Implantation du Modèle

1 Introduction

Afin de valider les concepts présentés dans les chapitres 2 et 3, nous avons construit un framework permettant de réaliser des agents d'orchestration. Ce framework est développé en Java, il exploite une architecture de cluster de bus logiciels afin de permettre l'exécution d'orchestration sur un environnement distribué hétérogène. Cette réalisation s'intègre totalement dans notre modèle PSM. En effet, les orchestrations à base d'EIP sont spécifiées au niveau PIM et implantées au niveau PSM. Nous souhaitons désormais aller plus loin dans notre approche et construire une implantation pour établir la faisabilité de notre approche. Notre but est de fournir une implantation basée sur la spécification de notre plateforme d'orchestration toute en préservant les propriétés temporelles du chapitre 3 (P1 à P5).

Nous présentons dans la première section de ce chapitre (cf : section 2) les frameworks et les technologies sur lesquels est basé notre implantation. Nous décrivons l'apport de ces outils et expliquons leurs choix par rapport aux autres outils concurrents. Dans la deuxième section (cf : section 3) nous présentons notre architecture logicielle en mettant en avant les composants de notre système et les modifications que nous avons apportées au framework sous-jacent afin qu'ils respectent notre spécification définie dans le chapitre 2. Dans la troisième section (cf : section 4) nous décrivons la structuration du cluster logiciel ainsi que la connectique utilisée afin de connecter les différents nœuds au sein de ce cluster. Nous détaillons aussi les technologies utilisées afin de permettre le déploiement distant à chaud des différents agents d'orchestration. Dans la dernière section (cf : section 5) nous décrivons le mécanisme de développement et de packaging d'un agent d'orchestration.

2 Introduction aux technologies utilisées

Cette section a pour objectif d'introduire les technologies utilisées dans le cadre de notre implantation. L'ISM est notre modèle d'exécution, c'est le lieu d'utilisation de ces technologies. Nous reprenons les couches applicatives illustrées dans la Figure 2-27 du chapitre 2 en justifiant les choix des technologies sur les couches importantes.

Nous avons défini trois niveaux de PSM dans la section 4 du chapitre 2, ces niveaux de transformation nous permettent d'aboutir à un agent d'orchestration prêt à être utilisé dans un cluster de bus logiciels. Nous nous attardons dans cette section sur la structure et les composants de ce cluster. Nous décrivons ainsi l'implantation du système qui permet d'évaluer nos agents d'orchestrations. Ce système est composé du conteneur d'application, du moteur d'exécution et du référentiel décrit respectivement dans les sections 2.4.1.4, 2.4.1.5 et 2.4.1.3 du chapitre 2.

2.1 Choix du bus logiciel

Les bus logiciels sont communément utilisés afin de mettre en place des architectures SOA (Service Oriented Architecture). Ce type d'architecture est basé sur l'identification des services faiblement couplés avec un middleware pour supporter l'intégration et l'orchestration de ces services. Un bus

logiciel est considéré comme un important facilitateur pour SOA et à juste titre, considéré comme l'épine dorsale de l'architecture SOA. C'est une architecture qui exploite les Web services, la messagerie middleware, le routage intelligent et la transformation de flux [74].

2.1.1 Caractéristiques du bus logiciel

Les ESB sont définis de manières différentes selon leurs fournisseurs. Il existe cependant un dénominateur commun entre les différents ESBs qui est formulé comme un ensemble de caractéristiques qu'un ESB doit implanter :

- fournir la transparence de localisation
- fournir des normes fondées sur une plate-forme de messagerie pour l'intégration des services d'entreprise
- assurer le couplage faible dans la communication des composants en faisant abstraction des protocoles et les formats de données
- prendre en charge un routage intelligent par contenu des messages et les transformations de flux
- être léger avec une architecture dynamique
- fournir des outils simples d'administration
- avoir la capacité d'être déployé dans un environnement distribué et pouvoir être à la base d'un cluster
- supporter la recherche dynamique de services, au niveau des registres, et des versions de services.

2.1.2 Critère de comparaison

Un ESB offre des services sous la forme de fonctionnalités. Ces services sont configurables à leurs tours et des services supplémentaires peuvent être facilement ajoutés. L'extensibilité d'un ESB permet également l'ajout de services externes pour inter-opérer avec l'ensemble existant de services. Ces services permettent l'intégration d'applications d'entreprise en autorisant la communication à travers des messages de médiation entre les composants. La connectivité entre l'application et l'ESB se doit d'être en cohérence avec l'approche basée sur les standards de l'architecture SOA et ne doit surtout pas en être propriétaire.

Les services offerts par les ESBs peuvent être étendus pour assurer la sécurité, le support transactionnel et la robustesse de la plate-forme. En outre, ils peuvent inclure d'autres services spécifiques tels que le service Registre, des processus métier d'orchestration, etc.

Nous ne fournissons pas une matrice comparative de tous les ESBs disponibles qui couvre les différents critères. Car nous considérons que ce n'est pas possible de créer une matrice correcte et constructive : les produits proposent souvent un trop grand nombre de fonctionnalités et concepts différents. De plus, la liste des fonctionnalités évolue d'une release à une autre d'un même ESB. Nous optons plutôt pour une approche qui permet de prédéfinir les besoins, puis d'évaluer les produits qui conviennent le mieux. Nous utilisons les critères suivants :

- **Facilité d'utilisation** : ce critère concerne la complexité de l'installation, le nombre d'outils nécessaires à cette installation.
- **Maintenabilité** : ce critère concerne l'administrabilité du produit, c'est-à-dire si la solution offre une interface pour le monitoring ?
- **Communauté** : ce critère concerne les forums publics actifs ou des listes de diffusion aussi bien que les articles, tutoriels, et vidéos qui permettent de cerner la solution
- **Efficacité** : ce critère concerne la couverture de l'ESB par rapport aux fonctionnalités requises pour la mise en œuvre de notre système.
- **Flexibilité** : ce critère concerne l'ouverture de l'ESB à des modifications profondes afin de personnaliser les fonctionnalités du produit pour répondre à nos besoins
- **Extensibilité** : ce critère concerne la possible d'étendre le produit, c'est-à-dire si les interfaces proposées par le produit sont basées sur des standards.
- **Connecteurs** : ce critère concerne les adaptateurs disponibles pour interfacer l'ESB avec des services développés avec d'autres technologies et d'autres protocoles.

Nous présentons dans la Figure 4-1 les avantages et les inconvénients des ESBs sur deux niveaux. Ce choix de représentation est basé sur le fait que la plupart des ESB payants ont une version gratuite. Dans les colonnes, nous faisons la distinction entre des solutions propriétaires et des solutions open source. Nous distinguons par un code de couleurs (vert == bien, jaune == acceptable, rouge == mauvais) les critères au sein de la même famille d'ESB.

Critère	Open Source	Propriétaire
Facilité d'utilisation	L'installation se résume à la décompression d'un dossier compressé, utilisation instantanée, multiplateformes	Installation complexe, nécessite l'intérvention d'un expert cértifié dans certains cas
Maintenabilité	Essentiellement en ligne de commande, les outils graphiques ne sont pas très riches. l'analyse du code source est nécessaire parfois pour du refactoring	Outils d'administration puissants, l'analyse du code source n'est pas nécessaire, refactoring via GUI
Communauté	Une grande communauté et des forums, les réponses dépendent cependant de la disponibilité des contributeurs.	Le support est payant mais efficace, il existe des forums mais pas de véritable communauté derrière
Efficacité	Couvre les différentes fonctionnalités requises (d'intégration, orchestration, etc.)	Couvre les différentes fonctionnalités requises (d'intégration, orchestration, etc.) plus d'autres
Flexibilité	Code source disponible avec possibilité de changer tout ce dont on a besoin	Toute demande de changement est payante avec une longue attente

Extensibilité	Dans la majorité des cas basés sur des standards	Les extensions sont difficiles est le mécanisme est mal documenté. Payant dans certains cas
Connecteurs	Ouverts aux technologies avec des interfaces standards	Ouverts aux technologies avec des interfaces standards

Figure 4-1 Comparatif des ESBs propriétaires et open source.

2.2 Présentations des bus étudiés

Nous avons fait un point sur les principales différences entre les produits propriétaires et open source, nous présentons dans cette sous-section certains produits avec un bref descriptif des différentes options disponibles.

2.2.1 Oracle Service Bus

Oracle Service Bus [75] est la solution ESB fournie par Oracle. C'est un composant d'OFM (Oracle Fusion Middleware) [76], qui est souvent défini comme une suite d'intégration qui possède d'autres produits, nous citons : *BPEL Process Manager* qui est dédié aux orchestrations, *Enterprise Messaging Service* qui est un brocker de messagerie et *Service Registry* qui est un annuaire de services.

Cette solution est propriétaire basée sur des standards comme Java EE, BPEL (Business Process Execution Language), SOAP et SCA (Service Component Architecture). Elle est stable et puissante. Elle dispose d'éditeurs graphiques pour la majorité de ces produits ainsi d'un support efficace. Cependant, ce produit est très compliqué, de plus, les licences sont très chères avec un coût du support élevé et une tarification non transparente. Cela est essentiellement dû au fait que l'OFM proviennent des multiples acquisitions d'Oracle, ainsi basé sur différentes bases de code, ces différents produits nécessitent souvent l'utilisation de différents outils de développement. La somme des téléchargements se mesure en dizaines de GigaOctets, leur installation peut prendre plusieurs jours et les ressources nécessaires à leurs l'exécution sont très élevées.

Nous n'avons pas retenu Oracle Service Bus malgré sa stabilité est sa puissance car sa complexité nous empêche de la faire évoluer dans le sens de la migration d'agent d'orchestration. De plus, la licence Oracle interdit toute redistribution de la solution modifiée.

2.2.2 WebSphere ESB

C'est l'ESB produit par IBM, ce produit [77] comprend un serveur, un outil de modélisation et un moniteur de processus métier. Au cœur de ce produit, il y *WebSphere Message Broker*. Ce broker de messages supporte les Web services et fournit des mécanismes de communication comme BizTalk, Java Message Service et HIPAA (Health Insurance Portability and Accountability Act) [78]. Cependant, WebSphere ESB est limité aux protocoles (WSDL, SOAP) et BPEL pour offrir la connectivité et les transformations de données.

Ce produit présente pratiquement les mêmes avantages et inconvénients que la solution d'Oracle décrite dans la section précédente. De ce fait, nous n'avons pas retenu cette solution pour les mêmes raisons que la solution d'Oracle.

