de composants Fractal est un modèle général dédié à la construction, au déploiement et à l'administration (e.g. observation, contrôle, reconfiguration dynamique) de systèmes logiciels complexes, tels les intergiciels ou les systèmes d'exploitation (voir annexe).

4.1.3. SCA

SCA (SERVICE COMPONENT ARCHITECTURE) est un ensemble de spécifications visant à simplifier la création de services, indépendamment de leur implémentation, dans le cadre d'architectures orientées services (SOA).

Ce modèle de programmation d'architecture à base de composants permet de construire et d'organiser des systèmes complexes. Lors de notre étude de SCA, nous avons trouvé que chaque orchestration était traduite en un nouveau composant. Cependant les exécutions de ces orchestrations ne sont pas elles-mêmes représentées par des composants.

Ainsi nous ne pouvions pas nous inspirer de cette spécification pour implémenter la solution 1-N. De plus nous n'avions que des informations sur les services fournis et non sur l'utilisation en elle même de la spécification, ce qui ne nous apportait rien de plus que les technologies dont nous disposions déjà. C'est pourquoi nous ne sommes pas allés plus avant dans nos recherches sur SCA.

4.1.4. ACTIVEBPEL

Comme nous dépendons beaucoup du serveur d'application ActiveBPEL, et dans le but de récupérer des informations sur le moteur, dans un premier temps nous avons dû effectuer un nombre important de recherches dans la documentation officielle.

Pour tester nos virtualisations, nous utilisons l'interface d'administration d'ActiveBPEL qui nous renseigne sur les états des orchestrations. Nous avons alors compris qu'il était intéressant de chercher dans cette direction, afin de mettre en place une stratégie qui puisse récupérer toutes ces informations. De plus, nous avons trouvé des systèmes d'écouteurs qui permettent de surveiller le moteur d'exécution ainsi que chaque exécution d'orchestration.

Comme la version d'ActiveBPEL utilisée est open source, toute la partie documentation, forum, aide technique est payante et donc inaccessible pour nous. Notre principale source fut alors l'étude des classes avec leur JavaDoc, afin de dénicher celles qui pourraient nous intéresser.

4.1.5. PROJET ASTRO

Astro est un projet Européen qui consiste à réaliser du monitoring de Web Services.

ASTRO développe des instruments soutenant l'évolution et l'adaptation de processus métiers distribués pendant leur cycle de vie, de la conception au temps d'exécution, c'est-à-dire, les instruments automatisés qui soutiennent la spécification et la conception de services et permettent la composition de processus d'affaires distribués.

L'idée étant que de tels instruments devraient fournir un support automatisé, transparent et centré sur l'utilisateur sur le cycle de vie entier du processus métier, de l'analyse à l'exécution.

Cependant, malgré les promesses de cette piste nous n'avons pas poursuivi dans cette direction à cause de l'absence de code source et de documentation.

4.1.6. DYNAMO (DYNAMIC MONITORING)

Dynamo-AOP est un framework qui utilise ActiveBPEL afin d'avoir une surveillance dynamique des orchestrations. Les caractéristiques annoncées sont intéressantes, on peut en effet ajouter des propriétés pour les activités qui communiquent avec des Web Services. Ces propriétés peuvent être des pré et/ou des post conditions, ainsi on peut par exemple être averti de la reprise d'une activité.

Toutes les propriétés de contrôle ont trois paramètres :

- *priority* : représente «l'importance» de la règle et peut être un entier allant de 1 à 5. Chaque processus peut alors définir une valeur seuil qui rend la surveillance des propriétés actives ou non, ce qui permet une évolution dynamique de la quantité d'activités effectuées pour le suivi du processus.
- *validity* : définit à quel moment dans le temps une propriété de surveillance doit être prise en compte.
- *trusted providers* : qui représente une liste de services pour lesquels la surveillance n'est pas nécessaire.

Toutefois nous n'avons pas pu extraire des informations utiles dans les sources fournies car bien que le langage JAVA soit utilisé dans une grande partie du projet, tout ce qui concerne la partie gestion des processus est une couche au dessus d'ActiveBPEL, réalisée en partie en AspectJ[2].

4.2. CONSTRUCTION DES PONTS

La virtualisation d'une orchestration BPEL consiste en la génération automatique des composants la représentant. Nous avons créé pour cet effet un outil BPEL2Fractal à partir du code expérimental fourni par Mlle Mercier.

Ce code expérimental permettait de déployer une nouvelle orchestration à l'aide du fichier BPEL et des définitions en WSDL correspondantes aux Web Services utilisés, puis de générer tous les proxies à l'aide des outils Fractal2WS, WS2Fractal et de nouvelles classes. En effet, pour chaque partenaire impliqué dans l'orchestration, un proxy serveur est généré pour recevoir les messages en provenance d'ActiveBpel et un proxy client pour envoyer les messages vers le moteur. Enfin, un composant englobant est généré pour contenir ces proxies et faire le bon assemblage.

Notre travail consistait en partie à comprendre la logique du processus de génération automatique. Ensuite, une fois que les proxies étaient prêts, nous avons déployé les proxies clients comme Web Services et transformé le fichier BPEL initial pour que les partenaires initiaux soient remplacés par ces nouveaux Web Services. Enfin, nous avons redéployé automatiquement cette nouvelle orchestration à la place de la première.

[2] AspectJ est un langage de programmation orienté aspect basé sur le Java. Il étant sa syntaxe par quelques mots clefs. Le tissage d'aspect est géré de manière statique.

4.3. SURVEILLANCE A LA DEMANDE

Nous avons remarqué que l'interface d'administration d'ActiveBPEL permet l'affichage de l'état d'une orchestration, ainsi que de chaque exécution (et chaque activité). Nous avons donc étudié le code source de la page et avons remarqué qu'elle utilisait des JSP. Grâce à ces derniers, nous avons trouvé les classes très intéressantes d'ActiveBPEL qui peuvent nous permettre de suivre l'état des exécutions et de recueillir des informations sur celles-ci.

Nous avons alors concentré nos recherches sur ActiveBPEL (étude approfondie de la documentation, questions sur les forums), et lors des nombreux déploiements dans Axis et Tomcat, nous avons trouvé un WebService de ActiveBPEL(BpelEngineAdmin). Ce dernier permet d'obtenir plusieurs informations intéressantes sur les exécutions d'une orchestration, comme les méthodes getProcessList ou getProcessDetail qui permettent de récupérer les détails d'un processus donné ou la liste de tous les processus. Nous avons également trouvé d'autres méthodes qui autorisent les actions directement sur ceux-ci (suspendProcess, resumeProcess, terminateProcess).

Afin de pouvoir l'utiliser au mieux dans notre architecture, nous avons effectué un composant Fractal proxy à l'aide de WS2Fractal, connecté à un composant intermédiaire. Ce dernier implémente l'analyseur syntaxique sur le document XML obtenu par le biais d'une méthode du Web Service d'ActiveBPEL, getProcessState, qui renvoie donc toutes les données de l'instance d'orchestration (état, arbre des activités et leurs états, variables...).

Grâce à ce système que nous avons mis en place, les clients peuvent interroger l'état du système quand ils le souhaitent : pour cela il suffit de se brancher sur le contrôleur (du composant englobant) et de faire appel aux méthodes désirées de l'interface. Le client possède ainsi une référence vers l'interface du contrôleur, il peut alors à tout moment et sans limite interroger le moteur d'orchestration.

