ASR9 - Projet de fin d'études

Sujet

Démonstrateur de composition de services répartis hétérogènes avec SCA

Réalisé par

Mohamed Habib ESSOUSSI et Mohamed Said MOSLI BOUKSIAA

Remerciements

Nous avons le plaisir d'exprimer nos plus sincères remerciements à tous ceux qui nous ont aidé pour la réalisation de ce projet en mettant à notre disposition les bonnes documentations et en nous soufflant les bonnes idées pour progresser. Nous voudrions très exceptionnellement remercier Madame Chantal TACONET qui a proposé ce sujet et qui a supervisé notre travail tout le long du semestre.

Résumé

Relativement nouvelle, la technologie SCA implémentée par Apache Tuscany sera objet de tests approfondis à travers plusieurs démonstrateurs où nous explorerons les limites des facultés d'hétérogénéité des composants en langages d'implémentations et en protocoles d'appel de services et de circulation des données. Ce projet vise à découvrir d'abord cette technologie à travers tests et lectures et à mettre enfin en œuvre des briques logiciels aussi hétérogènes que possible pour en tirer d'éventuelles anomalies ou limites.

Mots-clés : SCA, Apache Tuscany, applications réparties, middleware, composants

Table des matières

Table des figures

Introduction

De plus en plus, on tend à développer des applications de moins en moins coûteuses avec un minimum de code à mettre en place et à entretenir. La plus populaire des solutions est l'architecture SOA qui justement propose des briques de services prêts à l'emploi. Quoiqu'attractive, SOA est très rapidement confrontée à ses limites. En effet, le déploiement peut ne pas être simple d'autant plus que les technologies des services répartis mis à disposition peuvent être bien différentes. C'est justement là où interviennent l'architecture SCA (Service Component Architecture) et le projet Apache Tuscany qui implémente la dernière pour proposer un moyen d'enchainement des composants hétérogènes au sein d'une même application répartie. L'essentiel de notre mission est la mise en oeuvre de démonstrateurs en composants hétérogènes selon une architecture SCA et la mise en évidence d'éventuelles limites de cette technologie.

Ce projet de fin d'études, intitulé **Démonstrateur de composition de services répartis hétérogènes avec SCA**, s'inscrit donc dans le module **CSC5005** de la voie
d'approfondissement **TELECOM SudParis**. C'est un projet qui s'est étalé tout au long du semestre S9 de notre cursus sous l'encadrement de Madame **Chantal TACONET**, enseignant-chercheur à l'**Institut TELECOM**.

Le présent rapport s'articule autour de 4 chapitres comme suit :

- Architecture SCA : Une brève introduction sur l'architecture SCA et ses différentes implémentations en passant par un quelques bases et mots-clés
- Premières expériences avec SCA : Un ensemble d'applications introductifs à l'architecture SCA et qui visent la maîtrise des différentes technologies qui constitueraient notre futur démonstrateur d'hétérogénéité.
- Démonstrateur principal- Chat SCA : Description de l'architecture et de la conception des 3 versions de notre démonstrateur principal.
- Limites de SCA et Tuscany.

Chapitre 1

Architecture SCA

Sommaire

Introduction

Nous entamons ce rapport par introduire l'architecture SCA et les différentes implémentations qui existent en insistant sur l'utilisation de Java SCA runtime du projet open source Apache Tuscany.

1.1 L'architecture SCA

Service Component Architecture (SCA) [1] est un ensemble de spécifications qui décrivent un modèle pour bâtir des applications s'inscrivant dans une architecture orientée service. Les applications basées sur SCA sont en fait un assemblage de composants, chaque composant implémente une partie de la logique métier et peut dépendre de services, et à son tour exposer tout ou une partie de son comportement comme un service. Le but de SCA est de simplifier l'écriture d'application dans un cadre SOA indépendamment des produits et langages utilisés. SCA a été écrit dans le cadre de l'architecture SOA Ces spécifications se focalisent sur la partie écriture de la logique métier, qui se veut indépendante d'un langage de programmation (i.e. Java, C++, BPEL, Python etc) et du protocole d'appel de services et de circulation des données (services web, JSON, RMI, CORBA...). Ainsi, avec SCA, les services se manipulent tous de la même manière quelles que soient leurs caractéristiques techniques dans la limite des technologies supportées par SCA.
[2] Le diagramme suivant est une vue haut-niveau de SCA runtime (logiciel d'exécu-

1. *Service Component Architecture*, Wikipedia, **[en ligne]** sur http ://fr.wikipedia.org/wiki/Service_Component_Architecture, visité en Janvier 2012
2. *Apache Tuscany SCA Java Architecture Guide*, **[en ligne]** sur http ://tuscany.apache.org/sca-java-architecture-guide.html, visité en Janvier 2012

tion) qui consiste en un ensemble de modules décris ci-après :

- SCA Spec API : L'API définie par le client Java SCA et l'implémentation des spécifications
- API : Les APIs Tuscany APIs qui étendent les implémentations des spécifications de SCA
- Core : Développement des runtimes et des SPIs qui l'étendent
- Extensions.
- Plateformes hôtes.

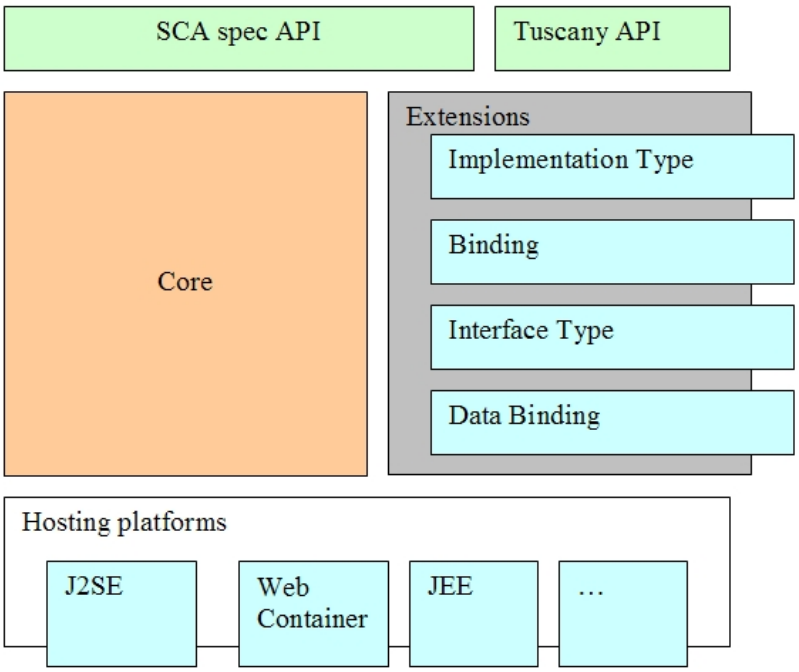

FIGURE 1.1: Diagramme de l'architecture de SCA runtime

1.2 Concepts de base et mots-clés

- **Contribution** : Une collection de fichiers composites et d'artefacts SCA indispensables pour un domaine SCA. Une contribution typique inclurait des fichiers composites, des classes et interfaces Java, des fichiers HTML etc ...
- **Composite** : Il décrit les composants de l'application, les services offerts par chaque composants, les références des services qu'utiliseront ces composants et les types ou protocoles de communications entre eux. La spécification d'un composite est textuelle est décrite par un fichier xml. Cela dit, il existe aussi une description graphique que nous utiliserons tout au long du rapport dont voici un exemple tiré du livre **Tuscany In Action**. (Pour plus de précisions, rendez-vous à la page 7 du livre) [3] :