2.2.3 Mule ESB

Mule ESB s'inscrit parmi les premiers succès des ESBs open source. Ces qualités se rapprochent des autres mentionnés précédemment et des autres ESBs open source. L'installation de celui-ci est très intuitive en plus d'un outillage basé sur Eclipse. Comme c'est le cas pour la majorité des ESBs open source, ce sont des solutions légères et extensibles. Mule ESB se base sur des éditeurs graphiques qui permettent une mise en œuvre efficace des scénarios d'intégration, ainsi que les connecteurs disponibles pour des produits tels que SAP ou Salesforce, ce qui correspond à ses avantages face aux frameworks comme Apache Camel ou Spring Integration.

Toutefois, Mule ESB ne dispose pas des fonctionnalités d'une suite logicielle comparable à celle des précédents ESBs. Pour ce type d'utilisation, Mule ESB doit être combiné avec des produits provenant d'autres fournisseurs. Cependant, les aspects négatifs de Mule ESB sont sa communauté réduite, le modèle restrictif de licence et l'accès limité au code source. Cela fait un handicap notable par rapport aux concurrents. Donc l'impossibilité de faire évoluer cette plateforme pour y introduire la migration.

Mule ESB est disponible en version open source gratuite, il est disponible aussi sous forme d'une version d'entreprise commerciale payante. Cette version offre des fonctionnalités supplémentaires ainsi qu'un support supplémentaire pour le produit.

Contrairement aux deux solutions précédentes (cf : sections 2.2.1 et 2.2.2), Mule ESB, dans sa version open source, est distribué sous licence (Common Public Attribution License). Cependant, cette licence introduit des contrôles sur les versions modifiées que nous ne souhaitons pas imposer aux utilisateurs de notre solution. De plus, les composants qui nous intéressent sont disponibles avec une licence (GNU General Public License).Nous avons opté de les utiliser directement tel que nous décrivons dans la section 2.2.7.

2.2.4 Fuse ESB

Assez semblable à Mule ESB, Fuse ESB est un pur ESB ne disposant pas d'une suite logicielle. Avec environnement de développement intuitif basé sur Eclipse, Il est bâti sur des standards de l'intégration supporté par Apache tels qu'Apache CXF (moteur de web services) et Apache Camel (routage intelligent des messages). Il dispose aussi d'une grande communauté active sur les forums et mailing listes.

Avant son acquisition par Red Hat qui appartient désormais à la division de JBoss, Fuse ESB faisait partie de FuseSource. Son évolution continue à être supportée car il est intégré dans la plateforme JBoss Enterprise SOA [79]. Ce produit présente pratiquement les mêmes avantages et inconvénients que la solution Mule ESB décrite dans la section précédente. De ce fait, nous n'avons pas retenu cette solution pour les mêmes raisons que la solution Mule ESB.

2.2.5 Talend ESB

Talend ESB [80] peut s'utiliser soit de manière indépendante, soit en combinaison avec d'autres composants de la suite de Talend. Une version gratuite est disponible pour tous les composants qui sont disponibles en open source. La version entreprise offre des fonctionnalités supplémentaires dont les sources ne sont pas disponibles et du support. Talend ESB offre un avantage notable par rapport aux produits propriétaires : tous les composants sont basés sur la même base de code, De plus, le même outillage est utilisé partout dans la solution. L'autre avantage notable est l'utilisation de différentes composantes de cette solution (l'ESB, le BPM, l'ETL, le MDM, etc.) ne sont pas séparées dans des outils distincts.

Etant basés sur Eclipse, tous les outils de la suite Talend profitent de l'utilisation intuitive d'Eclipse et d'un designer visuel qui permet de profiter d'une approche appelé « zero-coding ». Il est bien sûr toujours possible d'écrire et d'intégrer des classes Java (POJO) ou bien d'autres langages de scripting.

Basé sur plusieurs standards de l'intégration tels qu'Apache Camel, Apache CXF, Apache Karaf et Apache Zookeeper (négociation distribué), tout comme Fuse ESB, Talend ESB utilise des connecteurs pour des technologies comme JMS, HTTP ou FTP via Apache Camel. Il dispose aussi d'adaptateurs pour Alfresco (serveur de gestion de contenu), Jasper, SAP, Salesforce. Riche de plus de 500 connecteurs inclus par défaut, l'IDE de Talend exige des ressources matérielles plus élevées que ses concurrents.

Talend ESB est un bon candidat pour être la base de notre plateforme d'orchestration. Car disponible en version gratuite (GNU General Public License), nous sommes libres de rajouter la notion de migration à cette. Cependant, sa base de code étant unique et ces composants liés, les modifications de cette solution sont fastidieuses avec des implications sur des composants qui ne nous intéressent pas dans le cadre de notre approche. Donc cette solution ne dispose pas d'un apport notable par rapport à la solution présenté dans la section 2.2.7.

2.2.6 WSO2 ESB

WSO2 [81] propose une large gamme des composants, ceux-ci sont fournis sous forme d'une suite (Business Activity Monitor, Business Rules Server, Business Process Server, Governance Registry, etc.). Avec une installation relativement facile, cette plateforme offre un IDE léger basé sur Eclipse, elle utilise aussi des projets open sources dans ses composants (Synapse, Axis, ODE).

Un autre point où WSO2 est similaire à Talend : tous ces composants reposent sur une seule base de code. Donc son développement est basé sur un processus itératif incrémental. Par contre, son plus grand point faible est son interface graphique qui n'est pas aussi intuitif à utiliser que ses concurrents.

Comme Talend ESB, WSO2 est aussi un bon candidat pour être la base de notre plateforme d'orchestration. Cependant, nous n'avons pas retenu cette solution pour les mêmes raisons que la solution Talend ESB. Le coût des modifications serait trop élevé.

2.2.7 La suite d'intégration « home-made »

Combiner plusieurs frameworks ou plusieurs produits pour construire une suite d'intégration peut s'avérer inutilement coûteux et peut conduire à de nombreux pièges (écriture de tests et des corrections de bugs sur le code d'intégration de ces framewoks).

Dans notre cas, cela s'avère intéressant car nous avons besoin de fonctionnalités bien spécifiques et non pas d'une suite généraliste qui contient des fonctionnalités dont nous ne nous servons pas dans le cadre de notre approche. De plus, nous apportons des modifications sur des fonctionnalités telles que les modifications relatives à la migration (cf : section 3.2.1) qui ne seront pas maintenues par le fournisseur de la solution.

Nous optons pour une suite d'intégration basée sur Apache Karaf [82] qui nous permet une intégration simple d'Apache Camel, Restlet et Apache CXF qui sont les frameworks qui correspondent le plus aux spécifications de notre système. Ces solutions sont disponibles sous licence (GNU General Public License), nous pouvons les modifier et les redistribuer sans avoir des contraintes particulières.

Les codes source de ces frameworks sont maintenus par des grandes communautés très impliquées. De plus, nous avons une expérience dans le projet Apache Camel car nous faisons partie de sa communauté de développement en tant que contributeur. Durant les contributions que nous avons eu l'occasion de faire, nous nous sommes familiarisés avec le développement des connecteurs, notamment celui de AMQP (Advanced Message Queuing Protocol) sur lequel nous avons déjà contribué.

Ces arguments font de cette suite d'intégration basée sur ces frameworks le candidat idéal pour être la base de notre plateforme d'intégration.

2.3 Présentation d'Apache Karaf

Apache Karaf est un environnement d'exécution basé sur OSGi qui fournit un conteneur léger sur lequel différents composants et applications peuvent être déployées. Il couvre les spécifications du terme conteneur décrit dans la section 2.4.1.4 du chapitre 2. Il utilise les frameworks Apache Felix ou bien Eclipse Equinox comme conteneur OSGi.

La raison du nommage « conteneur léger » est qu'Apache Karaf est une solution combinant la facilité de développement d'une application Java EE et d'une application standalone utilisant Spring. Ainsi, le nommage « léger » est expliquée par Erik Gollot comme :

« SPRING est effectivement un conteneur dit " léger ", c'est-à-dire une infrastructure similaire à un serveur d'applications J2EE. Il prend donc en charge la création d'objets et la mise en relation d'objets par l'intermédiaire d'un fichier de configuration qui décrit les objets à fabriquer et les relations de dépendances entre ces objets. Le gros avantage par rapport aux serveurs d'application est qu'avec SPRING, les classes n'ont pas besoin d'implémenter une quelconque interface pour être prises en charge par le framework (au contraire des serveurs d'applications Java EE et des EJBs). C'est en ce sens que SPRING est qualifié de conteneur " léger ". »

2.3.1 Fonctionnalités générales

Apache Karaf fournit des fonctionnalités intégrées telles que :

- **Déploiement à chaud** : il offre deux méthodes de déploiement, soit par simple dépôt d'un fichier dans le répertoire de déploiement, (Apache Karaf détecte le type de fichier et essaye de le déployer), soit en utilisant le mécanisme de Provisioning afin de localiser, télécharger et installer le fichier. Le déploiement à chaud est indispensable pour déployer nos agents d'orchestration suite à une demande de migration sans interruption de service sur le conteneur cible.
- **Console riche** : il fournit une console avec un Shell Unix complet qui permet de gérer complètement le conteneur. La console dispose d'un accès distant qui nous permet d'appliquer la configuration nécessaire à l'installation de nos frameworks (cf : section 3.2.3) à distance suite à l'ajout d'un nouveau nœud à note cluster.
- **Configuration dynamique** : il fournit un ensemble de commande dédié pour la gestion à chaud de la configuration ainsi que des fichiers de configuration. Il est à noter que les fichiers de configuration sont centralisés et que tout changement est répercuté sur l'instance à chaud.
- **Système logs puissant** : il prend en charge plusieurs frameworks de Logging tels que slf4j (Simple Logging Facade for Java) et log4j.
- **Provisioning** : il prend en charge un grand nombre de types URLs (dépôt Maven, HTTP, fichier, etc) pour effectuer une installation d'un composant. Il fournit également la notion de « fonctionnalité » qui est une façon de regrouper un ensemble de composants interdépendants. Ce système de provisioning est la base de la méthode que nous utilisons pour intégrer notre plateforme d'orchestration (cf : section 3.2.3). Il est aussi la base utilisé pour le mécanisme de déploiement initial de nos agents d'orchestration et aussi sur les conteneurs distants suite à une demande de migration.
- **Monitoring** : il offre un grand nombre d'indicateurs de gestion et des opérations via JMX. Cela est utile pour tracer l'exécution des agents d'orchestration, surtout suite à une migration.
- **Sécurité** : il fournit un cadre de sécurité complet pour les accès aux services et la console d'administration.
- **Cluster** : plusieurs instances d'Apache Karaf peuvent être organisées en mode grappe ou cluster et gérées à partir d'une instance principale. Ce principe est indispensable afin de lier les nœuds et de permettre aux agents d'orchestration d'avoir une visibilité sur leur environnement de déploiement.
- **Compatibilité OSGi**: il fonctionne avec Apache Felix comme conteneur OSGi par defaut, mais il suffit de changer une propriété pour passer à Eclipse Equinox.