4.4. SURVEILLANCE PAR NOTIFICATION

Nous voulions offrir aux clients la possibilité de s'enregistrer comme observateur afin d'être notifié des différents changements d'état du moteur d'ActiveBpel ainsi que pour chaque exécution d'orchestration.

Le Web Service BpelEngineAdmin, dont nous parlions précédemment, offre cette possibilité à l'aide de méthodes tel que addProcessListener et addEngineListener qui prennent en paramètre entre autre un PID (identificateur d'un processus d'orchestration) et une URL du Web Service représentant l'écouteur que l'ont veut ajouter. Nous avons ensuite fait une recherche dans les sources d'ActiveBPEL à notre disposition, afin de trouver une classe implémentant le concept d'écouteur, comme on a pu le voir en Java. Nous avons trouvé une classe se rapprochant de notre idée, qui est AeRemoteDebugImpl. Ce qui nous a intéressé, c'est qu'elle contient des classes internes qui représentent les écouteurs d'ActiveBPEL . A l'intérieur de ces classes, on peut s'apercevoir que l'URL du Web Service mis par le client et servant d'écouteur, est ajoutée pour le moteur d'orchestration. A chaque fois qu'un événement est relevé, ce dernier va parcourir la liste de ses écouteurs et les notifier.

De nombreuses informations peuvent ainsi être transmises, on peut par exemple récupérer le type de l'événement qu'il y a eu et en déduire l'état d'un processus.

Une fois cette étude faite, nous avons implémenté l'interface IaeEventHandler dans notre composant intermédiaire afin d'en faire un écouteur et nous avons déployé le composant comme Web Service. Nous pouvons ainsi réceptionner les informations envoyées par le moteur d'orchestration depuis ce composant.

4.5. ARCHITECTURE RESULTANTE

Les résultats de nos recherches sur le fonctionnement du moteur d'orchestration nous ont permis de choisir l'architecture avec un seul composant représentant une orchestration et de l'étoffer avec des composants dédiés à la surveillance du moteur.

Un client peut se brancher sur le composant d'orchestration pour :

- Démarrer de nouvelles orchestrations (interfaces serveur).
- Lister les orchestrations déjà lancées (interfaces de contrôle).
- Recueillir des informations sur ces orchestrations ou s'enregistrer comme observateur (interfaces de contrôle).

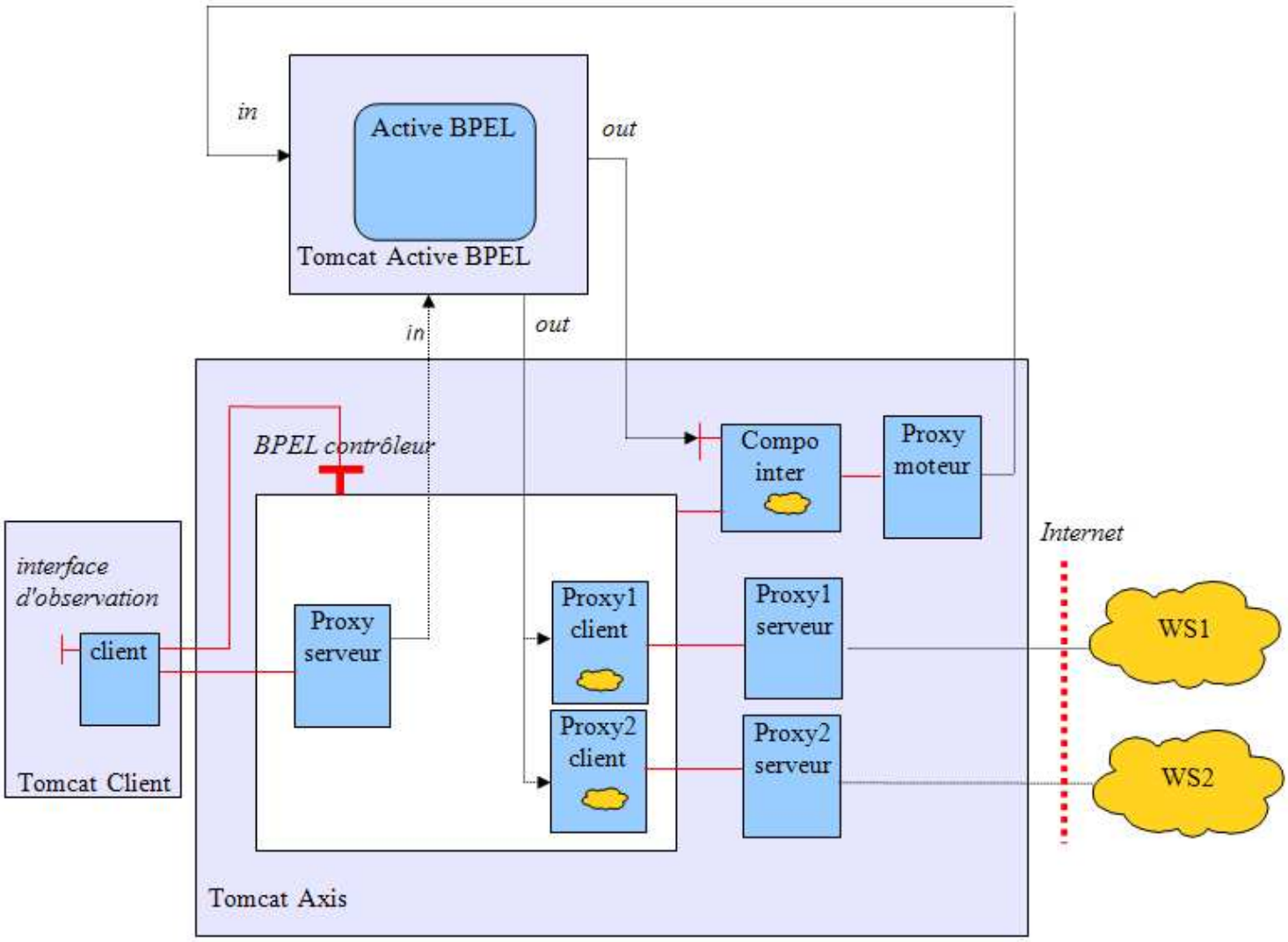


Figure 3 - Architecture finale

Nous avons donc adapté la membrane du composant d'orchestration pour y intégrer un contrôleur et un intercepteur.

Le contrôleur permet à un composant quelconque de demander des informations sur les orchestrations et le moteur. Un composant ayant une interface d'un certain type que nous avons défini peut également s'enregistrer comme observateur des événements du moteur ou des orchestrations.

Des intercepteurs sont placés sur les interfaces serveur du composant d'orchestration. Ces derniers permettent d'identifier les composants clients qui démarrent de nouvelles orchestrations et les ajoutent dans une liste de clients. Ainsi, un client peut surveiller toutes les orchestrations ou celles lancées par client précis (par exemple un composant A peut surveiller toutes les orchestrations du composant B).

Le système de surveillance est centralisé dans le composant intermédiaire. En effet, le contrôleur que nous avons décrit précédemment délègue tout le travail à ce composant. Celui-ci est relié au proxy du moteur d'orchestration et est donc capable de chercher toutes les informations sur les orchestrations.

