Chapter 1

Installation Guide

1.1 Introduction

The CHOReOS Enactment Engine (EE) provides a Platform as a Service (PaaS) that automates the distributed deployment of service choreographies in cloud environments. This chapter is targeted mainly to EE *administrators*, providing instructions about how to install, configure, and run the Enactment Engine.

We will describe now each one of the components running on the Enactment Engine execution environment. They are depicted in Fig. 1.1.

Infrastructure provider creates and destroys virtual machines (also called *nodes*) in a cloud computing environment. Currently, only Amazon EC2 and OpenStack are supported as infrastructure providers, but the Enactment Engine can be extended to support other virtualization technologies.

Chef-solo is installed by the Enactment Engine in each cloud node to manage "recipes" execution. *Chef recipes* are scripts written in a Ruby-like Domain Specific Language that implement the process of configuring operational system, installing required middleware, and finally deploying the services.

EE cliente is a script, written by deployers, that specifies the choreography deployment and invokes the EE to trigger the deployment process. *Deployer* is the human operator responsible by the deployment process.

Enactment Engine deploys choreography services according to the specification sent by the client.

1.2 Requirements

Before you run Enactment Engine, you will need:

- Git;
- Java 6 or later (we are using OpenJDK);
- Maven 3 (`http://maven.apache.org/download.html`);
- access to Infrastructure Provider services, as detailed in Section 1.3.

1.3 Cloud Provider

A CloudProvider is an Enactment Engine interface that specifies methods to CRUD virtual machines. It is expected that a Cloud Provider implementations will act just as a client of some Infrastructure Provider. In this section we describe the currently available CloudProvider implementations and how to use them. New CloudProviders may be implemented to support other virtualization tools. One example would be creating a VirtualBoxCloudProvider to create VMs using VirtualBox.

Whatever the cloud provider you choose, ensure that the required TCP ports of the created VMs are unblocked. Required ports: 22 to SSH, 8080, 8009, and 8005 to Tomcat, the ports used by your JAR services, and the port 8180 to EasyESB.

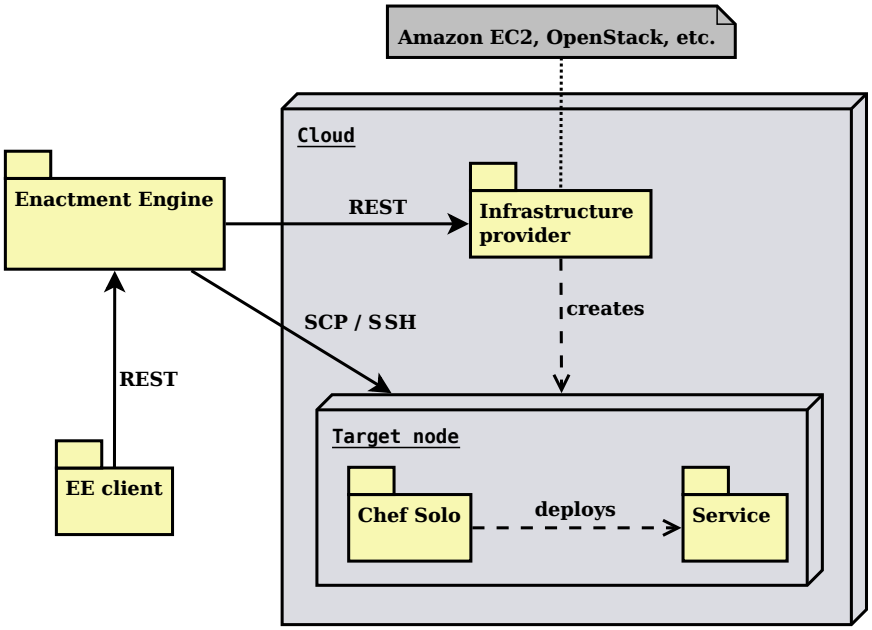

Figure 1.1: Enactment Engine execution environment.

1.3.1 Amazon EC2

Amazon EC2 service is the simplest choice to dynamically retrieve VMs as you need. You need just to create an account at `http://aws.amazon.com` and configure a pair of keys to access the VMs through SSH. The trade-off is that you must pay to Amazon!