2.3.2 Intégration avec Apache Camel

Afin d'avoir les mêmes fonctionnalités offertes par un ESB dans le conteneur Apache Karaf, nous proposons de l'associer à un conteneur de composants d'intégration nommé Apache Camel [47]. Nous considérons Camel comme étant un moteur de routage et de médiation fondé sur des règles qui peuvent être utilisées à l'intérieur d'un conteneur OSGi, un broker de messages ou un client intelligent pour Web services. Camel permet de transformer le conteneur Apache Karaf en un ESB car Apache

Camel est considéré comme un ESB léger, embarquable et il peut fournir la plupart des services communs d'un ESB comme le routage intelligent, la transformation, la médiation, la surveillance, l'orchestration etc.

2.4 Présentation d'Archiva

Dans notre approche, nous utilisons la notion de référentiel (cf : équation (2-88)) qui nous permet de partager nos agents d'orchestration entre les différents nœuds de notre cluster. Il existe plusieurs implantations qui sont plus ou moins équivalents, nous avons retenu une implantation distribuée sous licence GPL (GNU General Public License) nommée Archiva.

Apache Archiva [83] est un serveur d'artéfacts extensible, un artifact est un élément spécifique (se matérialise la plupart du temps sous forme de JAR, WAR ou EAR) issu de la construction (ou build) du logiciel. Ce serveur développé en Java nous permet la gestion des référentiels que nous avons définis dans la section 2.4.1.3 du chapitre 2 et les artéfacts issus des builds des agents d'orchestration.

2.4.1 Fonctionnalités générales

Archiva est fait pour fonctionner avec des outils de construction (ou build) tels que ANT, CONTINUUM ou. MAVEN que nous décrivons dans la section suivante. Archiva est utilisé essentiellement pour profiter des trois fonctionnalités qui suivent :

- Faire le stockage des d'agents d'orchestration sous forme d'artefacts (cf : équation (2-87))
- Partager les artefacts stockés (cf : équation (2-90))
- Gérer les dépendances lors des builds.

Archiva est utilisé aussi pour qu'on puisse déployer et télécharger les librairies utilisées par les agents d'orchestration. Il est configuré par défaut pour être un proxy aux référentiels Maven, c'est-à-dire qu'il stocke une copie des librairies afin qu'elles soient accessibles plus rapidement depuis un serveur interne. Archiva permet aussi d'autres fonctionnalités telles que :

- Purges automatiques des versions de développement (Snapshots) dans les référentiels (cf : section 5.2).
- Manipulations manuelles directes (des uploads et des suppressions) des artifacts.

2.4.2 Dépôt Maven

Maven est un outil de gestion et d'automatisation de production de projets logiciels, qui permet notamment de réduire la duplication des dépendances de bibliothèques requises pour construire les agents l'orchestration. Son but est de distribuer des agents d'orchestration aux clients qui les demandent, tout en offrant des fonctionnalités de gestion avancées.

Il existe trois types de dépôts Maven, chaque type est utile dans notre approche à un niveau donné. Ces trois types de dépôts sont complémentaires et chacun d'entre eux couvre une partie du cycle de vie d'un artefact, qu'il soit un artéfact représentant un agent d'orchestration ou bien un artefact

représentant une librairie externe utilisée comme dépendance dans de cadre de développement d'un agent d'orchestration :

- **Dépôt local** : Il est situé sur l'ordinateur du développeur. Toutes les bibliothèques utilisées pour construire les agents d'orchestration y sont téléchargées depuis un dépôt interne ou public (cf : section 5.2).
- **Dépôt privé interne** : C'est le dépôt que nous utilisons pour communiquer les agents d'orchestration entre les différents nœuds du cluster et le développeur. C'est un serveur situé à l'intérieur du réseau dans lequel se situe notre contexte d'exécution (cf : section 2.4.1.1. du chapitre 2).
- **Dépôt public** : C'est un dépôt public qui est par défaut fourni par « maven.org ». Il contient les dépendances utilisées par les agents d'orchestration et les fonctionnalités utilisées par notre ESB.

3 Architecture Logicielle

Dans cette section, nous présentons les composants qui interviennent dans la construction de notre système. Nous commençons par détailler le diagramme que nous avons déjà esquissé dans la Figure 2-28 dans le chapitre 2. Ensuite, nous identifions des composants que nous avons modifiés dans les frameworks dans le cadre de notre implantation. Enfin, nous illustrons les propriétés que nous avons prouvées dans notre chapitre 3.

3.1 Les composants de notre ESB

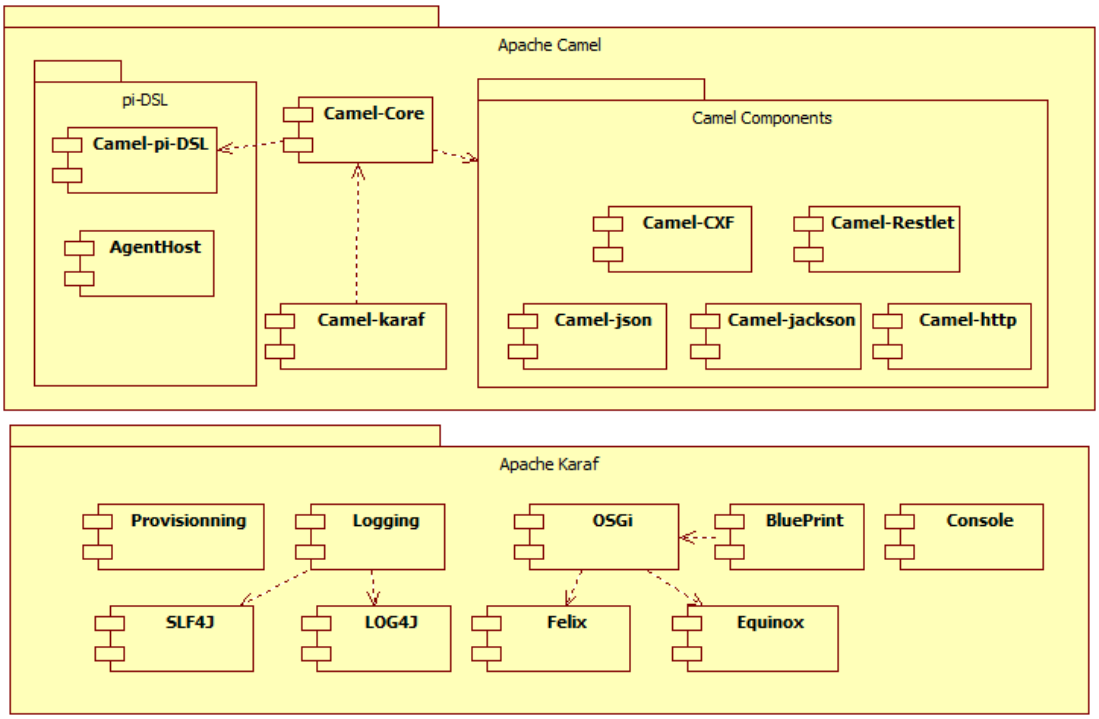

Figure 4-2 Digramme de composants

L'ESB que nous proposons est composé de deux parties, la première partie compose le conteneur Karaf (cf : section 2.3). Il est composé de fonctionnalités intégrées qui permettent le chargement d'un environnement d'exécution des différents composants de notre ESB. Nous rajoutons à ce conteneur une couche d'intégration qui constitue la deuxième partie de notre ESB. Cette couche d'intégration est basée sur une version modifiée d'Apache Camel et des composants spécifiques qui constituent l'implantation du support du langage π-DSL (cf : section 2.2 du chapitre 2).

La Figure 4-2 montre le diagramme de composants organisé en packages. Il est à noter que tous les composants de ces diagrammes sont packagés sous formes de bundles OSGi. Les composants du package « Apache Karaf » sont intégrés à la distribution de Karaf, Les composants du package « Apache Camel » sont déployés comme une fonctionnalité Karaf comme nous le définissons dans la section 3.2.3.

Le package Apache Karaf illustre les composants du conteneur tels que :

- Provisionning : est le composant qui implante le mécanisme de récupération des artefacts (cf : section 2.4.1)
- Logging : est le composant qui déclare le traçage des logs applicatifs et les logs du conteneur.
- slf4j : est un composant qui implante le mécanisme de logging
- log4j : est un composant qui implante le mécanisme de logging
- OSGi : est le composant qui déclare l'interface d'intégration avec les mécanismes de partage de chargeurs de code exécutable tels que définis dans les spécifications OSGi.
- Felix : est le composant qui implante les spécifications OSGi développé par Apache Software Fundation.
- Equinox : est le composant qui implante les spécifications OSGi développé par Eclipse Fundation.
- BluePrint : est le composant qui implante le mécanisme d'injection de dépendances qui permet le partage du code source entre plusieurs composants
- Console : est le composant qui implante l'outil d'administration d'Apache Karaf

Le package Apache Camel illustre les composants du niveau intégration, il dispose de deux sous packages et de deux composants exposés directement dans ce package :

- Camel-Karaf : est le composant qui implante l'intégration entre de framework Camel et le conteneur Karaf.
- Camel-Core : est le composant qui implante les EIPs définis dans la section 2.2 du chapitre 2. Il implante aussi l'activation des agents d'orchestration (cf : section 2.4.1.8 du chapitre 2) et la construction des routes (cf : section 2.4.1.7 du chapitre 2). Ce composant est modifié et construit depuis son code source afin de le rendre compatible avec nos définitions et pour qu'il puisse communiquer avec le package pi-DSL (nous utilisons la notation « pi-DSL » dans les contextes Java et UML au lieu de la notation « π-DSL » utilisé dans notre spécification).
- Package Camel Components : est le package qui contient les composants utiles pour interagir avec les agents d'orchestration et leurs partenaires via des interfaces Web services. Il contient les composants :

- Camel-CXF : est le composant qui implante les mécanismes de communication en Web services SOAP basés sur le framework CXF. Il permet la consommation et l'exposition de web services comme Messages Endpoints tels que définis dans la section 2.2.1 du chapitre 2.
- Camel-http : est le composant qui implante les mécanismes de communication avec le protocole http, il est notamment utilisé par le composant de Provisioning afin de récupérer les artefacts.
- Camel-Restlet : est le composant qui implante les mécanismes de communication en Web services REST basés sur le framework Restlet [84]. Il permet la communication avec le composant AgentHost afin de lui communiquer la demande d'installation d'un agent d'orchestration. Il permet aussi de transférer l'état d'un agent d'orchestration d'un hôte à un autre.
- Camel-json : est le composant qui implante le format d'encodage de données utilisé dans le cadre d'échanges avec des web services REST.
- Camel-jackson : est le composant qui implante le binding entre le format JSON et Java.