Ce composant s'occupe aussi de la notification des observateurs. Il gère la correspondance entre les clients et les PID des orchestrations qu'ils ont lancées. Ensuite, pour chacune de ces orchestrations, il garde une liste d'observateurs. Ce composant intermédiaire est déployé de plus comme un Web Service et enregistré comme écouteur du moteur d'orchestration. C'est lui qui reçoit les événements en provenance du moteur puis notifie les clients correspondants.

Grâce aux deux manières différentes de surveillance du moteur (push and pull : le client peut être notifié automatiquement ou aller demander quand il le souhaite les informations), nous avons un mécanisme de surveillance qui est plus abouti que ce que nous avions prévu.

4.6. TESTS ET VALIDATION

Notre projet étant long et très technique, nous avons décidé avec M. Collet de ne pas effectuer des tests unitaires avec JUnit mais uniquement des tests ad-hoc avec des composants Fractal.

Afin de valider notre architecture générale nous avons procédé aux opérations suivantes :

- tests de notre virtualisation avec plusieurs Web Services différents

- tests de toutes les méthodes implémentées dans le contrôleur (pour la surveillance par interrogation du moteur). Nous avons fait des clients qui interrogent plusieurs fois et de toutes les façons possibles le moteur, et enregistrent les résultats dans un fichier de log

- de la même façon, afin de tester les fonctionnalités de la surveillance par écouteurs, nous avons fait plusieurs clients qui s'enregistrent comme observateurs et qui reçoivent alors tous les évènements du moteur (et enregistrent sous fichiers de log)

- tests unitaires de virtualisation d'orchestrations différentes (une dizaine de BPEL différents avec indépendamment tous les types d'activités BPEL)

- tests d'intégration avec les activités ensembles

- tests avec deux BPEL différents et des partner link différents

- tests avec deux orchestrations BPEL qui partagent des Web Service en commun

- automatiser les lancements des tests

Il était prévu en dernière priorité, d'intégrer notre architecture dans le logiciel de communication instantanée AMUI. Avec l'accord de notre encadrant, nous avons préféré effectuer tous ces tests afin de bien valider notre virtualisation, et reporter l'intégration dans AMUI pour le stage qui débutera au mois de juin.

5. BILAN

5.1. PROBLEMES RENCONTRES

5.1.1. GENERATION DES PONTS

Lors de la génération des ponts nous déployons nos proxies comme Web Services et les utilisons comme partenaires, dans le BPEL, à la place des Web Services originaux. Les méthodes définies dans le WSDL de ces nouveaux Web Services doivent donc avoir exactement les mêmes signatures que les originaux hors ce n'est pas le cas pour les noms de paramètres. Nous utilisons donc, pour l'instant, des Web Services avec les noms de paramètres par défaut (in0,in1) mais nous corrigerons ce problème lors du stage en étudiant comment paramétrer WSDL2JAVA (Axis) qui effectue le déploiement.

La première approche pour résoudre le problème du 1-N n'est en fait pas réaliste. Pour créer un composant par exécution de l'orchestration, il faut modifier le fichier BPEL déployé pour qu'il utilise les nouveaux proxies crées et déployés pour chaque exécution. Cela revient à refaire tout le travail de déploiement du BPEL et de génération des proxies à chaque exécution et donc avoir autant d'orchestrations déployées dans le moteur que d'exécution du BPEL de départ. Nous nous sommes donc concentrés sur les moyens d'identifier l'instance d'orchestration responsable des appels vers les Web Services (Corrélations en BPEL et Écouteurs du moteur).

5.1.2. DEPLOIEMENT

Nous avons au départ utilisé un seul Tomcat qui devait contenir le moteur ActiveBPEL et nos composants Fractal déployés comme Web Services dans Axis. Nous avons eu beaucoup d'erreurs, incompréhensibles au départ, dues au fait que des librairies sont en commun entre le moteur d'orchestration et axis.

Nous avons donc opté pour une plateforme à base de deux serveurs Tomcat.

5.1.3. SURVEILLANCE PAR INTERROGATION DU MOTEUR

Nous avons rencontré des problèmes avec cette approche car lors de la récolte des informations, nous recevions un objet qui n'avait pas le même type que celui défini dans le fichier WSDL du BpelAdmin. Nous avons fait des recherches pour trouver le désérialiseur correspondant mais en vainc. Nous avons alors tenté de faire un proxy avec ce fichier WDSL à l'aide de WSDL2Java.

Nous avons rencontré plusieurs problèmes, notamment la définition de type="tns3:Throwable" qui était référencée mais non définie. De plus, un namespace était faux dans le fichier WDSL de base, les deux types org.activebpel.rt.bpel.impl.list.AeProcessInstanceDetail et org.activebpel.rt.bpel.impl.AeProcessInstanceDetail se mélangeaient. Nous avons modifié ce fichier WSDL pour corriger ces problèmes, et le nouveau composant créé, intégré dans notre composite, fonctionne correctement.

Grâce à la méthode qui nous renvoie un fichier XML avec toutes les données de l'instance d'orchestration nous pouvons à présent surveiller l'orchestration.

5.1.4. SURVEILLANCE PAR NOTIFICATION

L'implémentation des écouteurs nous a demandé une étude approfondie d'ActiveBPEL. En effet ce qui s'est révélé être un simple problème de déploiement de la part des concepteurs du moteur d'orchestration, nous a fait perdre énormément de temps, principalement au niveau du choix de notre architecture finale.

Lors de nos premiers tests, le seul message d'erreur qui nous apparaissait était une InvocationTargetException qui tenait sur une ligne et rien d'autre, aucune trace dans Tomcat ne montrant d'Exception, et de même aucun message d'erreur au niveau d'ActiveBPEL.

Devant le peu d'informations fournies sur le moteur d'orchestration (une recherche des classes concernant les écouteurs dans ActiveBPEL sur Google nous ramène uniquement à la page Wiki de notre TER !) et un forum officiel restreint aux utilisateurs ayant acheté une version commerciale d'ActiveBPEL (de même pour les tutoriaux parlant du sujet), nous avons choisi de recompiler nous même le moteur d'orchestration et ainsi suivre pas à pas comment était géré le système d'écoute. ActiveBPEL étant composé d'environ 2500 classes, une compilation prend en moyenne 1 min 30, ce qui ne nous a pas facilité les choses. Nous avons ainsi pu remarquer que le problème se situait au niveau d'un chargement dynamique d'une classe. La classe qui était censée être chargée permettait de définir la manière dont allait être envoyée les données à l'écouteur, soit en RPC, soit en local. Comme tout ceci se faisant à l'exécution, la classe n'était pas trouvée et aucune exception ne pouvait donc être capturée.

Le fait d'avoir recompilé nous même les classes avec l'IDE Eclipse nous a permis de résoudre cela, car la classe chargée dynamiquement était dans le CLASSPATH et a ainsi pu être compilée et mise dans un jar existant. Le fait de redéployer ActiveBPEL avec les scripts qu'ils fournissent a fait que l'application fonctionnait correctement, mais nous n'avions pas encore la justification réelle du problème qu'il y avait eu.

Après plusieurs recherches nous nous sommes rendus compte qu'en fait les concepteurs d'ActiveBPEL ne déployaient pas complètement l'application ainsi une simple modification du script d'installation a suffit à résoudre le problème convenablement.