Some hints:

- Request credits to education purposes: `http://aws.amazon.com/grants/`. The first of us earned $500, but the others $100. Maybe it depends on your research project description.
- Initially there is a limit of 20 VMs that you can run simultaneously.
- Request to increase Amazon EC2 instance limit: `http://aws.amazon.com/contact-us/ec2-request/`. At USP we got a 50 VMs limit.
- If you are going to use the EC2 API directly, pay attention to the "one-second rule": `http://www.a2sdeveloper.com/page-working-with-the-one-second-rule.html`. Nonetheless, Enactment Engine already implements the enforcement of this rule.
- As you will be charged per hour, don't forget to shutdown/stop unused VMs.
- You can use the Amazon EC2 web console to unblock TCP ports necessary to the choreography execution.
- You can also use the `ec2` command line tools to manage your VMs.
- The Enactment Engine creates VMs in the "US East" Amazon datacenter.
- The SSH keypairs are datacenter-dependent. Therefore if you create a keypair in the EU datacenter, it won't be valid for your VMs.

1.3.2 OpenStack

OpenStack is an open source private cloud platform that provides services to retrieve VMs as you need, in the same way that Amazon EC2. However, you must install OpenStack in your own infrastructure,

which means you must own at least a little cluster (or a very powerful machine) to host the created VMs. Moreover, the OpenStack installation and configuration is not a simple task.

Some hints:

- OpenStack does not provide public IPs, therefore some VPN configuration is necessary to log into the provisioned nodes.
- You can also use the `nova` command line tool to manage your VMs.
- You need to host an Ubuntu Server 12.04 image within your OpenStack infrastructure. You will use the ID of this image to configure Enactment Engine.

1.3.3 Fixed cloud provider

If you are learning how to use the Enactment Engine and want just to experiment it, the FixedCloudProvider may be the most suitable option to you. It is also useful if you want to use Enactment Engine with your own already existing cluster machines. With it you are responsible to manually creating and setting virtual machines and telling to Enactment Engine which machines must be used by it. This avoids the overhead of dealing with a cloud environment.

When creating a virtual machine to be used by the Enactment Engine, be sure:

- to use the Ubuntu 12.04 as operating system;
- it is possible to SSH into the node without typing a password: `http://www.integrade.org.br/ssh-without-password`[1];
- use sudo in the machine without typing a password: type `$sudo visudo` and add the line `<user> ALL = NOPASSWD: ALL` at the end (change `<user>` by the actual user);
- to synchronize the machine clock: `#ntpdate ccsl.ime.usp.br`;

To verify if your VM was properly set, you may run the org.ow2.choreos.deployment.nodes.cloudprovider.FixedConnectionTest test.

The Enactment Engine *will not* take care of bootstrapping (installing Chef) on your machines, since this process is taken only when creating new machines. You *must* bootstrap your machines by running the org.ow2.choreos.deployment.nodes.cm.BootstrapFixedMachines class. When you run the BootstrapFixedMachines class, Enactment Engine will bootstrap the configured fixed machines. We will talk about configuration soon.

Depending on how you create your VMs, some network configuration may be needed. In case of using VirtualBox, you can refer to `http://ccsl.ime.usp.br/foswiki/bin/view/Choreos/VMs`.

1.4 Checkout and Compilation

To checkout the code: `git clone https://github.com/choreos/enactment_engine.git`.

After installing Maven 3, open the terminal at the `enactment_engine` folder, and run the `build.sh` script. It can take several minutes. Internet access is necessary during compilation.

1.5 Configuration

Open the folder `EnactmentEngine/src/main/resources`, and create a `ee.properties` file by copying the `ee.properties.template` file. The new properties file must be created in the same folder. Open the just created properties file and edit it following instructions on the template file. The Listing 1.2 shows an example. Do the same to the `clouds.properties.template` file; in the `clouds.properties` file you will define configuration to access your infrastructure provider.

Listing 1.1: ee.properties example.

```
1 EE_PORT=9102
2
3 # Value must be a <cloud account name> in clouds.properties
```

[1] Obs: do not use a key with password.

```
4  DEFAULT_CLOUD_ACCOUNT=MY_AWS_ACCOUNT
5
6  # Values in node_selector.properties
7  NODE_SELECTOR=LIMITED_ROUND_ROBIN
8
9  # Maximum number of VMs that can be created; set if using NODE_SELECTOR=LIMITED_ROUND_ROBIN
10 VM_LIMIT=10
11
12 # Creates a reservoir of extra VMs to make the deployment faster and more scalable.
13 # The trade-off is the cost of some more VMs.
14 # If the pool size reaches the threshold, the pool size is increased by one.
15 # To not increase your pool size, set threshold as negative or greater than the initial pool size.
16 RESERVOIR=true
17 RESERVOIR_INITIAL_SIZE=5
18 RESERVOIR_THRESHOLD=-1
```