- Package pi-DSL : est le package qui contient les composants propres à notre approche de migration définie dans la section 2.3 du chapitre 2. Il est composé de deux composants
  - AgentHost : est le composant qui implante la couche d'interactions avec les instances distantes du conteneur (cf : équation (2-90)) dans le cadre de notre cluster. Il écoute sur une interface REST afin de permettre l'installation d'agents d'orchestration sur l'hôte distant.
  - Camel-pi-DSL : est le composant qui implante notre approche de migration (cf : section 2.3 du chapitre 2) et permet ainsi d'ajouter l'EIP « migrate » au framework Apache Camel.

3.2 Modifications des composants Camel

Camel-core est le module de base de d'Apache et de notre solution. Il contient l'API publique, le DSL Java et plusieurs paquets de mise en œuvre. Les paquets les plus importants sont :

- Builder : Contient Le DSL pour la création des routes qui est une implantation des équations (2-76), des prédicats tels que celui défini dans l'équation (2-96), des expressions et des gestionnaires d'erreur.
- Model : Contient les classes qui forment le DSL Java qui ainsi que la classe principale RouteBuilder qui est l'implantation de l'équation (2-92)
- Language : Contient l'API du langage et des plugins pour les expressions et prédicats
- Component : Contient certains composants comme les beans ou le logging. Il contient aussi la plupart des composants des autres modules de Camel tels que CXF et Restlet qui sont l'implantation des Message Endpoint décrits dans les équations (2-41) et (2-42)
- Impl : Contient l'implantation des classes du modèle et de Camel
- Processor : Contient les processeurs qui implantent les différents EIPs tels que ceux définis dans les équations (2-44) et (2-73)
- Spi : Contient les interfaces pour fournisseur de services spécifiques et l'API qui permet aux développeurs d'étendre Camel

- Camel : C'est le paquet base de Camel. Il contient des définitions comme message qui est défini dans la section 2.2.3 du chapitre 2 et CamelContext qui est une implantation de notre contexte d'orchestration défini dans l'équation (2-90)

3.2.1 Modifications du core de Camel

Afin de permettre le support de l'EIP « migrate » (cf : section 2.3 chapitre 2), des modifications sont nécessaires sur le composant Camel-Core. Nous avons modifié différentes classes dans l' implantation afin de rajouter le code nécessaire au support de cet EIP.

Parmi les classes fournies par Camel-Core, nous nous sommes intéressés à la classe `RouteBuilder` (cf : Figure 4-3) qui est l'implantation de l'équation (2-92). Cette classe comporte d'implantation des méthodes correspondantes aux réceptions telle que la méthode `from(String uri)` qui correspond à la réception sur le canal $from(uri)$ Nous avons modifié la classe de retour `RouteDefinition` afin que l'on puisse substituer la classe `RouteDefinition` (code barré) par la classe `MobileRouteDefinition` qui est notre implantation avec le support de la migration.

```
package org.apache.camel.builder;

import org.apache.camel.CamelContext;
...

/**
 * A <a href="http://camel.apache.org/dsl.html">Java DSL</a> which is
 * used to build {@link org.apache.camel.impl.DefaultRoute} instances in a {@link
CamelContext} for smart routing.
 *
 * @version
 */
public abstract class RouteBuilder extends BuilderSupport implements RoutesBuilder {

...

    /**
     * Creates a new route from the given URI input
     *
     * @param uri  the from uri
     * @return the builder
     */
    public RouteDefinition from(String uri) {
        getRouteCollection().setCamelContext(getContext());
        RouteDefinition answer = getRouteCollection().from(uri);
        configureRoute(answer);
        return answer;
    }

    public MobileRouteDefinition from(String uri) {
        getRouteCollection().setCamelContext(getContext());
        MobileRouteDefinition answer = getRouteCollection().from(uri);
        configureRoute(answer);
        return answer;
    }
...


}
```

Figure 4-3 Illustration de la classe « RouteBuilder »

La classe `MobileRouteDefinition` (cf : Figure 4-4) contient l'enrichissement de la définition d'une route afin qu'elle puisse faire appel au Template de migration défini dans l'équation (2-79) grâce à la méthode `migrate(String uri)`. Cet enrichissement permet d'ajouter des EIPs nouveaux à la liste des

EIP disponible. Notre but est qu'elle soit compatible avec la version finale du vecteur d'EIP tel qu'il est défini dans l'équation (2-77).

```
package org.apache.camel.model;

import org.apache.camel.CamelContext;
...

public class MobileRouteDefinition extends ProcessorDefinition<MobileRouteDefinition> {
    private List<FromDefinition> inputs = new ArrayList<FromDefinition>();
    private List<ProcessorDefinition<?>> outputs = new ArrayList<ProcessorDefinition<?>>();
 ...
    public RouteDefinition(String uri) {
        from(uri);
    }

...
    /**
     * Apply the migration template
     *
     * @param uri the uri to migrate to
     * @return the builder
     */
    public MobileRouteDefinition migrate(String uri) {
        applyMigrationTemplate(uri);
        return this;
    }


    private void applyMigrationTemplate(String uri)
    {
    // uri must be a http uri with host and port like http://localhost:8182
        to("restlet:"+ uri +"/restlet/InstansitateAgent?restletMethod=post")
        .process(new MigrationProcessor())
        .marshal().json()
        .to(uri +"/restlet/proxy")

        from("restlet:"+ uri +"/restlet/proxy?restletMethod=post")
        .unmarshal().json()
        .process(new ReverseMigrationProcessor());

    }

 ...
}
```

Figure 4-4 Illustration de la classe « RouteDefinition »

La méthode `applyMigrationTemplate` constitue l'implantation du Template de migration défini dans l'équation (2-79). Nous avons rajouté des transformations telles que la transformation d'encodage `.marshal().json()` et `.unmarshal().json()` ainsi que des méta données relatives au protocole REST tel que le schéma `restlet:` et `?restletMethod=post` .

Les processeurs de migration `MigrationProcessor` (cf : Figure 4-5) et `ReverseMigrationProcessor` (cf : Figure 4-6) sont des implantations des termes définis dans l'équation (2-80). Les deux processeurs implantent la méthode `process(Exchange exchange)` qui est invoquée suite à la réception d'un échange entrant.

Nous nous sommes intéressés dans le chapitre 2 essentiellement aux communications de notre plateforme d'orchestration, les transformations de données et le format des flux de données n'étaient pas observables dans notre spécification en π-calcul. L'implantation de ces deux processeurs est l'enrichissement apporté à notre spécification (cf : équations (2-80)) où nous nous intéressons aux notions non observables dans le chapitre 2.

```
package org.apache.camel.processor;

...

public class MigrationProcessor implements Processor{

    @Override
    public void process(Exchange exchange) throws Exception {
        AgentState state = new AgentState();
        state.setExachangeProps(exchange.getProperties());
        state.setHeaders(exchange.getIn().getHeaders());
        state.setBody(exchange.getIn().getBody());
        exchange.getOut().setBody(state);
    }
 ...
}
```

Figure 4-5 Illustration de la classe « MigrationProcessor »

Le processeur de migration `MigrationProcessor`, qui est un des composants de notre approche de migration (cf : section 2.3 du chapitre 2), permet de charger le contexte de l'échange entrant sur ce processeur dans un POJO `AgentState`. Il est fait pour contenir les propriétés, les entêtes et le core de l'échange en cours. Ce POJO est intégré comme sortie de l'échange en cours afin qu'il soit transmis à l'hôte distant.

```
package org.apache.camel.processor;

...

public class ReverseMigrationProcessor implements Processor{

    @Override
    public void process(Exchange exchange) throws Exception {
        AgentState state = exchange.getIn().getBody(AgentState.class);

        Map<String, Object> props= state.getExachangeProps();
        for (String key : props.keySet()) {
            exchange.setProperty(key, props.get(key));
        }

        exchange.getOut().setHeaders(state.getHeaders());
        exchange.getOut().setBody(state.getBody());
    }
 ...
}
```

Figure 4-6 Illustration de la classe « ReverseMigrationProcessor »

Le processeur de migration inverse `ReverseMigrationProcessor`, qui est un des composants de notre approche de migration (cf : section 2.3 du chapitre 2), permet de charger le contexte de l'échange depuis le POJO `AgentState`. Il est reçu depuis l'hôte source et restaure les propriétés, les entêtes et le core dans l'échange en cours. Cela permet à l'agent d'orchestration distant d'avoir le même contexte d'échange que celui qui a été créé au niveau de l'hôte source.

3.2.2 Modifications du composant BluePrint

Dans la section 3.2.1, nous avons ajouté le support de l'EIP « migrate » (cf : section 2.3 chapitre 2) dans le DSL Java de Camel en modifiant le composant Camel-Core. Nous avons aussi besoin de modifier le composant BluePrint afin d'intégrer l'EIP « migrate » dans Le DSL XML. L'intégration XML permet d'utiliser la classe `MobileRouteDefinition` que nous avons défini dans Camel-Core afin qu'elle soit liée à la Définition XML d'une route (cf : Figure 2-24) tel que la fournit l'outil Pi2Camel (cf : section 4.1 du chapitre 2). Pour ce faire, nous avons modifié différentes classes dans l'implantation du composant BluePrint afin de rajouter le code nécessaire au support de cet EIP.