5.1.5. CONTROLEUR ET INTERCEPTEUR

Dans un premier temps comme nous n'arrivions pas à faire fonctionner les écouteurs, nous avons développé en parallèle une solution alternative, afin de récupérer les informations pertinentes pour le suivi des processus. A l'aide des outils Fractal contrôleur et intercepteur nous avons intercepté les appels démarrant l'orchestration en implémentant la méthode « pre » de celui-ci.

Puis, en déléguant le traitement au contrôleur de la récupération des informations par un composant à part, qui stockait dans une hashmap les PIDs en clef et le document regroupant les informations de ce processus en valeur, nous remettions à jour la table à intervalles réguliers en implémentant une méthode run() .

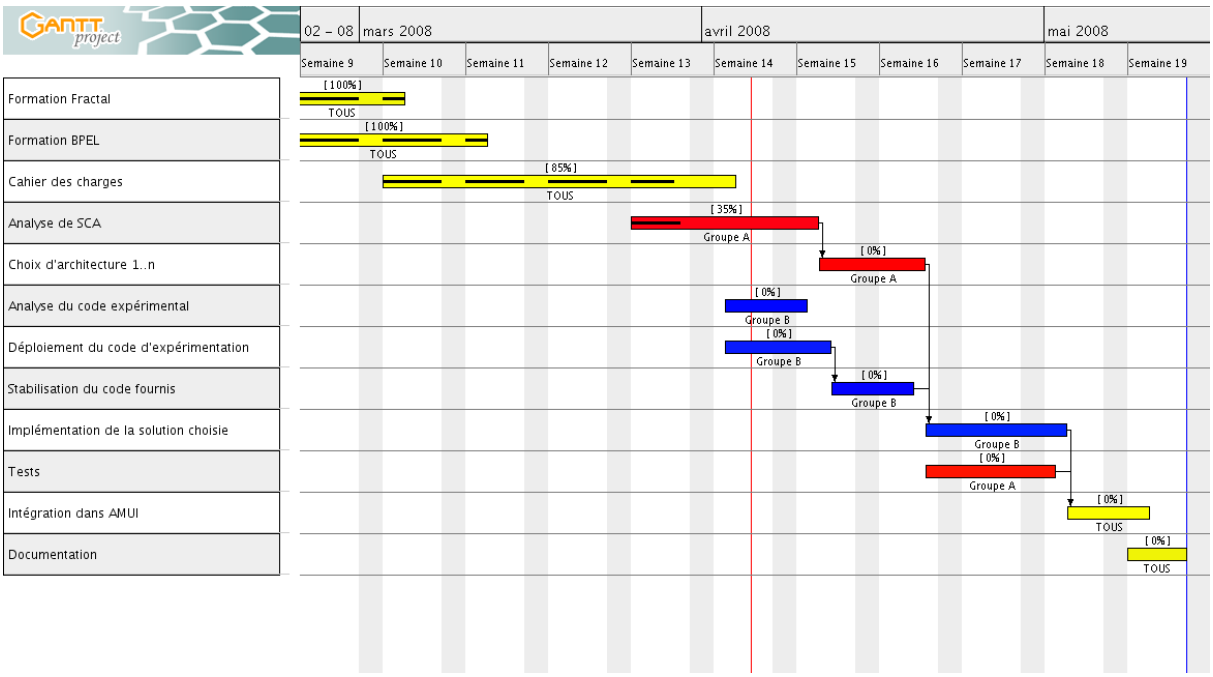
Un problème soulevé par la solution proposée est que l'intercepteur ne nous permet d'effectuer une tâche qu'avant l'appel d'une méthode où aprés qu'elle ait fini son travail. Ainsi si nous effectuions le traitement des PID dans la méthode « pre », nous étions obligés de faire un sleep pour attendre que le processus commence, pour qu'il puisse y avoir un PID, ce dont nous n'étions pas sûrs finalement, car il était possible que l'orchestration ne soit tout simplement pas démarrée. De plus, si plusieurs clients démarraient leurs orchestrations dans un temps inférieur au temps du sleep, il se pouvait que nous intervertissions le suivi des orchestrations par rapport aux clients.

De l'autre côté, si nous exécutions le traitement des PIDs dans la méthode « post », alors il fallait attendre que l'orchestration se termine, ce qui dans la plupart des cas arrive bien trop tard, puisque certaines orchestrations peuvent durer plusieurs semaines.

Nous sommes donc repartis sur la piste des écouteurs pour ne plus avoir à gérer le problème de l'interception des processus.

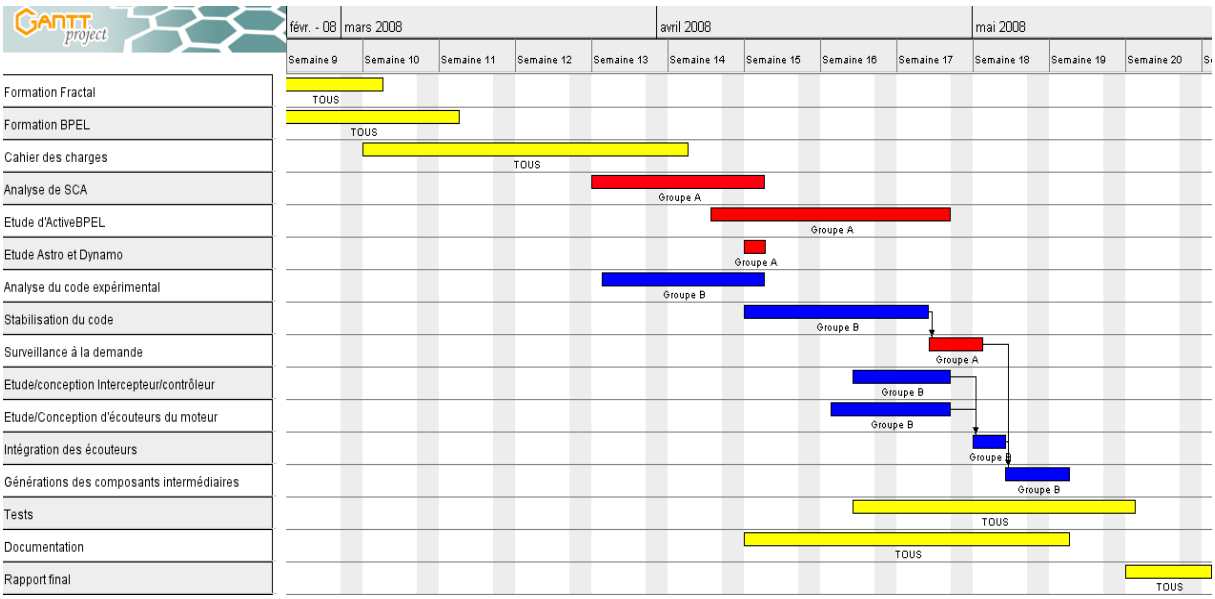
Grace à l'architecture que nous avons adoptée, plusieurs orchestrations peuvent être gérées en même temps. En conséquence un problème de synchronisation nous est apparu. En effet nous nous sommes aperçus que si plusieurs orchestrations arrivaient en même temps il pouvait y avoir une duplication des informations dans la table des processus. Pour y remédier nous avons intégré la vérification de mise à jour dans chaque accesseur.