3. Simon Laws, Mark Combellack, Raymond Feng, Haleh Mahbod, Simon Nash, *Tuscany SCA in Action*, 2007, 472 p.

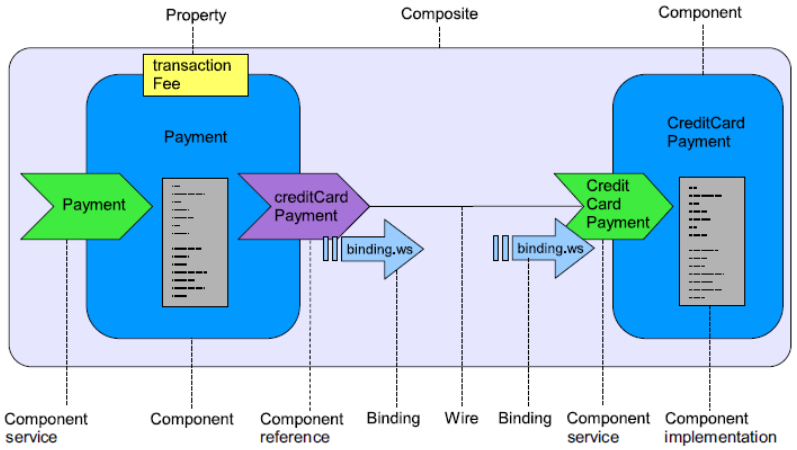

FIGURE 1.2: Exemple de schéma de composite

- **Nœud** : Dans Tuscany, chaque composite tourne dans ce qu'on appelle un nœud. Chaque nœud est configuré à travers ses contributions, ses composites, son port etc...
- **Domaine** : Là où sont configurés les contributions, composites et nœuds.

Ils existent aujourd'hui plusieurs implémentations des spécifications de SCA, y compris :

- Websphere d'IBM
- Aqualogic de BEA Systems
- Tibco ActiveMatrix de Tibco Software
- Tuscany d'Apache Software Foundation
- FraSCati, implémentation développée par l'INRIA dans le cadre du projet SCOrWare Fabric de Metaform Systems

Tout au long de notre projet, nous n'avons utilisé que Java SCA runtime du projet open source Apache Tuscany. Ils existent également d'autres implémentations d'Apache Tuscany pour SCA tel que Tuscany SCA C++, appelée aussi Native qui implémente un large choix de scripts tel que PHP et Python. L'implémentation de Java permet également l'utilisation de plusieurs scripts tel que JavaScript, Ruby, Groovy et Python quoique l'utilisation de Python est plus appropriée avec l'implémentation "Native" de Tuscany.

1.3 Environnement de développement

Système d'exploitation : Linux - Ubuntu 11.10
Java : java version "1.6.0_23"
Apache Ant : Apache Ant(TM) version 1.8.2 compiled on August 19 2011
Version Tuscany : Tuscany 1.6.2
IDE : Eclipse Indigo Service Release 1
OpenEJB : OpenEJB Standalone server version 3.1.1
Plugins :
- Apache Tuscany SCA Eclipse Tools 1.6.2
- SCA Tools 2.0
- UMLLab
- Checkstyle 2.2.1

Conclusion

Ce premier chapitre du projet constitue la synthèse de la phase de documentation et des tests d'applications introductives à cette technologie. Les bases théoriques ainsi établies, nous enchainons donc avec une phase de maîtrise et d'approfondissement avant de développer notre principal démonstrateur.

Chapitre 2

Premières expériences avec SCA

Sommaire

Introduction

Afin de concrétiser la phase de documentation, nous avons décidé de mettre au point une série d'applications "Hello World" en ciblant à chaque fois une technologie particulière à maitriser. Ceci servira par la suite comme point de départ pour la mise en œuvre de notre principal démonstrateur. Ainsi, ce chapitre décrira l'architecture et le développement de 4 applications utilisant comme technologies ce qui suit : Java, Python, JavaScript, Android, EJB, SOAP et JSON-RPC.

2.1 Première application à services répartis non hétérogènes (Java & SOAP)

L'application "Introduction Application" est une application répartie en SCA et qui constitue le premier démonstrateur de composition de services réparties en SCA quoique non encore hétérogènes. C'est une application qui appelle 2 services, l'un

pour retourner "Hello World" et l'autre pour retourner "Hello Tuscany", pour avoir à la fin les deux résultats concaténés. Il s'agit d'une application écrite entièrement en Java dont l'unique moyen d'enchainement entre les composites est les web services sous le protocole SOAP.

Sources disponibles sur :
https ://svnshare.it-sudparis.eu/csc5005/ASR9_2011_2012_G/tags/IntroductionApplication_V1.1/

2.1.1 Architecture de l'application

L'application comprend 3 composites totalement répartis n'ayant chacun qu'un seul composant :

- `HelloWorldComposite` : implémente le service `HelloWorld`
- `HelloTuscanyComposite` : implémente le service `HelloTuscany` et appelle la référence du service `HelloWorld`
- `ClientComposite` : implémente le service `Runnable` et appelle la référence du service `HelloTuscany`.

Tous les enchainements se font aus moyens de services Web sous SOAP. En effet, le service `HelloWorld` est exposé à l'adresse `http://localhost:9085/helloworldService` et le service `HelloTuscany` à l'adresse `http://localhost:9080/helloTuscany`. La figure ci-dessous décrit graphiquement l'architecture de cette application.

FIGURE 2.1: Architecture SCA - Introduction Application

2.1.2 Conception et développement

Chacun des 3 contributions constituant cette application obéit à la même logique : une interface définissant le service, son implémentation, un fichier décrivant le composite (services et références) et un lanceur (launcher) de l'application. Ci-dessous le diagramme UML de la contribution "HelloWorld".

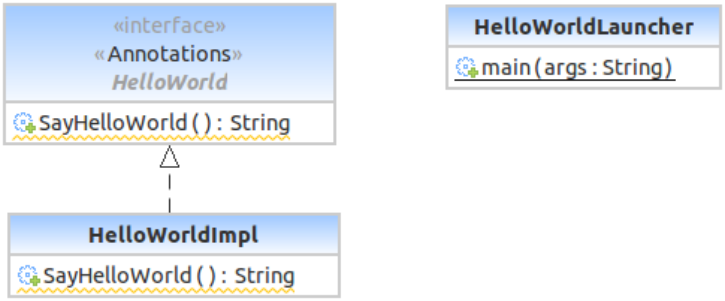

FIGURE 2.2: Diagramme UML Hello World - Introduction Application

Extrait du code du service `HelloWorld` et de son implémentation :

```
1 package helloworld;
2
3 import org.osoa.sca.annotations.Remotable;
4
5 @Remotable
6 public interface HelloWorld {
7     String SayHelloWorld();
8 }
```

```
1 package helloworld;
2
3 public class HelloWorldImpl implements HelloWorld{
4     public String SayHelloWorld() {
5         return "Hello World";
6     }
7 }
```

Plus de détails sur le fonctionnement, l'installation et le test de l'application sur :
https ://svnshare.it-sudparis.eu/csc5005/ASR9_2011_2012_G/tags/IntroductionApplication_V1.1/readme
Vous pouvez également suivre le tutoriel détaillé sur :
http ://www-tp.int-evry.fr/~essoussi/scatutos/doku.php ?id=helloworldtuscanyws

2.2 Première application à services répartis hétérogènes (Java & JavaScript & Client Android (non-SCA) sous JSON-RPC)

Cette application répartie est notre première application à briques hétérogènes. Elle inclut non seulement des implémentations en Java, mais aussi des composants en JavaScrit voire un client non SCA (ici un client Android). Il est à noter que les implémentations en scripts (JavaScript, Python, Ruby, Groovy ...) ne sont pas définies aux spécifications de SCA. Nous utiliserons dans ce projet, les implémentations d'Apache Tuscany.