Listing 1.2: cloud.properties example.

```
1  MY_CLUSTER.CLOUD_PROVIDER=FIXED
2  MY_CLUSTER.FIXED_VM_IPS=192.168.56.101, 192.168.56.102
3  MY_CLUSTER.FIXED_VM_PRIVATE_SSH_KEYS=/home/leonardo/.ssh/nopass, /home/leonardo/.ssh/nopass
4  MY_CLUSTER.FIXED_VM_USERS=choreos, choreos
5
6  MY_AWS_ACCOUNT.CLOUD_PROVIDER=AWS
7  MY_AWS_ACCOUNT.AMAZON_ACCESS_KEY_ID=SECRET
8  MY_AWS_ACCOUNT.AMAZON_SECRET_KEY=SECRET
9  MY_AWS_ACCOUNT.AMAZON_KEY_PAIR=leoflaws
10 MY_AWS_ACCOUNT.AMAZON_PRIVATE_SSH_KEY=/home/leonardo/.ssh/leoflaws.pem
11 MY_AWS_ACCOUNT.AMAZON_IMAGE_ID=us-east-1/ami-1ccc8875
12
13 MY_OPENSTACK_ACCOUNT.CLOUD_PROVIDER=OPEN_STACK
14 MY_OPENSTACK_ACCOUNT.OPENSTACK_KEY_PAIR=leofl
15 MY_OPENSTACK_ACCOUNT.OPENSTACK_PRIVATE_SSH_KEY=/home/leonardo/.ssh/nopass
16 MY_OPENSTACK_ACCOUNT.OPENSTACK_TENANT=CHOReOS_Sandbox
17 MY_OPENSTACK_ACCOUNT.OPENSTACK_USER=leofl
18 MY_OPENSTACK_ACCOUNT.OPENSTACK_PASSWORD=SECRET
19 MY_OPENSTACK_ACCOUNT.OPENSTACK_IP=http://172.15.237.10:5000/v2.0
20 MY_OPENSTACK_ACCOUNT.OPENSTACK_IMAGE_ID=RegionOne/1654b5b6-49b7-4039-b7b7-0e42e85480f4
```

The options to NODE_SELECTOR are:

ALWAYS_CREATE: a new VM is created to each deployed service instance.

ROUND_ROBIN: NodeSelector makes a round robin using the available VMs, without creating any new VM; usually it makes sense to use it only when using the fixed cloud provider.

LIMITED_ROUND_ROBIN: initially the NodeSelector behaves like the ALWAYS_CREATE, until a limit of created VMs is reached (VM_LIMIT). After this limit, the selector behaves like the ROUND_ROBIN. When using this selector, it is necessary to declare the integer VM_LIMIT property in the configuration file.

The AMAZON_IMAGE_ID enables you to specify a customized image to be used by Enactment Engine. This feature is intended to use an image of a node already bootstrapped. In this way, the bootstrap process becomes much faster. The same may be applied to the OPENSTACK_IMAGE_ID. But in both cases, the image must still provide an Ubuntu Server 12.04 system.

At the `clouds.properties`, each *cloud account* is configured by a group of properties grouped by a common prefix. Let's call this prefix as "cloud account name". These cloud account names will be compared with the `owner` attribute in the service specifications, so a service can be specified to be deployed under a specific cloud account. If there is no match, the DEFAULT_CLOUD_ACCOUNT value declared on `ee.properties` will be used as cloud account name.

Attention: inline comments are not allowed in properties files. Therefore, the following would not work: `VM\_LIMIT=3 \# how many instances we can afford to pay`.

1.6 Execution

After compiling the project, to run the Enactment Engine you have just to run the main method on the class org.ow2.choreos.ee.rest.EnactmentEngineServer.

This task can be easier accomplished if you import the Enactment Engine projects in the Eclipse IDE. After importing the project, open the menu `Window>>Preferences>>Java>>Build Path>>Classpath` variables, and set the `M2_REPO` variable pointing to your Maven repository folder, usually the `.m2/repository` folder within your home folder. Obs: we have used the Eclipse Indigo version.