5.2. CHANGEMENTS DANS LE PLANNING



Il est cependant, à dénoter quelques différences entre les diagrammes prévu et effectif. Cela se justifie par le fait que notre travail est orienté recherche. En effet, les domaines sur lesquels nous choisissions de nous concentrer tout au long de notre cheminement, furent essentiellement induits par les impasses et les voies que les premiers sujets abordés nous ont ouvertes. N'ayant pas le recul nécessaire pour apprécier avec finesse l'évolution de notre travail nous avons cependant conservé la

ligne directrice de notre première estimation, et les changements apportés sont relativement mineurs.



Comme nous pouvons le remarquer sur le diagramme ci-dessus, l'analyse de SCA à continué, mais, ne nous menant nulle part, nous nous sommes penchés sur l'étude d'ASTRO et de Dynamo afin d'y trouver l'inspiration d'un modèle d'architecture ainsi que des technologies utilisables pour notre projet.

Parallèlement nous avons analysé le code expérimental afin de pouvoir le stabiliser comme prévu. C'est ici que réside la différence majeure entre ce que nous avions prévu et le plan que nous avons suivi. L'architecture 1-n - un composant par instance d'orchestration de service - se révélant trop incertaine pour être réalisable en premier, nous avons préféré commencer par la surveillance qui était nécessaire et identique dans les deux cas (1-1 et 1-N) en gardant une architecture 1-1.

Une fois le cap du choix d'architecture atteint, nous savions plus clairement comment nous allions réaliser nos objectifs, mais le suivi du flux d'exécution demeurait alors toujours un problème. Nous pouvions de façon certaine mais laborieuse créer un mécanisme de surveillance à la demande en nous contentant d'intercepteurs et de contrôleurs, mais d'un autre côté l'utilisation d'écouteurs, pourvu que nous arrivions à les exploiter, nous offraient un moyen simple, propre et efficace d'arriver à nos fins. La suite se conçoit aisément une fois le succès dans l'exploitation des écouteurs établis puisque les finitions correspondent à ce qui était annoncé dans le diagramme précédent à cela prêt que le manque de temps ne nous a pas permis l'intégration dans AMUI, qui sera effectué lors du stage.

Voici la répartition des taches en fonction des quatre membres :

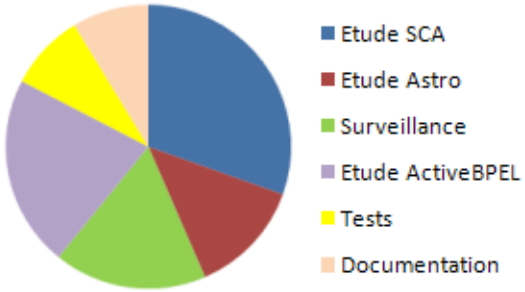


Figure 4 : Amselem Jonathan

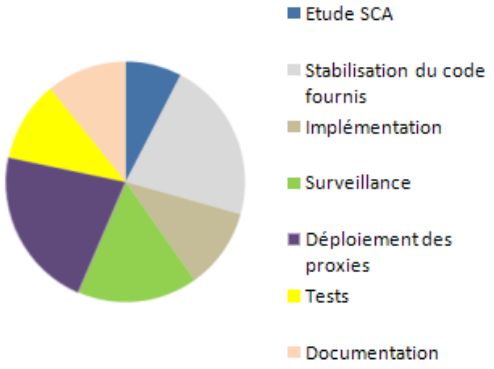


Figure 5 : Bali Rami

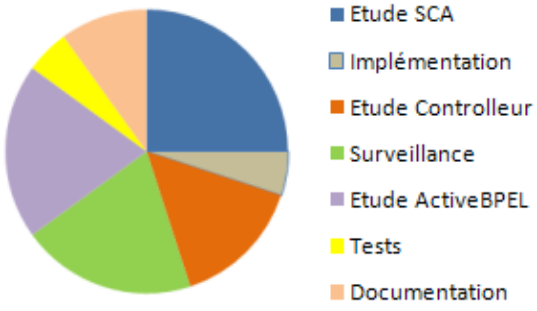


Figure 6 : Fayolle Samuel

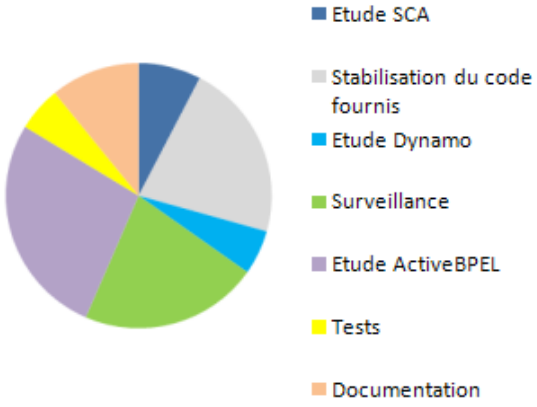


Figure 7 : Galea Nicolas

5.3. ETAT D'AVANCEMENT DU PROJET

Tout au long du projet, nous avons veillé à respecter les points importants mentionnés dans le cahier des charges, en particulier les fonctionnalités.

Comme cité dans la partie précédente, quelques changements ont été opérés au niveau des priorités. Nous pensions que notre principale priorité était l'implémentation d'une architecture 1-N c'est à dire pour chaque exécution d'orchestration la construction de nouveaux proxies afin d'avoir le maximum d'informations sur ces processus. Cependant lors de nos recherches sur les moyens de surveillance offerts par ActiveBPEL, nous avons expérimenté les écouteurs (vu dans la partie 3.4) et réussi à identifier chaque processus grâce à cette méthode. Nous avons alors mis en place une architecture 1-1 qui possède tous les avantages de celle prévue au départ, et qui permet d'avoir en plus de nouvelles informations sur le moteur d'orchestration et sur chaque processus. Ces fonctionnalités de surveillance sont devenues prioritaires, nous avons alors dû adapter notre planning en conséquence.

Nous avons abouti à une plate-forme expérimentale cohérente, mais nous n'avons pas intégré notre architecture finale dans AMUI. Ce sera le sujet d'un stage proposé par nos encadrants pour trois d'entre nous, dont le point de départ est notre prototype expérimental.

5.4. PERSPECTIVES

Ce travail d'étude et de recherche nous a permis d'apprendre de nouvelles technologies, de découvrir de nouveaux outils et de consolider notre expérience dans la gestion de projet.

Le fait de participer à la réalisation d'un projet en cours de développement, nous a donné une idée plus précise du travail dans le monde professionnel.

Cela nous a permis entre autre de réaliser l'importance de la communication dans la hiérarchie de développement (relations encadrant-étudiants). Le fait d'avoir pu élaborer un cahier des charges sur une période plus longue que de coutume, nous a montré la difficulté de conception et de suivi d'un planning avec des outils open source ou en cours de développement. La cohésion du groupe et les relations humaines sont un facteur important que nous avons appris à gérer tout au long de notre TER en nous réunissant tous les jours dans les salles informatiques du Petit Valrose, et en nous retrouvant en dehors du travail afin de mieux nous connaitre. (plus la journée couscous chez Rami).