Sources disponibles sur :
https ://svnshare.it-sudparis.eu/csc5005/ASR9_2011_2012_G/trunk/SCAJavascript
https ://svnshare.it-sudparis.eu/csc5005/ASR9_2011_2012_G/trunk/SCAAndroid

2.2.1 Architecture de l'application

L'application comprend 2 contributions :

- `SCAJavaScript` qui comprend un composite à deux composants : un composant `HelloJavaComponent` qui implémente le service `HelloJava` exposé en JSON-RPC à l'adresse `http://localhost:8080/HelloJava` et un composant `JavascriptClient` qui implémente le service `Widget` exposé sous HTTP à l'adresse `http://localhost:8080/client` dont l'implémentation est un fichier HTML contenant du JavaScript où on évoque la référence du service `HelloJava` du premier composant sous JSON-RPC. L'implémentation Web (HTML+JavaScript) doit pouvoir afficher au final la chaîne de caractère "Hello Java".
- `SCAAndroid` est un client non SCA pour terminal Android qui invoque le service exposé sous JSON-RPC du premier composant (Java). Cette contribution n'est donc pas un composite et il n'est donc pas possible de référencer le service en question à la manière de SCA. Il s'agit alors d'un client JSON classique qui invoque ses services à travers des requêtes saisies "manuellement".

La figure ci-dessous décrit graphiquement l'architecture de cette application.

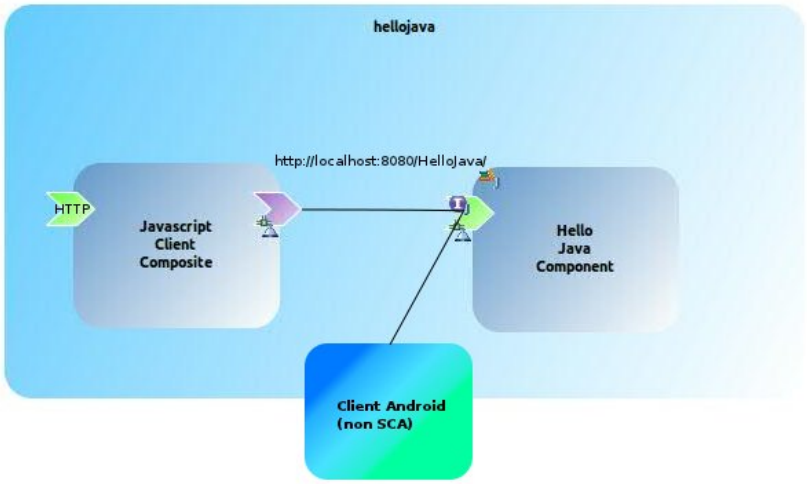

FIGURE 2.3: Architecture SCA - Application Hello Java, JavaScript et Android

2.2.2 Conception et développement

Le composant `HelloJavaComponent` obéit à la même logique de conception et développement de l'application "Introduction Application" décrite en 2.1.2. Le composant `JavascriptClientComponent` est décrit par son implémentation, son service fourni et l'appel de la référence du service `HelloJava`. En effet, l'implémentation du composant est un fichier HTML (`client.html`) qui inclut un fichier JavaScript généré automatiquement par Tuscany (`client.js`) et du code JavaScript développé par nous même pour illustrer l'appel à la référence du service `HelloJava` dont voici un extrait du code :

```
/*@Reference*/
var hellojava = new tuscany.sca.Reference('hellojava');

function hellojava_gethelloJavaResponse(hello) {
    document.getElementById('hello').innerHTML = hello;
}

function sayHelloJava() {
 hellojava.helloJava(hellojava_gethelloJavaResponse);
}
```

Par ailleurs, n'étant pas un client SCA, le développement du client Android est bien plus complexe du moment qu'il faille implémenter manuellement le client JSON-RPC soit les requêtes JSON correspondantes. Dans l'extrait suivant, remarquez la précision de l'adresse du service au niveau de la constante `jsonRPCServiceURI` et la requête JSON-RPC au niveau de la constante `jsonRPCRequest`. Dans un client SCA, les informations sur la localisation du service ne se trouvent pas au code, mais plutôt au fichier composite.

```
import com.android.scaclient.json.JSONRpc;

public class SCAAndroidActivity extends Activity {
    /** Called when the activity is first created. */

    private static final String jsonRPCServiceURI = "http://157.159.42.29:8080/↩
        HelloJava";
    private static final String jsonRPCRequest = "{\"id\": 1, \"method\": \"↩
        helloJava\", \"params\": []}";

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
.. .
        try {
              json = JSONRpc.invoke(jsonRPCServiceURI, jsonRPCRequest);
            }
.. .
        }
    }
```

Vous pouvez également suivre le tutoriel détaillé sur :
http ://www-tp.int-evry.fr/~essoussi/scatutos/doku.php ?id=hellojavascriptandroid

2.3 Première expérience avec Apache OpenEJB : Composant EJB 'Hello World' et client SCA

Apache OpenEJB est un serveur d'application Java EE au même terme que Glassfish, JBOSS, Tomcat ou TomEE. Loin de ne contenir qu'un containeur EJB, il inclut toutes les fonctionalités de TomEE hormis les Servlets, JSP et JSF. C'est un serveur facile à installer, à configurer et à démarrer. Par ailleurs, notre choix s'est surtout porté sur OpenEJB vue la disponibilité d'exemples décrits au livre **Tuscany In Action**. [1]

1. Simon Laws, Mark Combellack, Raymond Feng, Haleh Mahbod, Simon Nash, *Tuscany SCA in Action*, 2007, 472 p.

Ce qu'il faut savoir sur l'association EJB-SCA c'est que jusqu'à aujourd'hui on ne peut pas créer des services SCA deployés comme étant des EJB. Cela veut dire qu'un composant EJB ne peut pas être un composant SCA. Il doit être développé et deployé sur son serveur d'application à part et appelé comme référence par d'autres composants SCA. Néanmoins, l'interface du composant EJB déployé doit être copié également au niveau du composant SCA qui le consomme. Ainsi, et pour la première fois, une référence au niveau d'un composant SCA ne désigne pas forcément une interface de service d'un autre composant SCA. Là, il s'agit d'un EJB.

Au niveau du binding de la référence (`<binding.ejb uri ... >`), une URI doit être précisée. Dans le cas d'un EJB sous OpenEJB, il suffit de mentionner le `JNDI Name` qui est souvent le nom de la classe d'implémentation de l'interface concaténé à `"Remote"`. Ce nom est facilement repérable lors du déploiement du composant EJB. Malheureusement OpenEJB n'a pas d'interface web d'administration, les méthodes sont donc assez artisanales pour se procurer ce genre d'informations. Cela dit, cette aisance au niveau du binding encourage encore plus l'utilisation d'OpenEJB qui nous épargne les longues recherches des syntaxe des URI de référencements de resources de JNDI des autres serveurs d'applications.