Parmi les classes fournies par BluePrint, nous nous sommes intéressés aux trois classes qui portent sur le chargement de la définition d'une route :

- `org.apache.camel.blueprint.handler.CamelNamespaceHandler`
- `org.apache.camel.blueprint.CamelContextFactoryBean`
- `org.apache.camel.blueprint.CamelRouteContextFactoryBean`

Le code Figure 4-7 illustre les modifications apportées à la classe où le code barré représente le code original qui a été modifié :

```
package org.apache.camel.blueprint;

import org.apache.camel.model.IdentifiedType;
...
~~import org.apache.camel.model.RouteDefinition;~~
import org.apache.camel.model.MobileRouteDefinition;

@XmlRootElement(name = "routeContext")
@XmlAccessorType(XmlAccessType.FIELD)
public class CamelRouteContextFactoryBean extends IdentifiedType {

~~    @XmlElement(name = "route", required = true)~~
~~    private List<RouteDefinition> routes = new ArrayList<RouteDefinition>();~~

~~    public List<RouteDefinition> getRoutes() throws Exception {~~
~~        return routes;~~
~~    }~~

    @XmlElement(name = "route", required = true)
    private List<MobileRouteDefinition> routes = new ArrayList<MobileRouteDefinition>();

    public List<MobileRouteDefinition> getRoutes() throws Exception {
        return routes;
    }

}
```

Figure 4-7 Illustration de la classe « CamelRouteContextFactoryBean »

BluePrint utilise une correspondance Java/XML Basée sur le framework JAXB (Java Architecture for XML Binding) [85]. Il permet de charger une définition en XML dans une classe Java. Cette correspondance se base sur des annotations telles que :

- `@XmlRootElement(name = "route")` qui est utilisée afin de faire le lien entre la balise « route » définie dans le fichier XML et la classe `MobileRouteDefinition` qui sera valorisée par le contenu de cette balise.
- `@XmlAccessorType(XmlAccessType.PROPERTY)` qui est utilisée afin préciser que les propriétés de cette classe doivent être valorisées directement, c'est-à-dire que la classe `MobileRouteDefinition` n'a pas besoin d'implanter des accesseurs pour toutes les propriétés.

Le code Figure 4-8 illustre les annotations appliquées à la classe `MobileRouteDefinition` afin de configurer cette correspondance :

```
...
@XmlRootElement(name = "route")
@XmlType(propOrder = {"inputs", "outputs"})
@XmlAccessorType(XmlAccessType.PROPERTY)
public class MobileRouteDefinition extends ProcessorDefinition<MobileRouteDefinition> {
...
}
```

Figure 4-8 Illustration de la classe « MobileRouteDefinition »

Ces modifications nous ont permis d'intégrer le Template de migration implanté en Java dans le Core Camel au schéma BluePrint. Le nouveau schéma généré par JAXB prend en compte ces modifications, nous présentons dans l'extrait Figure 4-9 quelques parties intéressantes de ce schéma XSD.

```
<xs:schema elementFormDefault="qualified" version="1.0"
    targetNamespace="http://camel.apache.org/schema/blueprint">
    ...
    <xs:element name="from" type="tns:fromDefinition" />
    <xs:element name="process" type="tns:processDefinition" />
    <xs:element name="migrate" type="tns:migrateDefinition" />
    ...
    <xs:complexType name="routeDefinition">
        <xs:complexContent>
            <xs:extension base="tns:processorDefinition">
                <xs:sequence>
                    <xs:element ref="tns:from" minOccurs="0" maxOccurs="unbounded" />
                    <xs:choice minOccurs="0" maxOccurs="unbounded">
                        ...
                        <xs:element ref="tns:process" />
                        <xs:element ref="tns:migrate" />
                        ...
                    </xs:choice>
                    ...
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    ...
    <xs:complexType name="migrateDefinition">
        <xs:complexContent>
            <xs:extension base="tns:output">
                <xs:sequence />
                <xs:attribute name="uri" type="xs:string" use="required" />
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    <xs:complexType name="camelRouteContextFactoryBean">
        <xs:complexContent>
            <xs:extension base="tns:identifiedType">
                <xs:sequence>
                    <xs:element ref="tns:route" maxOccurs="unbounded" />
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    ...
</xs:schema>
```

Figure 4-9 Extrait du schéma XSD Camel

Les modifications apportées à ce schéma XSD permettent de le rendre compatible avec terme « Route » (cf : équation (2-78)) qui définit l'intégration de l'EIP « migrate » dans le cadre de la définition d'une route.
Cette intégration est marquée par la balise `<xs:element ref="tns:migrate" />` qui est intégrée au type de définition d'une route grâce à la balise `<xs:complexType name="routeDefinition">`. Le type de l'élément « migrate » est définit dans la balise `<xs:complexType name="migrateDefinition">` ajouté aussi au schéma XSD.

3.2.3 Intégration avec les composants π-DSL

L'un des objectifs de notre démarche est de fournir une façon simple d'installer notre solution. C'est l'un des critères que nous avons avancé dans notre étude comparatifs des ESB (cf : section 2.1.2). Nous proposons pour ce faire de profiter de l'un des avantages d'Apache Karaf qui est le packaging en « fonctionnalités » (ou feature) (cf : section 2.3). Nous avons défini une fonctionnalité « camel-pi-dsl » qui nous permet d'installer notre solution d'une façon simple comme montré dans le code suivant.

```
<features name='camel-2.6.0'>

  <repository>mvn:org.apache.cxf.karaf/apache-cxf/2.6.1/xml/features</repository>
  <repository>mvn:org.jclouds.karaf/jclouds-karaf/1.4.0/xml/features</repository>

  ...

  <feature name='camel-pi-dsl' version='2.6.0' resolver='(obr)' start-level='50'>
    <feature version='2.6.0'>camel</feature>
    <bundle>mvn:edu.u-pec.lacl/agenthost/0.0.1-SNAPSHOT</bundle>
  </feature>

  ...

  <feature name='camel' version='2.6.0' resolver='(obr)' start-level='50'>
    <feature version='2.6.0'>camel-core</feature>
    <feature version='2.6.0'>camel-spring</feature>
    <feature version='2.6.0'>camel-blueprint</feature>
  </feature>
  <feature name='camel-core' version='2.6.0' resolver='(obr)' start-level='50'>
    <feature version='[3,4)'>spring</feature>
    <feature version='1.9.0'>xml-specs-api</feature>
    <bundle>mvn:org.apache.camel/camel-core/2.6.0</bundle>
    <bundle>mvn:org.apache.camel.karaf/camel-karaf-commands/2.6.0</bundle>
  </feature>

  ...

</features>
```

Figure 4-10 Définition de la fonctionnalité « camel-pi-dsl »

La description de la fonctionnalité `camel-pi-dsl`, décrite en gras dans la Figure 4-10, permet d'installer la version de Camel que nous avons modifiée ainsi que l'agent hôte. Cette fonctionnalité est composé de deux éléments : une balise **<bundle>** qui permet d'installer le bundle `agenthost` correspondant au terme « Runtime » (cf : équation (2-90)). Puis la balise <feature> nommé `camel` est elle-même composé de fonctionnalités constituant la version du framework Camel. Nous avons modifiée celui-ci pour implanter une partie des outils que nous avons défini dans le chapitre 2.

Cette description est ajoutée au descripteur de fonctionnalité, cela permet de pouvoir utiliser les commandes du Shell Karaf Figure 4-11.

```
karaf@root> features:addurl mvn:org.apache.camel.karaf/apache-camel/2.6.0/xml/features
karaf@root> features:chooseurl camel 2.6.0
karaf@root> features:install camel-pi-dsl
```

Figure 4-11 commandes du Shell Karaf

Cette facilité d'utilisation est l'un des critères que nous avons avancé dans notre étude comparative des ESB (cf : section 2.1.2).

3.3 Illustration des propriétés

Dans le but d'illustrer les propriétés que nous avons établies au chapitre 3, nous avons intégré deux mécanismes de validation. Ils nous permettent de couvrir les propriétés relatives à la construction (cf : section 3.2 du chapitre 3) et les propriétés d'activation (cf : section 3.3 du chapitre 3). Notre objectif est de montrer que ces propriétés sont conservées au niveau de notre implantation. Autrement dit, nous souhaitons établir que notre implantation est conforme aux exigences décrite au chapitre 2.

Le premier mécanisme que nous utilisons basé sur l'usage des tests unitaires : Nos agents d'orchestration sont définis comme des routes Camel BluePrint (cf : section 3.2.2). Cependant, BluePrint est une technologie spécifique à OSGi, écrire des tests unitaires est assez difficile. Camel offre

une bibliothèque basée sur PojoSR (Pojo Service Registry) qui fournit un registre de service sans l'aide d'un conteneur OSGi. Cela permet de définir des tests unitaires qui traitent le comportement des agents d'orchestration, notamment les propriétés P1, P2 et P3 (cf : section 3 du chapitre 3). Ce mécanisme offre un moyen d'isoler la construction des routes afin de reporter des assertions sur le comportement résultant de l'exécution de notre plateforme d'orchestration.

Le deuxième mécanisme que nous utilisons est la programmation par contrats : un contrat dans ce contexte n'est pas une simple définition d'interface en Java. Nous allons plus loin que la simple définition d'un contrat statique pour définir une partie du contrat dynamique. En effet, nous précisons des préconditions et post-conditions afin de reporter les assertions liées à l'utilisation de notre plateforme d'orchestration par rapport à propriétés exprimées. Les assertions sont cependant sans état, elles ne nous permettent pas d'associer directement les différentes propositions vues au chapitres 3. Pour pallier à cela, nous utilisons les processeurs de migration (cf : section 3.2.1) que nous enrichissons afin de partager l'état de la vérification de chaque propriété. Nous utilisons ce mécanisme essentiellement pour illustrer les propriétés P4 et P5 vues dans le chapitre 3. Ce mécanisme offre un moyen de valider si les entrées et sorties d'un agent d'orchestration sont valides (cf : Propriété de communication, section 3.3.2 du chapitre 3) et si l'hôte sur lequel se déroule une étape d'orchestration est conforme (cf : Propriété de migration, section 3.3.1 du chapitre 3).

3.3.1 Illustration des propriétés par test unitaire

Afin d'illustrer les propriétés de construction de notre plateforme d'orchestration, nous définissons dans la Figure 4-12 un contexte Camel BluePrint simple qui est utilisé par nos tests unitaires.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
        <propertyPlaceholder cache="true" ignoreMissingLocation="true"
              location="classpath:org/apache/camel/component/properties/cheese.properties"/>
        <route>
            <from uri="timer:test"/>
            <migrate uri="host2"/>
            <to uri="log:test"/>
        </route>
    </camelContext>
</blueprint>
```

Figure 4-12 Contexte Camel BluePrint de tests

Ce contexte définit un agent d'orchestration simple qui reçoit en entrée un événement d'un « timer » sur l'hôte source et migre vers l'hôte cible « `host2` » afin d'écrire la valeur de ce « timer » dans les logs. Nous enregistrons ce contexte dans un fichier nommé « `agentOrchestration.xml` » accessible dans le classpath de notre projet de test.

Nous chargeons cette définition d'agent d'orchestration dans un contexte de test isolé grâce à l'API de test Camel BluePrint tel que montré (en gras) dans le code Figure 4-13.