ANNEXE A : DEFINITIONS

[1]Composant Fractal :

Le modèle de composants Fractal a été défini par France Télécom R&D et l'INRIA depuis 2001. Il se présente sous la forme d'une spécification et d'implémentations dans différents langages de programmation comme Java, C, C++, SmallTalk ou les langages de la plate-forme .NET. Les principales caractéristiques du modèle Fractal sont motivées par l'objectif de pouvoir construire, déployer et administrer des systèmes complexes tels que des intergiciels ou des systèmes d'exploitation. Le modèle est ainsi basé sur les principes suivants :

- **composants composites** (i.e. composants qui contiennent des sous-composants) pour permettre d'avoir une vue uniforme des applications à différents niveaux d'abstraction.
- **composants partagés** (i.e. sous-composants de plusieurs composites les englobant) pour permettre de modéliser les ressources et leur partage, tout en préservant l'encapsulation des composants.
- **capacités d'introspection** pour permettre d'observer l'exécution d'un système.
- **capacités de (re)configuration** pour permettre de déployer et de configurer dynamiquement un système.

La base du développement Fractal réside dans l'écriture de composants et de liaisons permettant aux composants de communiquer. Un composant Fractal est ainsi une entité d'exécution qui possède une ou plusieurs interfaces. Une *interface* est un point d'accès au composant. Une interface implante un *type d'interface* qui spécifie les opérations supportées par l'interface. Il existe deux catégories d'interfaces : les interfaces *serveurs* (en rouge sur la Figure 8)- qui correspondent aux services fournis par le composant -, et les interfaces *clients* (en vert) qui correspondent aux services requis par le composant.

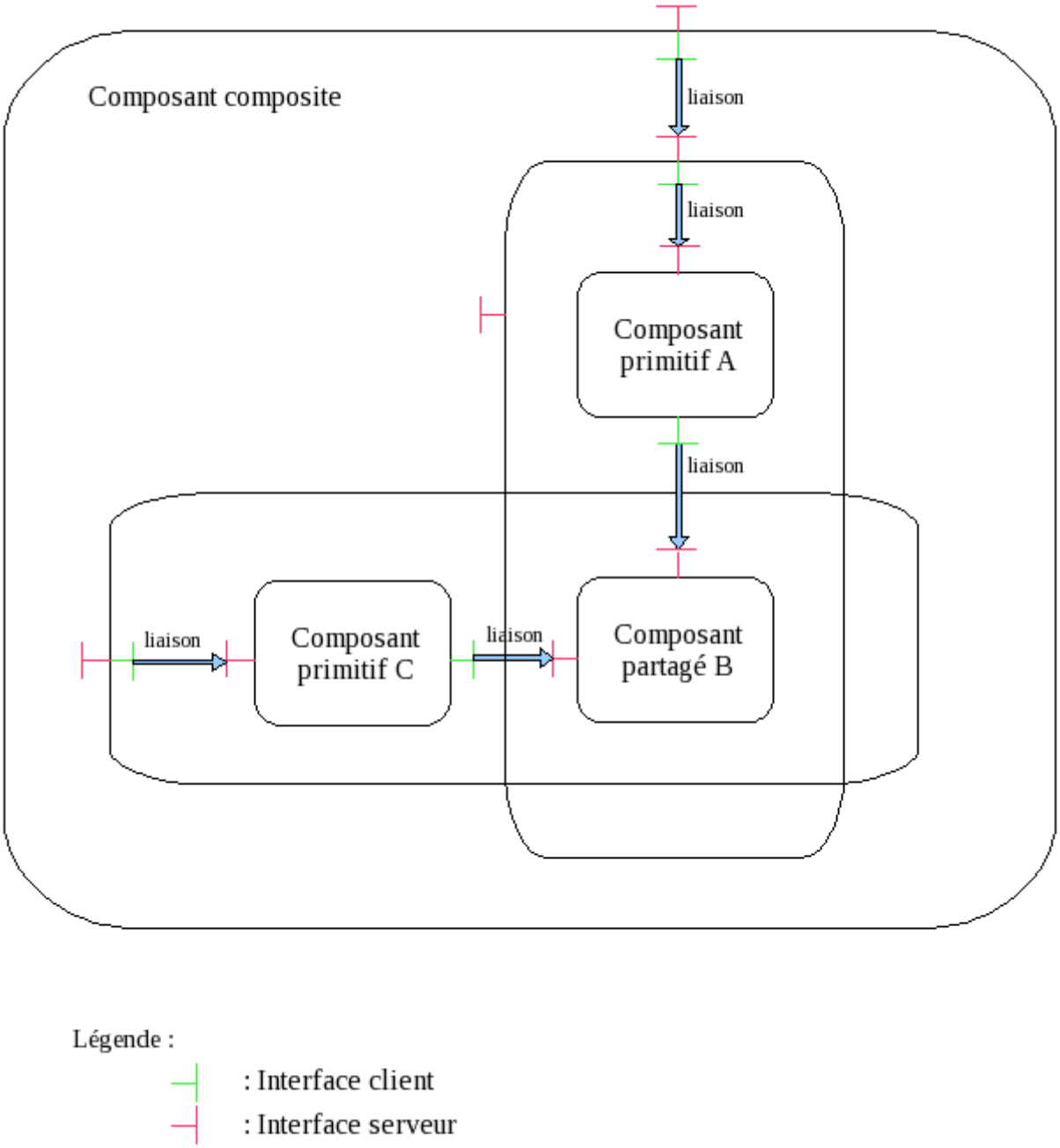


Figure 8 : Exemple de composants Fractal

FRACTAL-WS :

Fractal WS est une boîte à outils qui vise à fournir des moyens pour rendre compatible n'importe quel composant Fractal avec la technologie des Web Service et réciproquement. L'une des interfaces fournies par un composant Fractal peut être transformée en un Web Service, les rendant ainsi accessibles par le biais de protocoles Web. D'autre part, tout Web service (externe) peut être consulté par un composant Fractal, en utilisant un proxy dédié. Des services ou des composants sont

générés afin que la passerelle soit opérationnelle. Fractal WS est basé sur l'implémentation de référence “Julia”, en Java , du modèle de composants Fractal.