Sources disponibles sur :
https ://svnshare.it-sudparis.eu/csc5005/ASR9_2011_2012_G/trunk/SCAEJB/

2.3.1 Architecture de l'application

Cette application comprend 2 contributions :
- Un composant EJB (3.0) `Sless` (non SCA) déployé sur un serveur d'application OpenEJB et qui expose son service Hello World.
- Un composant client SCA `HelloWorldClientEJB` qui appelle la référence du service Hello World de `Sless`.

Le client SCA implémenté en Java doit pouvoir afficher la chaîne "Hello, World" transmise par le composant EJB.

2.3.2 Conception et développement

Conceptuellement, cette application n'a rien de particulier à signaler. Côté composant EJB, il ne s'agit que d'une simple interface Java et de son implémentation ainsi qu'un lanceur du serveur OpenEJB.

En effet, le composant `Sless` est un EJB Session `''Stateless''` qui n'offre comme service que la chaîne "Hello World". Un fichier `ejb-jar.xml` est introduit sous le répertoire `META-INF` pour le déploiement automatique du composant EJB sur OpenEJB. Le démarrage du serveur et le déploiement se font au niveau de la classe launcher `ejb30.launcher.OpenEJbLauncher`.

Ci-dessous les extraits de code de l'interface du service et son implémentation :

```
1 import javax.ejb.Remote;
2
3 @Remote
```

```java
public interface Sless {
    public String hello();
}
```

```java
import javax.ejb.Stateless;

@Stateless
public class SlessBean implements Sless {
    public String hello() {
        return "hello, world!\n";
    }
}
```

Au niveau du binding (au fichier composite) du composant client, il a suffit à spécifier comme "uri" le `JNDI Name` qui est en l'occurence `''SlessBeanRemote''`.

Le client SCA appelle classiquement la référence du service offert par le composant EJB, comme le montre le code ci-après :

```java
package clientEJB.sca;

import org.osoa.sca.annotations.Reference;
import org.osoa.sca.annotations.Service;

import ejb30.Sless;

@Service(Runnable.class)
public class ClientEJB implements Runnable{

    @Reference
    protected Sless sless;

    public void run() {
        System.out.println(sless.hello());
    }
}
```

Au niveau du Launcher, quelques propriétés doivent être modifiées au context à savoir le `INITIAL_CONTEXT_FACTORY` et le `PROVIDER_URL` (port 4201) comme le montre l'extrait de code ci-dessous :

```java
.. .
public class Launcher {
    public static void main(String[] args) {

        System.setProperty(Context.INITIAL_CONTEXT_FACTORY, "org.apache.
        openejb.client.RemoteInitialContextFactory");

        System.setProperty(Context.PROVIDER_URL, "ejbd://localhost:4201");

        SCADomain scaDomain = SCADomain.newInstance("client.composite");

        Runnable use = scaDomain.getService(Runnable.class,
                "ClientComponent/Runnable");
        use.run();
.. .
    }
}
```

Conclusion

Cette étape du projet nous a permis non seulement d'acquérir le nécessaire pour bâtir un démonstrateur poussé d'hétérogéneité, mais aussi de nous fixer *a priori* sur les technologies à garder parmi celles qui ont été testées. Il a été également décidé d'abandonner le développement d'un client pour terminal Android, n'étant pas un client SCA et ne répondant donc pas aux objectifs du projet. En revanche, nous maintenons les implémentations en Java, EJB, Python et JavaScript.

Chapitre 3

Démonstrateur principal : Chat SCA

Sommaire

Introduction

Dans ce chapitre, nous arrivons à la description de notre démonstrateur principal, l'application où nous avons essayé de profiter de ce qu'offre le standard SCA et son implémentation Tuscany, pour composer des services à la fois hétérogènes et utiles. Nous avons choisi d'implémenter une application de chat qui comprend un serveur et des clients. L'idée a ensuite évolué apportant de nouveaux composants à l'application. C'est pour cela que cette dernière existe en 3 versions que nous allons présenter successivent, en argumentant les choix des technologies et de l'architecture.

3.1 Chat SCA, Version 1.0 : Serveur, Administrateur et Client

C'est la première version fonctionnelle de notre application. Elle permet à des clients désignés à l'avance de discuter entre eux après authentification. Les utilisateurs connectés ont accès à une session de chat, une session nouvelle étant démarrée

à chaque fois que le serveur est remis en marche.

Sources disponibles sur :
https ://svnshare.it-sudparis.eu/csc5005/ASR9_2011_2012_G/tags/ChatSCA_V1.0/

3.1.1 Architecture

L'application comprend 2 composites comme suit :

– `PythonChatServer` : comporte deux composants `AdminComponent` et `PythonServerComponent`. Ce dernier implémente les services `ChatServer` et `AdminServer`. En effet, ce composant propose deux sortes de services : d'une part, un service d'authentification proposé aux utilisateurs qui souhaitent participer à la discussion (partie cliente), auxquels il offre aussi la possibilités de poster des messages et de consulter les nouveaux messages. Ce premier service est celui qu'on a appelé `ChatServer` qui s'expose en service web sous SOAP à l'adresse `http://localhost:9000/server` D'autre part, le service `AdminServer` est offert par ce composant à l'administrateur `AdminComponent` sous JSON-RPC à l'adresse `http://localhost:8080/admin` et lui permet de s'authentifier à travers son interface web disponible sur `http://localhost:8080/webadmin/` pour pouvoir après ajouter/supprimer des utilisateurs. Voici le schéma de ce composite :

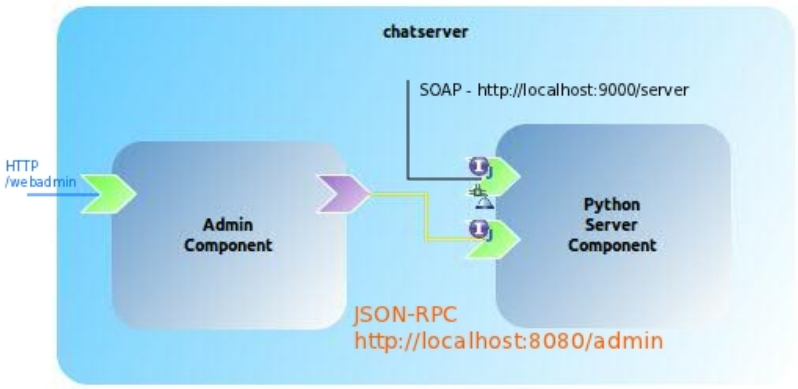

FIGURE 3.1: Architecture du composite serveur

– `SCAClient` : Ce composite est notre client de chat. il a besoin des services offerts par le composant `PythonServerComponent` pour pouvoir s'identifier et échanger des messages avec les autres utilisateurs.

La figure suivante résume l'architecture de l'ensemble de l'application dans sa première version.