```
package fr.upec.lacl.charif.test.blueprint;

import javax.xml.bind.JAXBContext;
...
import static org.junit.Assert.assertEquals;
...

public class BlueprintPiDslTest {
```

```
    private JAXBContext context = null;
    private Element elem = null;

    @Before
    public void init()  throws Exception {
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        dbf.setNamespaceAware(true);
        DocumentBuilder db = dbf.newDocumentBuilder();
        Document doc = db.parse(getClass().getClassLoader()
                                    .getResourceAsStream("agentOrchestration.xml"));

        NodeList nl = doc.getDocumentElement().getChildNodes();
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            if (node instanceof Element) {
                elem = (Element) node;
                break;
            }
        }
        CamelNamespaceHandler.renameNamespaceRecursive(elem);

        context = JAXBContext.newInstance("org.apache.camel.blueprint:"
                                        + "org.apache.camel:org.apache.camel.model:"
                                        + "org.apache.camel.model.config:"
                                        + "org.apache.camel.model.dataformat:"
                                        + "org.apache.camel.model.language:"
                                        + "org.apache.camel.model.loadbalancer");

    }

 ...
}
```

Figure 4-13 Initialisation de la classe de tests

Cet extrait de la classe de tests « `BlueprintPiDslTest` » montre l'initialisation d'un contexte Camel BluePrint isolé basé sur le fichier de définition de l'agent d'orchestration « `agentOrchestration.xml` » dans l'attribut d'instance de classe « `elem` ». Cet attribut est utilisé dans les méthodes de tests pour interagir avec ce contexte. La méthode de test « init() » est annotée par « `@Before` » afin que le contexte soit rechargé avant chaque méthode de tests.

Nous avons défini une méthode de test par propriété de construction (cf : section 3.2 du chapitre 3) :

P1. Afin d'illustrer cette propriété, nous avons implanté une méthode de test nommée « `testP1()` » définie dans la Figure 4-14.

```
public class BlueprintPiDslTest {
...
    @Test
    public void testP1() throws Exception {

        Unmarshaller unmarshaller = context.createUnmarshaller();
        Object object = unmarshaller.unmarshal(elem);
        assertNotNull(object);
        assertTrue(object instanceof CamelContextFactoryBean);
        CamelContextFactoryBean ccfb = (CamelContextFactoryBean) object;
        assertNotNull(ccfb.getRoutes());
        assertEquals(1, ccfb.getRoutes().size());
        assertNotNull(ccfb.getRoutes().get(0));
        assertNotNull(ccfb.getRoutes().get(0).getInputs());
        assertEquals(2, ccfb.getRoutes().get(0).getInputs().size());
        assertNotNull(ccfb.getRoutes().get(0).getOutputs());
        assertEquals(7, ccfb.getRoutes().get(0).getOutputs().size());
    }
...
}
```

Figure 4-14 Test de propriété P1

Le but de ce test est de valider que les étapes du Template de migration (cf : section 2.3 du chapitre 2) (composé de sept étapes dont une entrée), sont bien ajoutées aux deux étapes définies initialement dans la route, composé d'une entrée et une sortie. Afin de valider cela nous avons défini deux assertions (en gras dans le code) qui valident que le nombre d'entrées et de sorties dans la définition de l'orchestration sont respectivement égales à deux (1+1) et sept (1+6) qui est le nombre d'étapes après l'application du template de migration. Les autres assertions ont pour objectifs de vérifier des contraintes techniques relatives au chargement de la route.

P2. Afin d'illustrer cette propriété, nous avons implanté une méthode de test nommée « `testP2()` » définie dans la Figure 4-15

```
public class BlueprintPiDslTest {
...
    @Test
    public void testP2() throws Exception {

        Unmarshaller unmarshaller = context.createUnmarshaller();
        Object object = unmarshaller.unmarshal(elem);
        assertNotNull(object);
        assertTrue(object instanceof CamelContextFactoryBean);
        CamelContextFactoryBean ccfb = (CamelContextFactoryBean) object;
        assertNotNull(ccfb.getRoutes());
        assertEquals(1, ccfb.getRoutes().size());
        assertNotNull(ccfb.getRoutes().get(0));
        assertNotNull(ccfb.getRoutes().get(0).getInputs());
        assertNotNull(ccfb.getRoutes().get(0).getOutputs());
        assertEquals(9, ccfb.getRoutes().get(0).getInputs().size()
                        + ccfb.getRoutes().get(0).getOutputs().size());
    }
...
}
```

Figure 4-15 Test de propriété P2

Le but de ce test est de valider que dans le cas où la définition de l'agent d'orchestration contient l'EIP « Migrate », alors les étapes associées au Template de migration sont appliquées par le RouteBuilder (cf : section 3.2.1). Pour cela, nous avons ajouté des assertions sur le nombre total d'étapes qui doit être égale à l'addition de nombre d'étape définies (deux) et les étapes ajoutées par le RouteBuilder (sept). Notons qu'à la différence du test précédent, ce test fait abstraction du nombre d'entrée ou de sorties. Nous n'observons que le nombre total d'étapes.

P3. Afin d'illustrer cette propriété, nous avons implanté une méthode de test nommé « `testP3()` » définie dans la Figure 4-16 Test de propriété P3:

```
public class BlueprintPiDslTest {
...
    @Test
    public void testP3() throws Exception {

        Unmarshaller unmarshaller = context.createUnmarshaller();
        Object ctx1 = unmarshaller.unmarshal(elem);
        assertNotNull(ctx1);
        assertTrue(ctx1 instanceof CamelContextFactoryBean);
        Object ctx2 = unmarshaller.unmarshal(elem);
        assertNotNull(ctx2);
        assertTrue(ctx2 instanceof CamelContextFactoryBean);
        assertTrue(!ctx1.equals(ctx2));

    }
...
}
```

Figure 4-16 Test de propriété P3

Le but de ce test est de valider que le RouteBuilder se reinitialise après chaque chargement d'un agent d'orchestration. Pour cela, nous chargeons deux fois la définition de notre agent d'orchestration défini dans le fichier « `agentOrchestration.xml` » dans des variables « `ctx1` » et « `ctx2` » représentant la fabrique « `CamelContextFactoryBean` » qui crée le RouteBuilder. Suite à ce chargement, nous faisons une assertion sur le fait que les deux fabriques de RouteBuilder sont différentes (en gras dans la Figure 4-16). Cela nous permet de vérifier que les RouteBuilder sont reinstanciés à chaque construction de la définition de l'agent d'orchestration.

3.3.2 Illustration des propriétés par contacts

Afin d'illustrer les propriétés d'activation de notre plateforme d'orchestration, nous profitons du contexte mobile introduit par notre approche de migration afin de partager les informations nécessaires aux assertions.

L'implantation du Template de migration nous permet d'avoir des processeurs, un premier processeur « `MigrationProcessor` » qui s'exécute sur l'hôte source afin de récupérer le contexte d'exécution, un deuxième processeur « `ReverseMigrationProcessor` » qui s'exécute sur l'hôte cible afin de restaurer le contexte d'exécution.

Nous modifions le premier afin d'intégrer une information relative à l'hôte dans le contexte tel qu'illustré dans le code Figure 4-17:

```
package org.apache.camel.processor;

...

public class MigrationProcessor implements Processor{

    @Override
    public void process(Exchange exchange) throws Exception {
        exchange.setProperty("HOST_SOURCE", InetAddress.getLocalHost().getHostAddress());
...
        exchange.getOut().setBody(state);
    }
 ...
}
```

Figure 4-17 Illustration de la classe « MigrationProcessor »

Ce code nous permet de garder une trace de l'hôte sur lequel s'est exécuté l'agent d'orchestration avant sa migration, cette information, récupérée grâce à l'instruction Java « **`InetAddress.getLocalHost().getHostAddress()`** », est enregistrée dans une propriété du contexte nommé `"HOST_SOURCE"` comme le montre la ligne en gras.

P4. Afin d'illustrer cette propriété, nous avons ajouté une assertion (en gras dans la Figure 4-18) sur le « `ReverseMigrationProcessor` » pour valider que l'hôte sur lequel s'exécute ce dernier est diffèrent de l'hôte qui est ajouté au contexte par « `MigrationProcessor` ». Pour ce faire, nous utilisons l'information `"HOST_SOURCE"` qui est accessible dans le contexte de l'agent d'orchestration après la migration. Cette information est comparée avec le retour de l'instruction Java « **`InetAddress.getLocalHost().getHostAddress()`** » exécutée cette fois sur l'hôte cible.

```
package org.apache.camel.processor;

...

public class ReverseMigrationProcessor implements Processor{

    @Override
    public void process(Exchange exchange) throws Exception {
...
        Map<String, Object> props= state.getExachangeProps();
        for (String key : props.keySet()) {
            exchange.setProperty(key, props.get(key));
        }

        assert exchange.getProperty("HOST_SOURCE", String.class) !=
                            InetAddress.getLocalHost().getHostAddress() :
                                "The source host is the same as the target host ";
...
    }
 ...
}
```

Figure 4-18 Illustration de la classe « ReverseMigrationProcessor »

Cette assertion permet de lever une alerte si l'hôte source et l'hôte cible ne sont pas différents l'un de l'autre.

La propriété de communication concerne le client et le partenaire de l'orchestration, pour l'illustrer Nous avons appliqué la notion « contrat » dans l'outil SoapUI (cf : section 2.1.1 du chapitre 5). Car cet outil supporte la validation par assertions implantée dans des scripts Groovy.

P5. La définition de cette propriété a été implantée sous forme d'un test d'intégration composé de deux parties : un Mock que nous avons défini afin simuler les actions effectuées, le service partenaire. Ce Mock retourne un résultat (ou playload), illustré dans la Figure 4-19, qui nous permet de l'identifier.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:flig="http://flight.srv1.comparator.charif.lacl.upec.fr/">
   <soapenv:Header/>
   <soapenv:Body>
      <flig:flightInfoResponse>
         <!--Optional:-->
         <return>
            <!--Optional:-->
            <code>SoapService</code>
            <price>100</price>
         </return>
      </flig:flightInfoResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 4-19 Payload retourné par le Mock

Ce Mock retourne un code statique contenu dans la balise « `<code>SoapService</code>` ». Ce code est repris par notre agent d'orchestration et propagé jusqu'au client. Cette première partie de l'implantation de la propriété nous permet d'intégrer le code du partenaire appelé dans le retour de l'agent d'orchestration. De ce fait, nous sommes capables d'identifier le partenaire appelé par l'agent d'orchestration grâce à ce code.

Dans la deuxième partie de l'implantation de cette propriété, nous définissons un client en SoapUI qui permet d'invoquer l'agent d'orchestration et de valider le fait que suite à chaque invocation, le Mock partenaire (identifié par son code) a été exécuté. Pour faire cette validation, nous définissons un script Groovy (cf : Figure 4-20) coté client qui nous permet de

valider que le code retourné par l'agent d'orchestration correspond au Mock que nous avons défini dans la première partie de cette propriété. Nous rajoutons ce script au client SoapUI afin d'activer son exécution suite à chaque appel de l'agent d'orchestration.