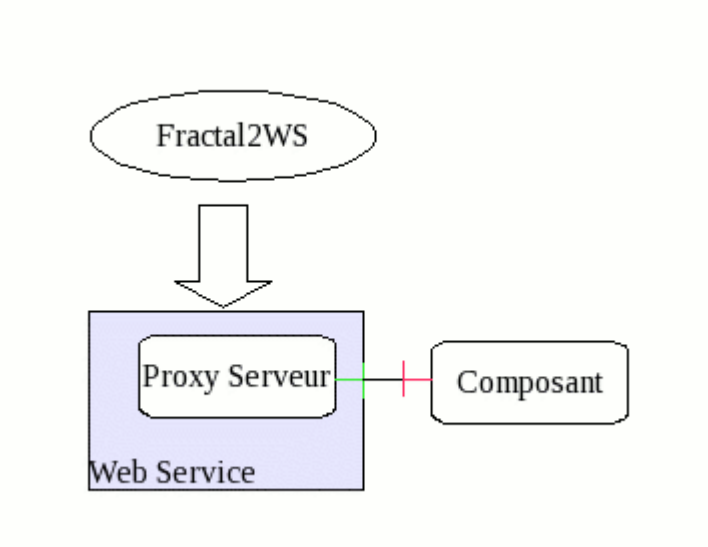


Figure 9 : Fractal2WS

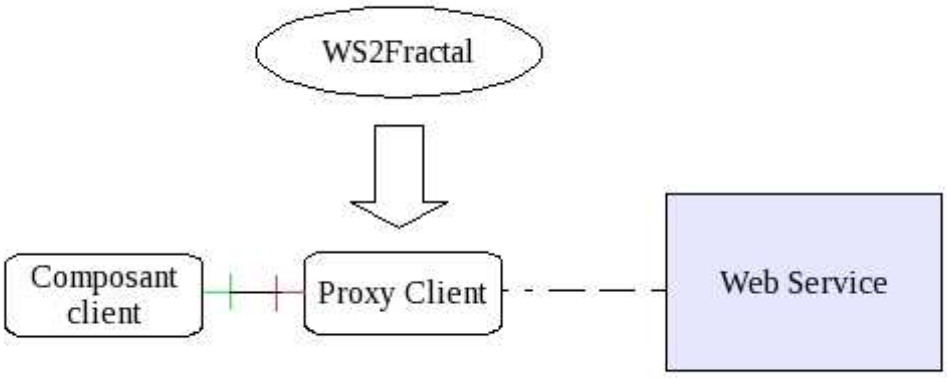


Figure 10 : WS2Fractal

[2]Orchestration :

Il s'agit de la partie qui décrit les règles de distribution des différentes tâches à exécuter aux différents Web Services. Autrement dit, le compositeur en train d'« orchestrer » distribue consciemment ses instruction selon les services qu'il souhaite obtenir.

[3]BPEL :

BPEL et une spécification visant à créer une infrastructure d'orchestration pour services Web.

Les liens de partenaires :

Un lien de partenaire (partnerLink) correspond au service avec lequel le procédé échange des informations. Le lien de partenaire représente la relation de conversation entre deux procédés partenaires. Chaque lien de partenaire est typé par un partnerLinkType, il est chargé de définir le rôle que joue chacun des deux partenaires dans une conversation.

Les activités :

Le procédé dans BPEL est constitué d'activités liées par un flot de contrôle. Ces activités peuvent être basiques ou structurées.

Les activités basiques sont :

L'invocation d'une opération dans un service Web, l'attente d'un message d'une source externe, la réponse à une source externe, l'attente un certain temps, la copie des données d'une place à l'autre, le lancement d'une erreur d'exécution. La terminaison de l'instance de service en entier.

Les activités structurées sont composées d'autres activités basiques et structurées. Les types d'activités structurées sont : la définition d'un ordre d'exécution, l'acheminement conditionnel, les boucles, l'attente d'arrivée d'évènements, l'acheminement parallèle, le regroupement des activités afin qu'elles soient traitées par le même gestionnaire d'erreur et l'invocation des activités de compensation par le gestionnaire d'erreur, pour défaire l'exécution déjà complétée d'un regroupement d'activité.

Les données :

Le procédé dans BPEL a un état, cet état est maintenu par des variables contenant des données. Ces données sont combinées afin de contrôler le comportement du procédé. Elles sont utilisées dans les expressions et les opérations d'affectation. Les expressions permettent d'ajouter des conditions de transition ou de jointure au flot de contrôle. L'affectation permet de mettre à jour l'état du procédé, en copiant les données d'une variable à une autre ou en introduisant de nouvelles données en utilisant les expressions.

Dans BPEL il n'y a pas de flot de données, BPEL se sert des variables pour passer une donnée d'une activité à une autre, à l'aide de l'affectation (contenant les données).

- Process : regroupe une série d'activités de partenaires et d'éléments d'extensibilité.
- Partners : ils sont définis comme un ensemble de liens de partenaires (partnerLink).

[4]ActiveBPEL :

L'outil de conception Active BPEL open source et le premier environnement de développement intégré pour le déploiement, la construction, et les testes des applications basée sur la spécification BPEL.

[5]Web Services :

Un Service Web est un programme informatique permettant la communication et l'échange de données entre applications et systèmes hétérogènes dans des environnements distribués. Il s'agit donc d'un ensemble de fonctionnalités exposées sur Internet ou sur un Intranet, par et pour des applications ou machines, sans intervention humaine, et en temps réel.

- **[6]WSDL : (**Web Services Description Language)

  Il s'agit d'une normalisation regroupant la description des éléments permettant de mettre en place l'accès à un service réseau.

  Le **WSDL** décrit une Interface publique d'accès à un service Web. C'est une description basée sur le XML qui indique « comment communiquer pour utiliser le service »; le Protocole de communication, et le format de messages requis pour communiquer avec ce service.

- **[7]SOAP : (**Simple Object Access Protocol**)**

  Il s'agit d'un protocole permettant la transmission de messages entre objets distants, ce qui veut dire qu'il autorise un objet à invoquer des méthodes d'objets physiquement situés sur un autre serveur.

[8]SCA : SERVICE COMPONENT ARCHITECTURE

SCA fournit deux niveaux de modèle :

- **Un modèle d'implémentation** : Construire des composants qui fournissent et consomment des services ;
- **Un modèle d'assemblage** : Construire une application métier à forte valeur ajoutée en liant entre eux un ensemble de composants.

Ainsi, SCA insiste sur une séparation forte entre l'implémentation des services et leur assemblage. Le modèle SCA se veut agnostique vis-à-vis :

- Des technologies d'implémentation des composants de service. Il inclut entre autre les technologies d'implémentation suivantes : Java, C++, BPEL, XQuery ou SQL.

- Des technologies d'exposition et d'invocation des composants de services (même si WSDL et les interfaces java sont mis en avant).

SCA permet de décrire des services et leur assemblage indépendamment de toutes considérations techniques d'implémentation.

Le modèle d'implémentation :

L'élément de base de SCA est le composant qui constitue l'unité élémentaire de construction.

Un composant est une instance configurée d'implémentation où une implémentation est un code source fournissant des fonctionnalités.

Ces fonctionnalités sont exposées en tant que services en vue de leur utilisation par d'autre composant. Les services sont décrits au travers d'interfaces qui constituent le contrat de service. Ces contrats sont implémentés par le composant.

Les paramètres des services sont des structures de données pour lesquelles SCA recommande fortement l'utilisation de la spécification SDO (Service Data Objects) dont il est également l'instigateur.