3.1.2 Conception et développement

Les composants administrateur et serveur de messagerie de notre composite serveur sont développés en Python et en Javascript respectivement. Le choix du langage python réside dans la facilité de programmation avec ce langage, notamment dans

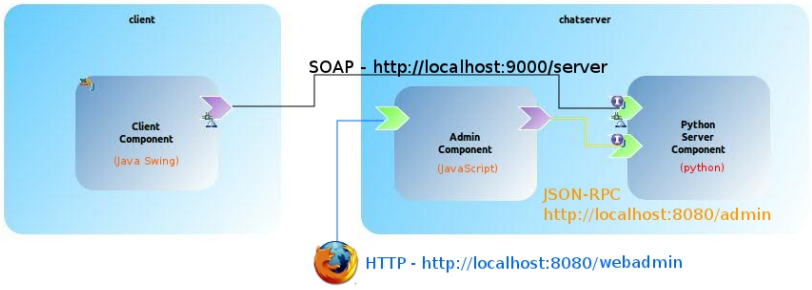

FIGURE 3.2: Architecture de la version 1.0

la manipulation des collections (listes, dictionnaires) et des chaînes de caractères. Le javascript est choisi pour avoir un "client" web dont voici une capture d'écran :

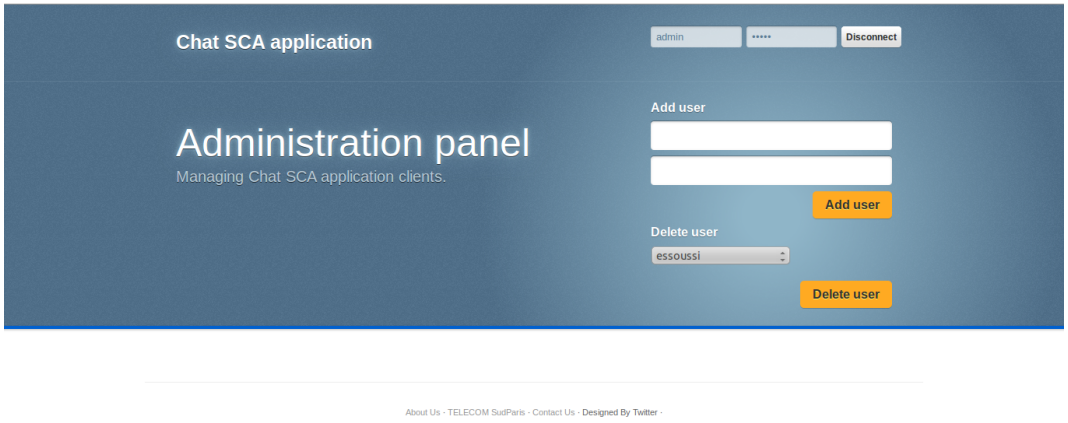

FIGURE 3.3: Interface web de l'administrateur - http ://localhost :8080/webadmin

Le deuxième composite et réalisé en Java couplé avec la librairie Swing. En voici le diagramme UML et une capture écran :

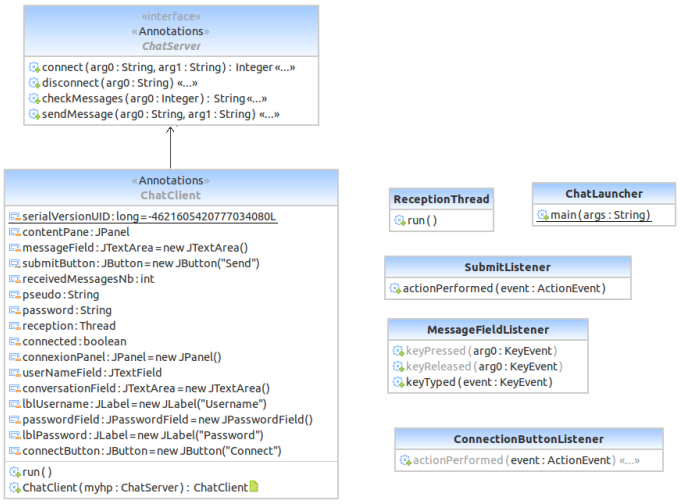

FIGURE 3.4: Diagramme UML du composant client

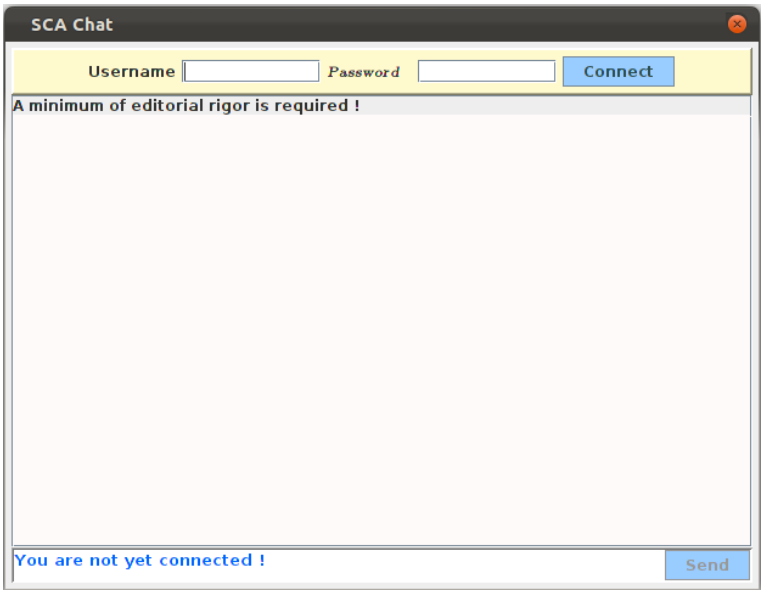

FIGURE 3.5: Interface Java Swing du client

Voici de petits extraits de la classe qui implémente notre partie cliente :

```
1  @Service(Runnable.class)
2  public class ChatClient extends JFrame implements Runnable {
3
4      .. .
5
6      /* Declaration of the reference that will give the access to the services */
7      @Reference
8      protected ChatServer hp;
9
10     .. .
11
12     /* This class listens to the events related to the button which is used
13     to send messages */
14
15     class SubmitListener implements ActionListener {
```

```
16
17          public void actionPerformed(ActionEvent event) {
18              String messageToSend = messageField.getText();
19
20              /* Access to the service sendMessage by the reference variable hp */
21              hp.sendMessage(pseudo, messageToSend);
22          }
23      }
24
25      .. .
26
27      /* This class is used to regularly verify whether there are new messages */
28      class ReceptionThread implements Runnable {
29
30          public void run() {
31
32              String[] newMessages;
33              String messages;
34
35              while (connected) {
36                  try {
37
38                      // The thread sleeps 1 second then looks for the new messages
39                      Thread.sleep(1000);
40
41                  } catch (InterruptedException e) {
42                      e.printStackTrace();
43                  }
44
45                  /* The service returns a String which contains the new messages
46                  separated by a '\n' character. This point will be explained after
47                  these code lines */
48                  messages = hp.checkMessages(receivedMessagesNb);
49
50                  /* Extraction of a list of messages from the String returned
51                  by the server */
52                  newMessages = messages.split("\n");
53
54                  .. .
55              }
56          }
57      }
58  }
```

Le composant en python stocke la liste des utilisateurs dans un dictionnaire facilitant la recherche ou la suppression d'un utilisateur par son pseudo. De même, il garde les messages dans une liste qu'il doit envoyer au composant client à chaque fois qu'il invoque le sevice `checkMessages()`. Le problème réside dans l'impossibilité d'envoyer une liste de type python au composant client écrit en java. C'est pour cela que la liste est transformée au niveau du serveur en une chaîne de caractères qu'il envoie au client qui la "splitte". Pour ne pas être obligés de faire cela, nous avons apporté un changement majeur à la technologie utilisée niveau serveur que nous allons décrire dans le paragraphe sur la version 3. On distingue les utilisateurs connectés de ceux qui ne sont pas encore connectés, ce qui nous amène à l'emploi des structures suivantes :

```
1  #this dictionary contains the users : the keys are the usernames
2  #while the values are the passwords
3  users = {}
4
5  #the list of the connected users
6  connectedUsers = []
7
```

```
#the list of the messages
messages = []
```