```
// check for the identifier element of the partner service in response
def holder = new XmlHolder( messageExchange.responseContentAsXml )
assert holder["//flig: flightOrchResponse/return/code"] == "SoapService"
```

Figure 4-20 Script de validation Groovy

Ce script Groovy se base sur le retour XML de l'agent d'orchestration afin de valider si le Mock partenaire a bien été exécuté. Cela nous permet de valider que suite à chaque invocation du client, le service partenaire est exécuté sur l'hôte distant.

4 Cluster de bus logiciels

L'architecture que nous avons défini dans la section précédente doit relever de nombreux défis afin qu'elle puisse être évolutive, capable de prendre un compte un nombre croissant d'utilisateurs et d'assurer une haute disponibilité. Dans cette section, nous expliquons le sens des termes évolutivité et disponibilité dans le dans le cadre de l'architecture d'ESB que nous proposons.

L'investissement de départ sur un système qui utilise notre approche porte sur des équipements matériels et logiciels capables de gérer un nombre de client faible. Avec la possibilité d'ajouter ultérieurement des matériels et des logiciels pour gérer un nombre beaucoup plus important de clients. Globalement, l'objectif est d'étendre notre système en fonction de l'évolution de sa clientèle, sans jamais interrompre la disponibilité des services.

Pour ce faire, nous proposons l'utilisation de notre ESB en cluster, ou grappe. Un cluster est un groupe logique de serveurs qui exécutent simultanément des agents d'orchestration tout en donnant l'impression au monde extérieur de ne constituer qu'un seul serveur. De plus, ce type de configuration facilite la migration des agents d'orchestration et étend dynamiquement leurs champs d'intervention en fonction de la charge (c'est-à-dire du volume de clientèle).

Dans les approches classiques, les équilibreurs de charge sont utilisés pour distribuer les requêtes des clients entre les différents serveurs du cluster. Ils peuvent être basés sur du matériel, sur du logiciel ou les deux. Dans notre cas, la logique de l'équilibrage de charge est implantée dans les agents d'orchestration grâce à des EIP tels que le routage dynamique (cf : section 2.2.8 du chapitre 2) qui peut conditionner la migration d'un agent d'orchestration sur le serveur qui dispose d'assez de ressources pour exécuter l'agent par exemple.

Comme nous l'avons évoqué, les deux notions de base associées à un cluster sont :

- **Montée en charge** : est la capacité d'un système à accepter un nombre croissant d'utilisateurs. Plusieurs facteurs entrent dans cette mesure, notamment le nombre d'utilisateurs simultanés et le temps de réponse. Dans notre cas, notre système doit être capable d'atteindre ses objectifs de performances en dépit d'un nombre croissant d'utilisateurs. C'est-à-dire que si un agent d'orchestration répond à une requête en 20 millisecondes en moyenne, cette moyenne ne doit pas changer s'il doit répondre à 10000 requêtes simultanées. Cela dit, dans la réalité, ce délai moyen peut augmenter jusqu'à l'engorgement complet.

- **Disponibilité** : appelée aussi haute disponibilité, elle est communément définie comme la redondance des nœuds dans le cluster. Son objectif est de permettre la reprise de traitement, de manière transparente des requêtes sur un autre nœud du cluster dans le cas où le nœud initial est défaillant.

Karaf offre une capacité de basculement grâce à un système de verrouillage au niveau du conteneur qui fournir des performances de basculement plus rapide. Cette capacité est utilisée afin de basculer entre les réplicas d'un conteneur tel que défini dans le cluster de notre système (cf : équation (2-89) du chapitre 2)

4.1 Mise en cluster d'Apache Karaf

L'approche classique de mise en cluster d'Apache Karaf utilise le projet Cellar [86] qui est un sous-projet du Karaf Server. Il fournit une configuration en cluster pour une mise à disposition de plusieurs serveurs Karaf. La base de Cellar est une configuration de cluster de mémoire basé sur Hazelcast [87] (Data Grid Java en mémoire) qui est reproduite à tous les nœuds connus dans le cluster. Le comportement par défaut pour Hazelcast est de rechercher d'autres instances dans le même sous-réseau par multidiffusion. Ce faisant, il est suffisamment rapide pour détecter d'autres instances et commencer à synchroniser la configuration et l'état des serveurs.

Notre approche de mise en cluster ne se base pas sur une approche classique de réplication, mais plutôt sur les principes de migration apportés par les agents d'orchestration. La réplication des agents d'orchestration est effectuée au moment de la migration. Nous ne répliquons pas systématiquement les agents d'orchestrations à leurs installations car leurs réplications sont effectuées dans le cadre de l'installation de la route secondaire (cf : section 2.3 du chapitre 2). L'installation de la deuxième instance sert bien sûr à exécuter la partie distante de l'agent d'orchestration. Cependant, la première partie de l'agent d'orchestration, qui n'est pas exécutée dans le cadre de la migration, nous sert de point d'entrée pour le réplica.

L'instance de l'agent d'orchestration installée dans le réplica n'est pas supprimée après l'exécution de la partie distante de l'agent, elle est utilisée afin de permettre à d'autres clients d'envoyer des requêtes sur ce réplica, dans ce cas, l'agent ne va pas migrer car il s'exécute déjà sur l'hôte distant.

Nous n'avons pas approfondi le mécanisme d'invocation directe des instances distantes dans notre implantation. Cependant, une solution serait de créer un loadbalancer utilisant un registre UDDI. Il permettrait d'adresser l'instance distante directement si besoin. Ce travail reste encore à faire, nous le considérons hors du cadre de ce travail.

4.2 Déploiement sur le cluster

Nous fournissons un moyen facile pour un utilisateur d'installer un agent d'orchestration et ses dépendances. Les « fonctionnalités » Karaf que nous avons décrits dans la section 2.3 nous permettent de regrouper les composants de notre ESB pour une installation facile avec une seule commande (cf : section 2.3.2). En combinant un descripteur de fonctionnalité avec l'URL PAX et le protocole Maven (cf : section 2.4.2), nous pouvons fournir à nos utilisateurs un mécanisme de déploiement simple pour effectuer l'approvisionnement des agents d'orchestration. Notons que les URLs PAX nous permettent

de former des URLs d'un agent d'orchestration en nous basant sur des éléments tels que les identifiant du groupe, d'artéfact, de version et de type. Nous utilisons ensuite le protocole Maven pour d'accéder à des artefacts correspondants à ces agents d'orchestration depuis le référentiel (cf : section 2.4.1.3 du chapitre 2) en nous basons sur ces URLs.

Comme illustré dans la Figure 4-21, la récupération des artéfacts se fait via un processus de résolution où le conteneur Karaf fera usage du dépôt local si possible « Résolution (1) ». Si ce dernier ne contient pas l'artefact associé à l'agent d'orchestration, il se connecte au référentiel interne « Résolution (2) » (cf : section 2.4.2) où le conteneur retrouvera l'artefact.

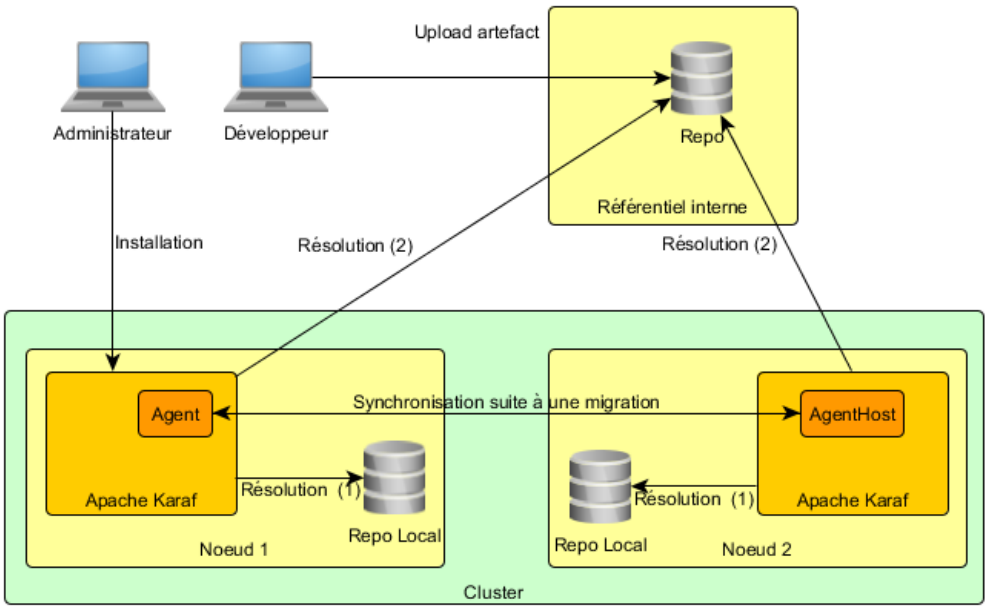

Figure 4-21 Priorité de résolution durant déploiement

La Figure 4-21 illustre aussi le mécanisme de déploiement d'un agent d'orchestration dans le cadre d'un cluster. Suite à une demande d'installation qui est effectué par l'administrateur, l'agent d'orchestration est récupéré et installé sur le premier nœud « Nœud 1 ». Une fois invoqué, l'agent envoie une demande de synchronisation telle que définie dans le Template de migration (cf : section 2.3 du chapitre 2). Cette demande permet à l'AgentHost de se synchroniser avec le référentiel interne Maven afin de récupérer l'artefact.

4.3 Cycle de vie d'un agent dans un cluster

Comme dans tout conteneur, les éléments contenus dans Apache Karaf possèdent un cycle de vie bien particulier puisque Karaf, comme tout conteneur OSGi, doit gérer les composants qui y sont déployés. Nous pouvons distinguer deux familles d'états supportés :

- **Non opérationnel** : cette famille d'états gère la présence d'un agent dans le conteneur sans qu'il soit utilisable. Elle marque la fin de l'étape de construction d'un agent d'orchestration ;
- **Opérationnel** : cette famille d'états correspond aux moments où l'agent est utilisable ou en phase de l'être Elle marque l'activation d'un agent d'orchestration.

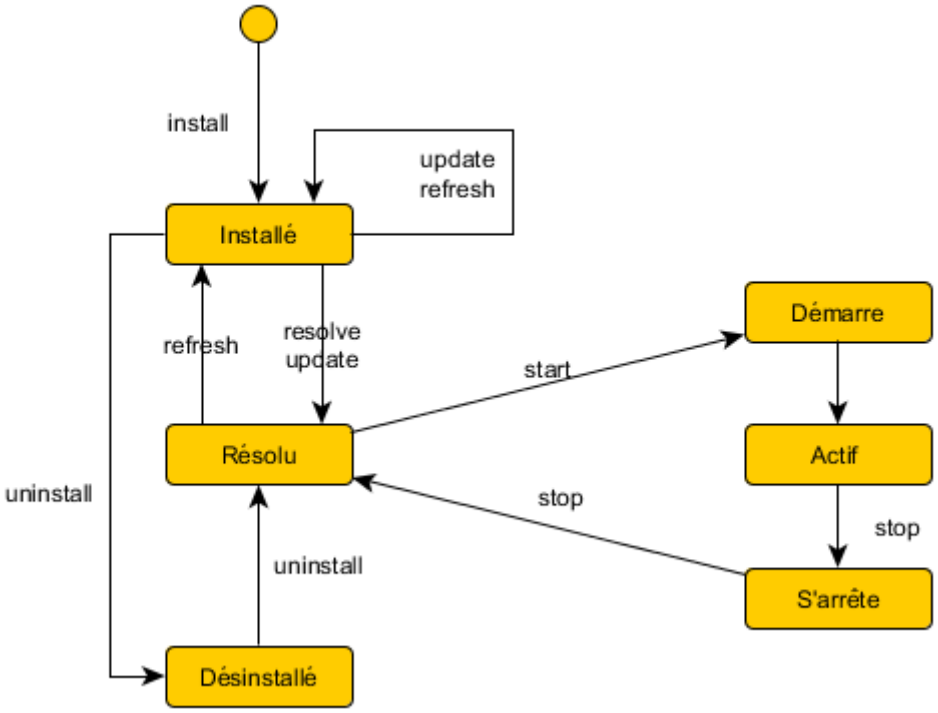

Figure 4-22 Cycle de vie d'un agent

Apache Karaf offre une API standardisée pour gérer le cycle de vie des composants. La Figure 4-22 illustre les différents états de gestion des bundles par le conteneur ainsi que leurs transitions possibles. Les différents états supportés par le conteneur sont :

- Installé : après l'installation de l'agent, mais avant résolution des dépendances et la construction de l'agent.
- Résolu : après l'installation de l'agent, la résolution des dépendances et construction.
- Démarre : l'agent est en cours de démarrage. Cet état correspond à un état transitoire où l'agent d'orchestration est en train d'être construit.
- Actif : l'agent d'orchestration a fini avec suces son activation. Cet état représente que l'agent est actif.
- S'arrête : l'agent est en cours d'arrêt. Cet état correspond à un état transitoire entre les événements Actif et Résolu.
- Désinstallé : l'agent a été désinstallé.

Parmi les composants intégrés à Karaf (cf : section 2.3.1), la console fournit des commandes afin de gérer le cycle de vie des agents et de visualiser leurs états. Nous utilisons une partie de ces commandes dans le cadre de l'installation des modifications apportées au framework Camel (cf : Figure 4-11). Les commandes les plus intéressantes sont :

- install : Installation des agents.
- ps : Affichage de la liste des agents installés.
- refresh : Rafraichissement du contenu d'un agent.
- resolve : Résolution d'un agent et de ces dépendances.
- start : Démarrage d'un agent.

- stop : Arrêt d'un agent.
- uninstall : Désinstallation d'un agent.
- update : Mise à jour d'un agent.

Avec cette console, le conteneur peut être complètement administré à chaud en ligne de commandes. Le scénario suivant peut être mis en œuvre afin d'installer un agent d'orchestration, de le démarrer, puis l'arrêter avec les commandes respectives install, start et stop.

5 Construction d'un agent d'orchestration

Dans le cadre de notre approche, nous avons présenté dans la section 4.1 du chapitre 2 l'outil Pi2Camel et nous avons présenté les trois niveaux de transformations du PSM afin d'aboutir à un code source assez riche pour être considéré comme abouti.

Nous présentons dans cette section le packaging et le build d'un agent l'orchestration à partir de son code source. Nous décrivons les spécificités d'un build dédié à un conteneur OSGi. Nous décrivons aussi le mécanisme de déploiement des composants construits dans le référentiel interne.

5.1 Caractéristiques d'un artefact OSGi

Avec la technologie OSGi, le concept de composant est mis en œuvre par l'intermédiaire d'artefacts appelés bundles. Ces derniers permettent de mettre en œuvre les différents concepts des composants qui sont déployés dans les conteneurs OSGi. Un composant (bundle) OSGi est stocké dans un fichier JAR de Java. Il contient des informations de déploiement spécifiées dans fichier « MANIFEST.MF » localisé dans le répertoire META-INF accessible à la racine du JAR. Ce fichier n'est pas spécifique à OSGi, c'est un fichier standard en Java, le conteneur OSGi le reprend en y ajoutant différents en-têtes afin de configurer le composant.

Le conteneur Karaf possède des caractéristiques pour à la gestion des composants telles que la gestion des dépendances entre composants. Il offre aux composants la possibilité de rendre visible les packages dans le conteneur ou bien de gérer des versions de leurs dépendances. Nous construisons nos orchestrations comme des composants, elles sont donc basées sur ces caractéristiques. L'une des applications les plus courantes est la gestion des dépendances entre l'agent d'orchestration et le framework CXF qui permet de charger les consommateur et les producteurs (cf : section 2.2.1 du chapitre 2).

5.2 Packaging et build d'un agent orchestration

Les composants Camel sont disponibles sous forme de bundles dans le référentiel distant de Maven, ils sont donc accessibles par toutes les instances de Karaf.

Concernant les agents d'orchestration, nous utilisons Maven pour les packager. Nous utilisons un plugin dédié afin d'injecter un descripteur OSGi dans le « jar » résultant et aussi obtenir un livrable sous forme d'un bundle OSGi. La Figure 4-23 présente le plugin qui sert à packager le code obtenu à partir de l'équation $AgentOrch$ (2-82) après son enrichissement.