Another way is using maven:

```
EnactmentEngine$ mvn exec:java
```

If you successfully start the EE, you must see the following message on the console:

```
Enactment Engine has started [http://localhost:9102/enactmentengine/]
```

To verify if it is everything OK, run the org.ow2.choreos.chors.SimpleChorEnactmentTest. This test will deploy a simple choreography composed of two services and try to invoke it.

Chapter 2

Enactment Engine API

2.1 Introduction

This chapter provides detailed information about the Enactment Engine REST API and its target mainly to choreography deployers[1]. Understanding the API enables you to write your own code to enact a choreography.

This chapter is organized as follows. Section 2.2 presents the data model that defines XML representations exchanged by API messages. Section 2.3 describes all the operations provided by the REST API, detailing parameters and return structures. Section 2.4 presents our client implementation that can be used within any Java program.

2.2 Data model

As in any API, Enactment Engine operations receive and return complex data structures representing real world concepts. Figure 2.1 presents these concepts in the UML notation. Although the REST API handles XML representations, we use here the UML notation, since it makes easier to the reader to understand the concepts.

We proceed with a brief explanation about each class:

ChoreographySpec: represents what the middleware needs to know to enact a choreography;

ServiceSpec: a super class for common data of `DeployableServiceSpec` and `LegacyServiceSpec`;

LegacyServiceSpec: represents an already existing service to used by the choreography;

DeployableServiceSpec: represents what the middleware needs to know to deploy a service (with one or more instances for load balancing);

ServiceDependency: represents dependencies among services (if `service A` invokes `service B`, we say `service A` depends on `service B`);

Choreography: provides information about a choreography instance;

Service: a super class for common data of `DeployableService` and `LegacyService`;

LegacyService: provides information about a legacy service;

DeployableService: provides information about a deployed service;

ServiceInstance: provides information about a specific instance, also called replica, of a deployed service (URI, node data etc.).

LegacyServiceInstance: provides information about a specific instance of a legacy service.

Node: information about the node, including IP address, where a service instance is running.

[1] Deployer is the human operator responsible by the deployment process.