Les services offerts à l'administrateur et qui sont décrits au paragraphe précédent sont implémentés à l'aide de fonctions, dont voici les entêtes :

```
def connectAdmin(pseudo, password):
    """Service that makes the connection for the administrator"""

def disconnectAdmin(pseudo):
    """Service that allows the administrator to disconnect"""

def addUser(pseudo, password):
    """Service that allows the administrator to add a user"""

def deleteUser(pseudo):
    """Service that allows the administrator to delete a user"""

def getUsers():
    """Service that returns the list of users to the administrator"""
```

La manière simple d'ajout/suppression d'un utilisateur est illustrée par les lignes de code suivantes :

```
#Ajout
users[pseudo] = password

#Suppression
del users[pseudo]
```

Les services offerts à un client de chat (l'application : java, swing) sont implémentés par les fonctions suivantes :

```
def connect(pseudo, password):
    """Service that lets a user connect"""

def disconnect(pseudo):
    """Service that lets a user disconnect"""

def checkMessages(received):
    """Service that allows the client to send the number of messages he has
    already received so as to receive the new messages"""

def sendMessage(pseudo, msg):
    """Service available to send a message, the client component sends the
    username as well in order that the server displays it with the message"""
```

3.2 Chat SCA, Version 2.x : Un composant EJB de modération

Cette nouvelle version vient avec l'idée d'un composant d'auto-modération des messages à transmettre que nous intercalons entre le client et le serveur. Ainsi les messages entrés par les utilisateurs n'arrivent au serveur qu'après avoir été traités, dans une première version, contre les mots vulgaires du registre familier, en masquant ceux-ci avec des étoiles. Cela dit, cette idée nous a été dictée par des contraintes liées à l'architecture l'application. En effet, devant l'importance d'inclure

un composant non SCA tel qu'un composant EJB pour enrichir notre démonstrateur d'hétérogéneité, nous nous sommes heurtés à la contrainte qu'un composant EJB ne communique qu'avec des composants implémentés en Java. Or, jusqu'à ici, notre seul composant en Java est notre composant client (Swing). Il a été donc question de trouver une idée pour lier directement le composant EJB avec, et exclusivement avec notre composant client d'où l'idée de la modération au niveau client.

Sources disponibles sur :
https ://svnshare.it-sudparis.eu/csc5005/ASR9_2011_2012_G/tags/ChatSCA_V2.0/
https ://svnshare.it-sudparis.eu/csc5005/ASR9_2011_2012_G/tags/ChatSCA_V2.1/
https ://svnshare.it-sudparis.eu/csc5005/ASR9_2011_2012_G/tags/ChatSCA_V2.2/
https ://svnshare.it-sudparis.eu/csc5005/ASR9_2011_2012_G/tags/ChatSCA_V2.2.1/

3.2.1 Architecture

L'unité de modération, nommée `ModerationUnit` comprend à la version 2.0 un composant EJB (non SCA) déployé sur son propre serveur d'application "Apache OpenEJB" et qui est en communication avec `SCAClient` en Java Swing. Le composite `PythonChatServer` n'admet aucun changement au niveau de son architecture et continue donc de délivrer ses services d'administration et d'utilisation à travers ses deux composants (décris à la section 3.1.1). En revanche, le composite client `SCAClient` consomme désormais une nouvelle référence, celle du service de modération offert par le composant EJB de la `ModerationUnit` inscrit à l'annuaire JNDI sous le nom de `ModerationImplRemote` (tous les détails sur la mise en œuvre de la lisaison SCA - EJB sont décris à la section 2.3)

La figure suivante décrit graphiquement l'architecture de la version 2.0 :

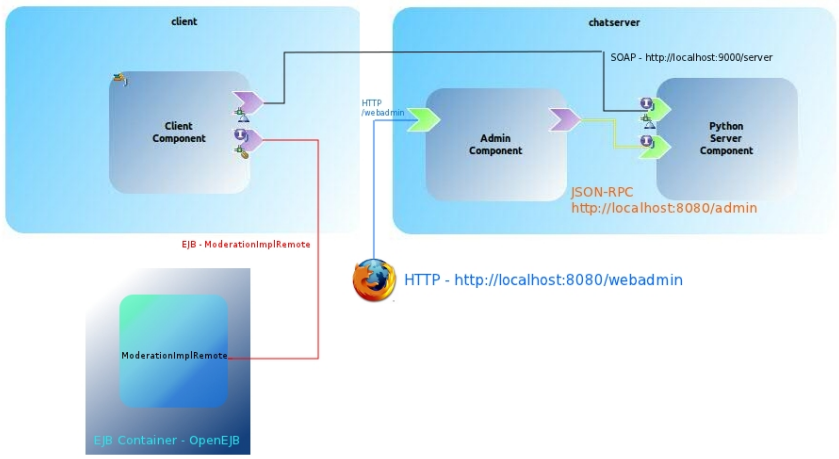

FIGURE 3.6: Architecture SCA - Chat SCA version 2.0

A la version 2.2, nous proposons un nouveau composant EJB `ModerationSubsImplRemote` au sein de l'unité `ModerationUnit` qui susbtitue le langage SMS et GEEK par des mots de registre courant. Bien évidemment, la liste que nous proposons n'est pas exhaustive. Nous nous contentons en effet de quelques substitutions dans l'unique but de valider notre démonstrateur. En voici quelques unes :

Langage SMS et GEEK 1	Langage courant 2
kawai	mignon
leet	élite
THX	Merci
PLZ	S'il vous plaît
BRB	Je reviens bientôt
wOOt	Youpi !
KTHXBYE	Ok merci, au revoir

La figure ci-dessous décrit graphiquement l'architecture de la version 2.2.

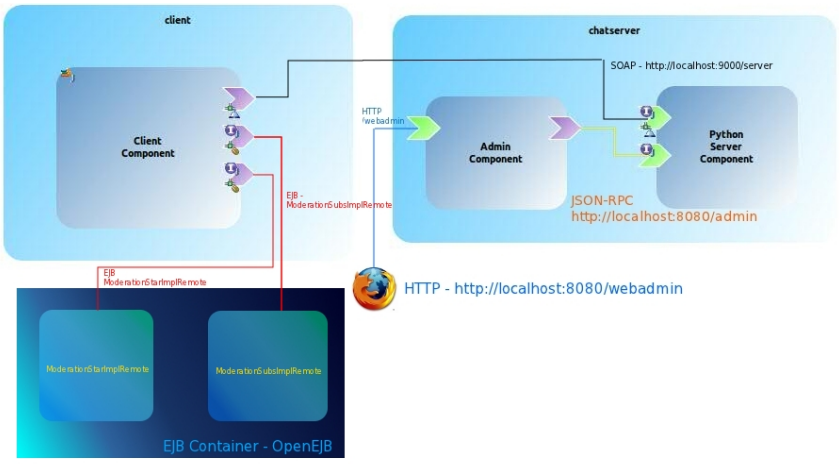

FIGURE 3.7: Architecture SCA - Chat SCA version 2.2

3.2.2 Conception et développement

Dans cette partie nous nous intéresserons à la conception et au développement de l'unité de modération `ModerationUnit` à sa version 2.0 (composant EJB `ModerationImplRemote` de masquage des mots vulgaires) et à sa version 2.2 (ajout d'un composant EJB `ModerationSubsImplRemote` de substitution de mots SMS et Geek par du langage courant).
Typiquement, un composant EJB 3.0 est décrit par son service présenté comme une interface Java et son implémentation. En l'occurence, à la version 2.0, la classe `ModerationImpl` implémente l'interface `Moderation` qui est donc `Remotable` comme décrit au diagramme UML suivant :