```
<!-- to generate the MANIFEST-FILE of the bundle -->
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <version>2.3.7</version>
  <extensions>true</extensions>
  <configuration>
    <instructions>
      <Bundle-SymbolicName>OrchAgent</Bundle-SymbolicName>
      <Private-Package>fr.upec.lacl.demo.proc; *</Private-Package>
      <Import-Package>fr.upec.lacl.service; *</Import-Package>
    </instructions>
  </configuration>
</plugin>
```

Figure 4-23 Plug-in pour packaging OSGi

C'est le plug-in illustré dans la Figure 4-23 qui marque l'appellation « Bundle » d'un agent d'orchestration. Après cette étape, la définition d'un agent d'orchestration devient compatible avec le format « bundle ». Le build avec ce plug-in permet d'enrichir le manifeste en rajoutant des informations supplémentaires telles que illustrés dans le contenu Figure 4-24. Nous remarquons dans cette figure l'importation des classes implantées par le partenaire appelé par l'agent d'orchestration grâce à la propriété Import-Package.

```
Manifest-Version: 1.0
Bnd-LastModified: 1407710830268
Build-Jdk: 1.7.0_45
Built-By: Charif
Bundle-ManifestVersion: 2
Bundle-Name: AgentOrch
Bundle-SymbolicName: AgentOrch
Bundle-Version: 0.0.1.SNAPSHOT
Created-By: Apache Maven Bundle Plugin
Export-Package:
fr.upec.lacl.charif.comparator.orchestrator.flight;uses:="org.apache.camel,fr.upec.lacl.charif.comparator.srv1.flight,org.apache.camel.builder,org.apache.camel.model,fr.upec.lacl.charif.migration.processor,fr.upec.lacl.charif.migration.processor.metric";version="0.0.1.SNAPSHOT",fr.upec.lacl.charif.migration.processor;uses:="org.apache.camel,fr.upec.lacl.charif.comparator.srv1.flight,fr.upec.lacl.charif.comparator.orchestrator.flight,fr.upec.lacl.charif.migration.processor.metric";version="0.0.1.SNAPSHOT",fr.upec.lacl.charif.migration.processor.metric;uses:="org.apache.camel,org.apache.commons.logging";version="0.0.1.SNAPSHOT"
Import-Package:
fr.upec.lacl.charif.comparator.srv1.flight;version="[0.0,1)",org.apache.camel;version="[2.10,3)",org.apache.camel.builder;version="[2.10,3)",org.apache.camel.model;version="[2.10,3)",org.apache.commons.logging,org.osgi.service.blueprint;version="[1.0.0,2.0.0)"
Tool: Bnd-1.50.0
```

Figure 4-24 Manifeste OSGi

6 Conclusion

Dans ce chapitre, nous avons fait le tour de l'architecture logicielle que nous proposons pour notre solution. Nous avons présenté les critères importants dans le choix d'un ESB en décrivant brièvement les solutions ESB les plus pertinentes du marché en évoquant à la fois les solutions gratuites et payantes. Nous avons décidé de nous baser sur Apache Karaf afin de construire une solution adaptée au besoin de notre système.

Nous avons expliqué l'implantation de nos différents outils décrits au chapitre 2 avec des technologies tels qu'Apache Camel, Archiva et Maven. Nous avons expliqué les modifications que nous avons dû

apporter aux frameworks utilisés afin d'élaborer leurs implantation à partir de nos spécifications. Nous avons aussi décrit les principaux ajouts en termes d'implantation et de configuration qui nous ont permis d'enrichir les frameworks pour y introduire la migration d'agents d'orchestration.

L'importance de la notion de cluster, nous conduit à fournir notre vision du mécanisme de clustering en la comparant avec les approches classiques et les frameworks dédiés pour le faire. Nous avons expliqué les mécanismes de déploiement liés à notre environnement de déploiement clustérisé. De plus nous avons décrit le cycle de vie d'un agent d'orchestration qui se base sur le framework OSGi pour une gestion modulable des agents d'orchestration et de leurs dépendances.

Dans la dernière partie, nous avons expliqué la procédure pour construire un livrable déployable à partir du code source obtenu par l'outil Pi2Camel. Nous avons aussi décrit les caractéristiques d'un artefact déployable OSGi appelé Bundle. Nous décrivons les plug-ins et métadonnées qui sont utiles pour que le conteneur Karaf puisse traiter l'agent d'orchestration.

Dans le chapitre suivant, nous utilisons ces résultats pour la phase d'évaluation. Nous intégrons un mécanisme de prise de mesures dans notre implantation. Nous décrivons des données recueillies grâce à ce mécanisme de mesure et nous les analysons dans le but de les commenter et d'en tirer des informations utiles à nos utilisateurs.