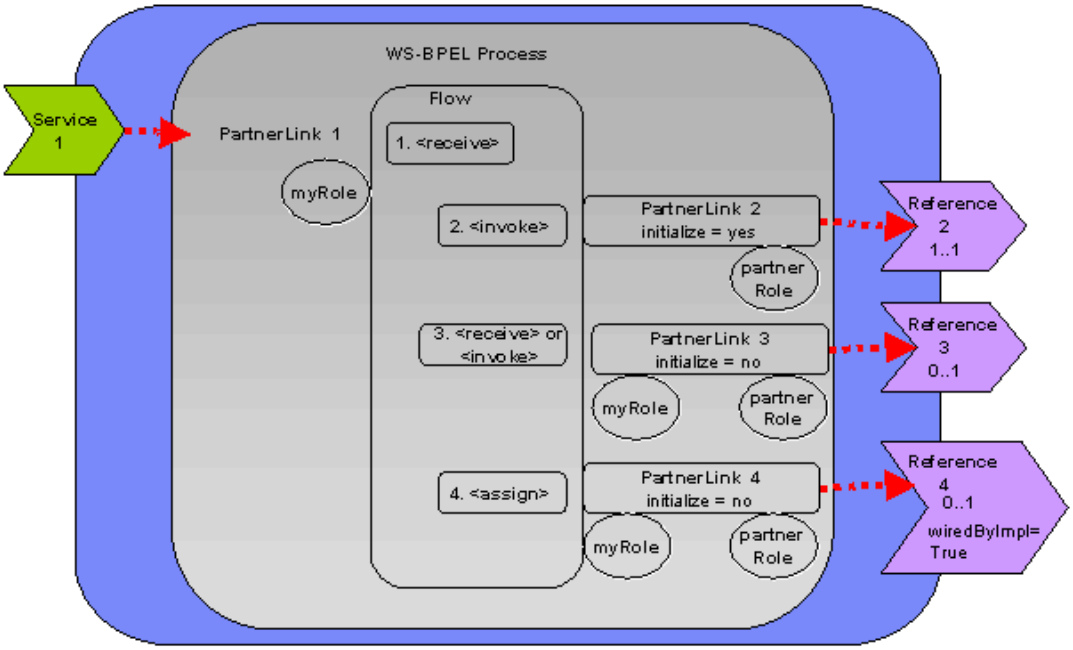


Figure 11 : Orchestration BPEL dans SCA

L'**implémentation** peut s'appuyer sur des services fournis par d'autres composants dont elle dépend. Ces dépendances sont appelées **références**. Elles sont associées à des services qui peuvent être soit exposés par d'autre composants SCA, soit exposés par des systèmes tiers. L'implémentation peut être paramétrable au travers de **propriétés** qui influencent le comportement d'une fonctionnalité. C'est le composant qui configure l'implémentation en fournissant des valeurs à ses propriétés et en liant les références aux services fournis par d'autres composants.

Le système de références associé à celui des interfaces permet de réaliser un couplage lâche (ou faible) entre les composants : Un composant consommateur de services ne connaît des composants fournisseurs de services sur lesquelles il s'appuie que les interfaces (contrat de service) des services qu'il consomme.

Le modèle d'assemblage

Le deuxième élément définit par SCA est le **composé** qui est un assemblage de composants, services, références, propriétés et des liens qui existent entre ces éléments.

Un composé n'est donc rien d'autre qu'un composant de plus haut niveau que ceux qui le compose (Il fournit des services, dépends de références et a des propriétés). Un composé peut donc à son tour être référencé par d'autres composants et utilisé au sein d'autres composés.

L'utilisation première du composé peut être "détournée" pour regrouper un ensemble d'éléments non nécessairement liés mais qui constitue un ensemble fonctionnel cohérent.

Au plus haut niveau, les composés sont déployés dans des **domaines** SCA qui regroupe l'ensemble des services pour un système fonctionnel.

ANNEXE B : ETAT COMPLET D'UNE ORCHESTRATION

```
<processState alarmId="0" coordinator="false" currentState="Finished"
                    endDate="2008.04.16 13:29:55.578 CEST"
                    endDateMillis="1208345395578"
                    exiting="false"
                    hasCoordinations="false"
                    hasImplicitCompensationHandler="false"
                    hasImplicitFaultHandler="false"
                    hasImplicitTerminationHandler="false"
                    invokeId="-1"
                    locationPath="/process"
                    maxLocationId="20"
                    nextVariableId="15"
                    normalCompletion="true"
                    participant="false"
                    pid="1"
                    process="{http://AeGetVersionTest}AeGetVersionTest"
                    processState="3"
                    processStateReason="-1"
                    snapshotRecorded="false"
                    startDate="2008.04.16 13:29:55.046 CEST"
                    startDateMillis="1208345395046"
                    stateDocVersion="3.0"
                    terminating="false">
         <bpelObject currentState="Finished" locationPath="/process/sequence" terminatin
            <bpelObject currentState="Finished" locationPath="/process/sequence/receive"
                        terminating="false"/>
            <bpelObject alarmId="-1" currentState="Finished" eid="0" jid="0"
                        locationPath="/process/sequence/invoke"
                        queued="false"
                        terminating="false"
                        transmissionId="0"/>
            <bpelObject currentState="Finished" locationPath="/process/sequence/assign"
                        terminating="false"/>
            <bpelObject currentState="Finished" locationPath="/process/sequence/reply"
                        terminating="false"/>
         </bpelObject>
         <variable dataIncluded="no" hasData="true" name="getGenericClientRequest" versi
         <variable dataIncluded="no" hasData="true" name="getGenericClientResponse" vers
         <variable dataIncluded="no" hasData="true" name="getVersionRequest" version="5"
         <variable dataIncluded="no" hasData="true" name="getVersionResponse" version="1
         <partnerLink locationPath="/process/partnerLinks/partnerLink[@name='myPLType']"
                      name="myPLType"
                      version="9">
            <partnerRole>
               <wsa:EndpointReference xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/a
                                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instan
                                      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                                      xmlns:ns5="http://localhost:8080/active-bpel/servi
                                      xmlns:soapenv="http://schemas.xmlsoap.org/soap/env
                  <wsa:Address>s:anyURI</wsa:Address>
                  <wsa:ServiceName PortName="Version">ns5:VersionService</wsa:ServiceNam
               </wsa:EndpointReference>
            </partnerRole>
         </partnerLink>
         <partnerLink locationPath="/process/partnerLinks/partnerLink[@name='genericClie
                      name="genericClient_PL"
                      version="10">
            <myRole>
               <wsa:EndpointReference xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/a
                                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instan
                                      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                                      xmlns:ns5="http://localhost:8080/active-bpel/servi
                                      xmlns:soapenv="http://schemas.xmlsoap.org/soap/env
                  <wsa:Address>http://localhost:8080/active-bpel/services/genericClient_
                  <wsa:PortType>ns5:GenericClient</wsa:PortType>
                  <wsa:ServiceName>ns5:genericClient_PL</wsa:ServiceName>
               </wsa:EndpointReference>
            </myRole>
         </partnerLink>
         <queue/>
         <variableLocker/>
      </processState>
```

Figure 12 : Etat complet d'une orchestration fourni par ActiveBPEL

BIBLIOGRAPHIE

1 - « Universal Description Discovery and Integration » , http://uddi.xml.org/

2 - « SOAP » , http://www.w3.org/2002/07/soap-translation/soap12-part0.html

3 - « Web Services Description Language » , http://www.w3.org/TR/wsdl

4 - « Service Web » , http://ws.apache.org/axis/

5 - « Fractal Project » , http://fractal.objectweb.org/

6- « Service Component Architecture Home »,
http://www.osoa.org/display/Main/Service+Component+Architecture+Home

7 - «Le système de composants Fractal» , *Intergiciel et Construction d'Applications Réparties,* T. Coupaye, V.Quéma, L. Seinturier, J.-B. Stefani, licence Creative commons , 2006

8 - « Astro Project », http://www.astroproject.org/

9 – « Dynamo-AOP user Manual », Guide utilisateur de Dynamo

10 – Relationship between SCA and BPEL, Relationship between SCA and BPEL