Algorithmiquement parlant, le composant EJB en question retourne une chaine de caractères où il remplace les vulgarités du message tapé par l'utilisateur en entrée par des étoiles (*) au nombre de caractères du mot masqué. Ceci est exécuté grâce à la méthode `replaceAll` de la classe `String` de Java qui prend en paramètre entre autre une expression régulière qu'on bâtit comme suit :
Etant donné un mot de 3 caractères **abc** de notre liste de mots à modérer, l'expression régulière serait donc **[aA][bB][cC]**. Ainsi indépendemment de la case (aBc, AbC etc...), le mot sera masqué par *** et à la fin du message, sera concaténée la chaine `[auto-moderated]`. Voici le code qui implémente l'algorithme décrit ci-haut :

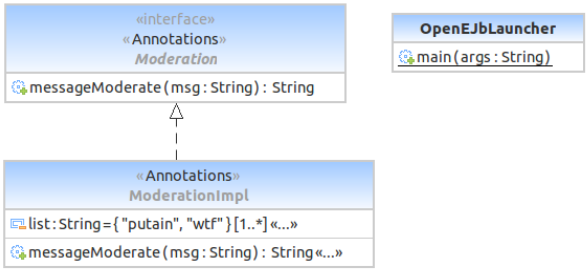

FIGURE 3.8: Diagramme UML de l'unité de modération - Chat SCA version 2.0

```
1  /* list variable contains the list of words to be moderated*/
2  for (String str : list) {
3              replacement = "";
4              regex = new char[str.length()* 4];
5
6              for (int i = 0; i < str.length(); i++) {
7                  replacement += "*";
8                  int j = 4 * i ;
9                  regex[j] =  '[';
10                 regex[j + 1] = str.toLowerCase().charAt(i);
11                 regex[j + 2] = str.toUpperCase().charAt(i);
12                 regex[j + 3] = ']';
13             }
14             /* moderating the message according to the built regex*/
15             modifiedMessage = modifiedMessage.
16                     replaceAll(new String(regex), replacement);
17         }
18         if (!modifiedMessage.equals(messageCopy)) {
19             modifiedMessage += " [auto-moderated]"; }
```

À la version 2.2, la `ModerationUnit` comprend désormais deux composants EJB : `ModerationStarImplRemote` (l'ex `ModerationImplRemote`) et `ModerationSubsImplRemote` (anti SMS et GEEK). Chacun des deux classes `ModerationStarImpl` et `ModerationSubsImpl` implémentent désormais et respectivement les interfaces `ModerationStar` et `ModerationSubs`. Celles-ci sont `Remotables` et étendent l'interface `Moderation` qui n'est donc plus `Remotable` comme le décrit le diagramme UML suivant :

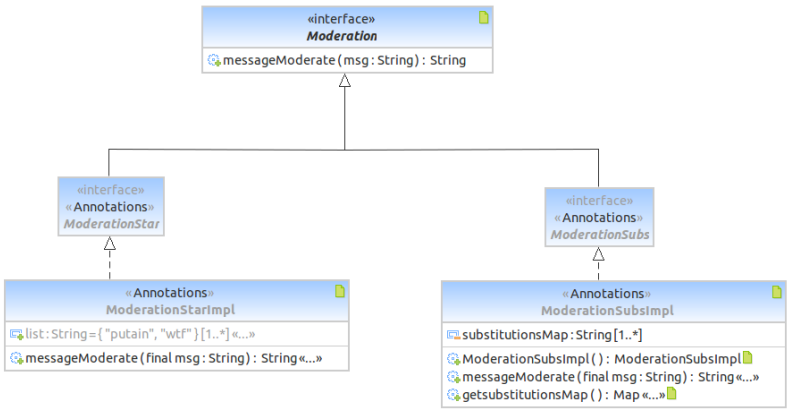

FIGURE 3.9: Diagramme UML de l'unité de modération - Chat SCA version 2.2

La structure algorithmique de la modération des mots SMS et GEEK est analogue à la structure algorithmique de la modération des mots vulgaires dans le sens où le remplacement par des étoiles (*) est substitué par une correspondance à partir d'une structure `HashMap` entre des mots SMS et GEEK et des mots du registre courant. En voici un extrait de code :

```
1  /* substitutionsMap variable is a HashMap which matches
2  the SMS and GEEK words by correct words */
3  Set<String> keySet = substitutionsMap.keySet();
4
5          for (String str : keySet) {
6              regex = new char[str.length() * 4];
7
8              for (int i = 0; i < str.length(); i++) {
9                  int j = 4 * i;
10                 regex[j] =  '[';
11                 regex[j + 1] = str.toLowerCase().charAt(i);
12                 regex[j + 2] = str.toUpperCase().charAt(i);
13                 regex[j + 3] = ']';
14             }
15             /* moderating the message according to the built regex*/
16             modifiedMessage = modifiedMessage.
17                     replaceAll(new String(regex), substitutionsMap.get(str));
18             if (!modifiedMessage.equals(messageCopy)) {
19                 /* The result is concatenated to the string
20                 [auto-antiGeek-moderated]*/
21                 modifiedMessage += " [auto-antiGeek-moderated]";
22             }
23         }
```

Quelques modifications sont également à signaler au niveau du client `SCAClient`. En effet, il consomme de ce fait 3 références dont 2 de modération. Cette dernière se fera désormais en deux étapes soit l'appel au service qui vire les "gros" mots et ensuite imbriquer le résultat au service qui modère le langage SMS et GEEK. Voici une capture d'écran du résultat obtenu au client :

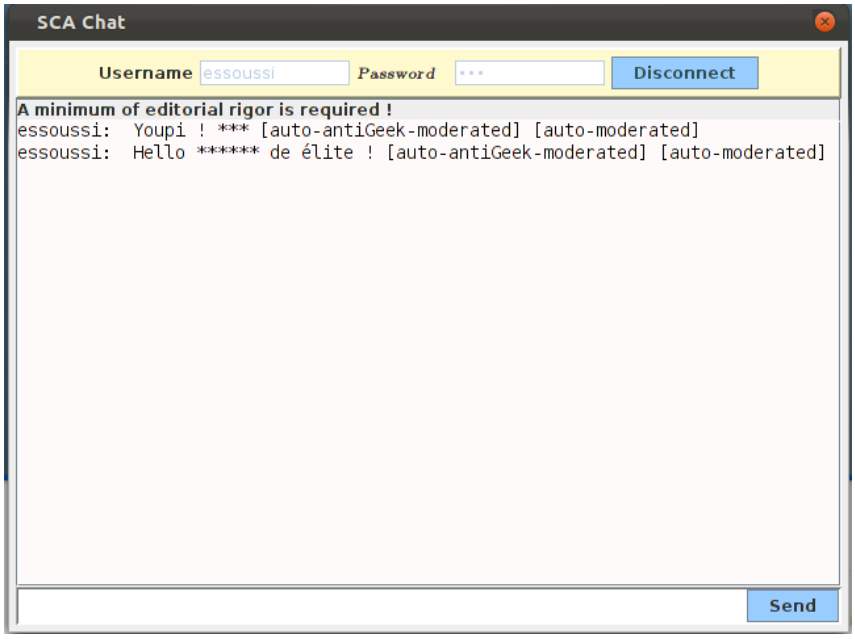

FIGURE 3.10: Client - Chat SCA version 2.2

3.3 Chat SCA, Version 3.x : Un nouveau composant serveur (gestion des messages) et introduction de la persistance

Cette nouvelle version est principalement le fruit de l'idée suivante : Si nous sommes obligés, à chaque fois que nous voulons transmettre la liste des nouveaux messages à un utilisateur, de transformer cette liste en chaîne de caractères et de regénérer la liste au niveau client, alors nous n'avons pas gagné grand chose avec l'hétérogénéité. Certes, nous avons fait communiquer un composant en python avec un composant en java. Cependant, les limites de cette communication commenceraient vite à se montrer si on essayait quelque chose de plus compliqué. La solution que nous avons utilisée est expliquée dans le paragraphe suivant. La raison qui nous empêche d'envoyer des listes en python à un composant java est explicitée dans le quatrième chapitre sur les limites de SCA.

Sources disponibles sur :
https ://svnshare.it-sudparis.eu/csc5005/ASR9_2011_2012_G/tags/ChatSCA_V3.0/
https ://svnshare.it-sudparis.eu/csc5005/ASR9_2011_2012_G/tags/ChatSCA_V3.1/

3.3.1 Architecture

Notre solution consiste à ne laisser au composant en python `PythonServerComponent` que la gestion des utilisateurs et d'en enlever la gestion des messages que nous attribuons à un nouveau composant au sein du composite `ChatServer` et que nous avons appelé `JavaServerComponent`. Nous avons profité du fait que le nouveau composant est en java pour introduire à notre application un binding en java RMI qui justement relie le `JavaServerComponent` au composant client `SCAClient`. Voici un extrait du fichier qui décrit l'architecture du composite `ChatServer`. Dans cet, extrait, il s'agit seulement du nouveau composant `JavaServerComponent` :

```
1 <component name="JavaServerComponent">
2     <implementation.java class="chatserver.message.MessagesManagerImpl" />
3     <service name="MessagesManager">
4         <interface.java interface="chatserver.message.MessagesManager" />
5         <tuscany:binding.rmi host="localhost" port="8099"
6          serviceName="MessagesManagerRMI" />
7     </service>
8 </component>
```

EXPLIQUER LE BINDING !!!!!!!!!!!!

Les figures qui suivent résument la nouvelle architecture du composite serveur `ChatServer` et de toute l'application dans sa version finale :

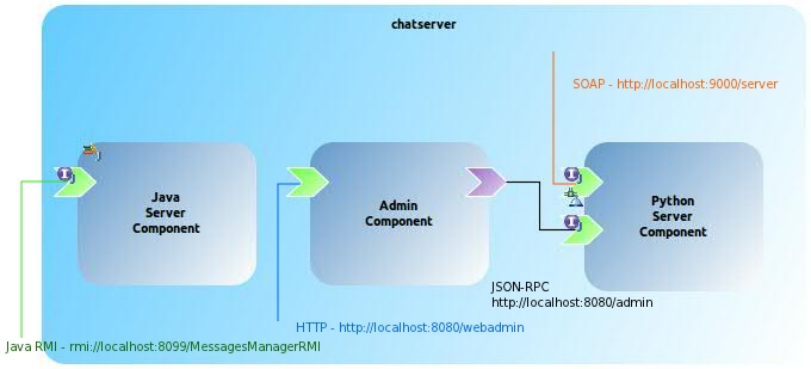

FIGURE 3.11: Architecture SCA - Le composite `ChatServer`

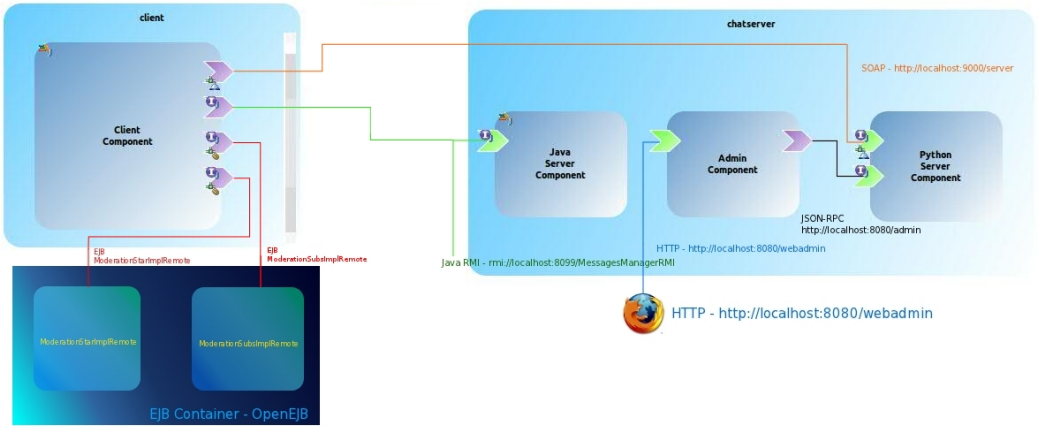

FIGURE 3.12: Architecture SCA - Chat SCA version 3

3.3.2 Conception et développement

Cette nouvelle version comporte un module java qui implémente les services du nouveau composant `JavaServerComponent` en l'occurrence l'envoi et la réception de messages. On constate alors l'apparition de l'interface `MessagesManager.java` et de la classe qui l'implémente `MessagesManagerImpl.java`. Ces derniers font référence à la classe `Messages` qui n'est autre qu'un conteneur d'une liste de messages (String). En effet, pour des contraintes que nous expliciterons dans le dernier chapitre, nous avons eu recours à la définition d'une classe qui porte les messages à envoyer à un utilisateur afin de surpasser l'impossibilité d'envoi d'une `ArrayList` Le scéma ci-dessous donne le diagramme UML du serveur de gestion des messages :

Le module en java qui gère les messages n'a rien de compliqué, il se contente de gérer une `ArrayList` de `String` dont voici la déclaration :

```
1 /**
2  * Static list of messages to be sent or checked over JAVA RMI.
3  */
4  private static ArrayList<String> myMessages = new ArrayList<String>();
```

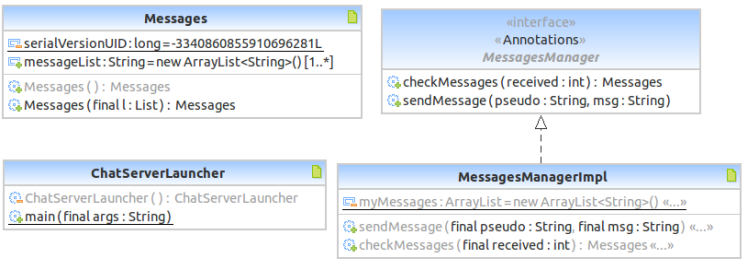

FIGURE 3.13: Diagramme UML du serveur de messages - Chat SCA version 3

Nous avons employé le mot-clé `static` après nous avoir rendu compte qu'à chaque appel du gestionnaire de messages, une nouvelle instance de celui-ci est crée, ce qui engendre la perte de tous les messages. L'emploi de `static` nous a permis de conserver la même instance de la liste des messages malgré ce fait.

Le deuxième changement important apporté à cette version consiste à persister les utilisateurs.