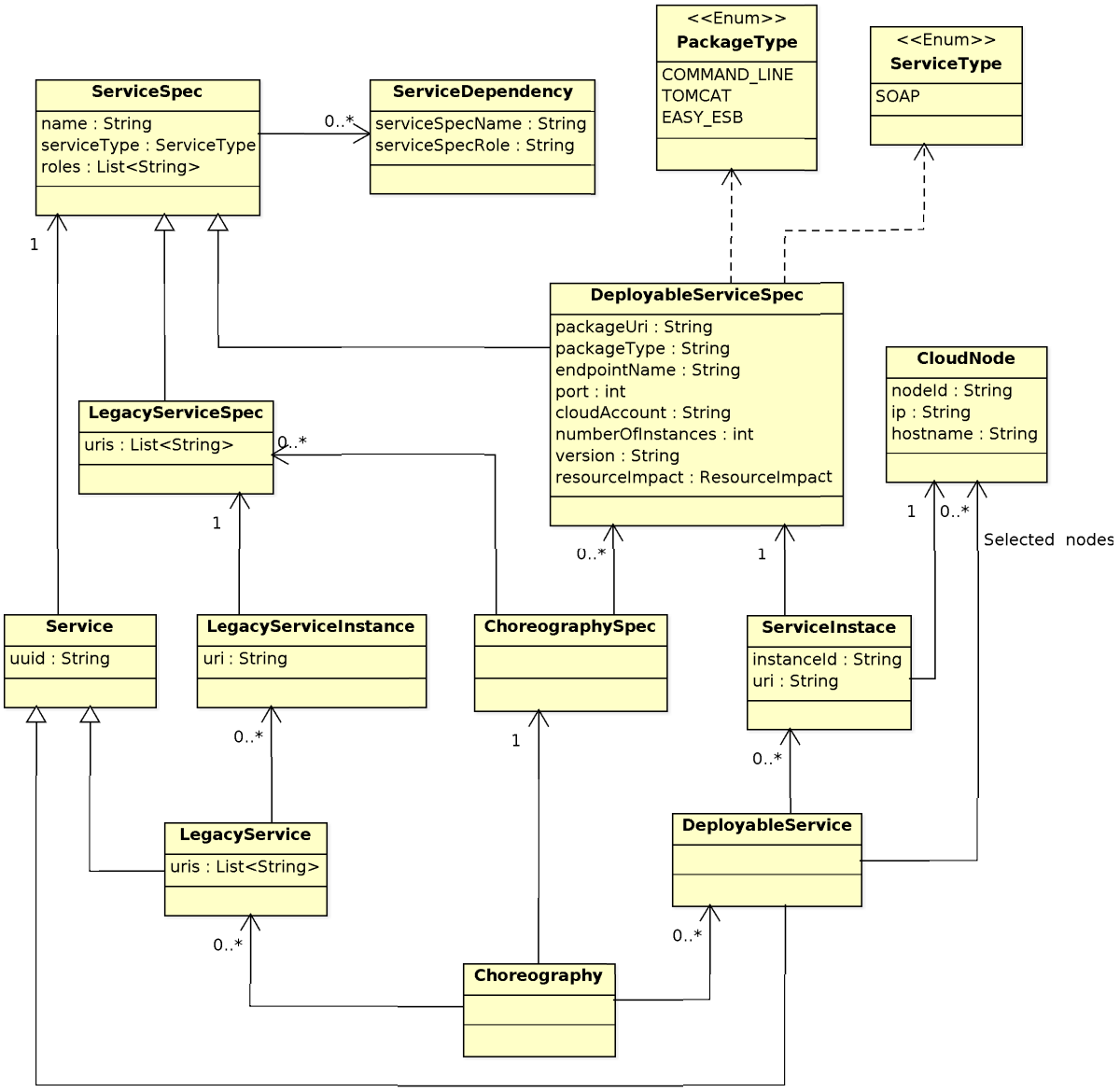

Figure 2.1: Enactment Engine REST API data model.

To request a choreography deployment, it is important to understand very well the `ServiceSpec`, `DeployableServiceSpec`, and the `LegacyServiceSpec` classes. Therefore, the description of them follows:

1. `ServiceSpec`

   name: a unique character sequence within the choreography specification;

   type: whether the service is a SOAP service or a REST service. More types can be added as necessary.

   roles: list of roles implemented by the service;

   dependencies: list of `ServiceDependency` entries; each entry describes the name of the dependency (matching the `name` attribute), and the the role provided by the dependency;

2. `DeployableServiceSpec`

   packageUri: the location of the binary file to be deployed;

   packageType: the type of the deployable package, according to the `PackageType` enumeration[2].

   endpointName: the endpoint suffix after deployment. For example, if the service will be deployed as `http://<some_ip>/choreos/service`, the endpoint name is `choreos/service`. Note that multiple replicas of a single service will all use the same endpoint;

   port: the TCP port used by the service. Note that multiple replicas of a single service will all use the same port. Mandatory if type is COMMAND_LINE;

   owner: must match a *cloud account* name configured on EE. It will define under which cloud infrastructure the service will be deployed.

   numberOfInstances: How many instances of the service should be deployed (onto different virtual machines) in order to allow the load to be distributed;

   version: the service version, which is used by the Enactment Engine to define which services must be redeployed in a choreography update (not used currently);

   resourceImpact: General information regarding the expected type of machine needed to run the service (see *Resource impact specification*);

3. `LegacyServiceSpec`

   URIs: The URIs for the various replicas of the service.

More about dependencies

In a service composition, some services depends on other services. A service that depends on other services is a *consumer* service, and the service that provides functionality to the dependent service is the *provider*. In simple service compositions, such dependency relations are hardcoded on consumer services. But decoupling the consumer service implementation from the actual provider endpoint is a good practice, which enables dynamic adaptation. Moreover, dependency hardcoding is not possible on cloud environments, since we do not know service addresses before deployment. Therefore, in the CHOReOS environment each consumer service is declared as depending on *roles* rather than other service implementations. The consumer service must receive the actual provider endpoint of a service fulfilling the required role through the `setInvocationAddress` operation, which every consumer service must implement.

The Enactment Engine will use `ServiceDependency` data to know which calls it must perform to the `setInvocationAddress` operation of participant services. Thus, the Enactment Engine will be able to tell, for example, to `ServiceA` that it must use `ServiceB` as `Role1`, where `ServiceB` is the list of endpoint URIs corresponding to the multiple instances of `ServiceB`. In this way, the CHOReOS middleware provides a *dependency injection*[3] mechanism to wire up service dependencies.

Obs: to SOAP services, the URI passed to the `setInvocationAddress` operation does not contain the '`?wsdl`' suffix.

[2]When the package type i s COMMAND_LINE the service will be executed by the "java -jar" command.

[3]Dependency Injection pattern, by Martin Fowler: `http://martinfowler.com/articles/injection.html`

Resource impact specification

The `DeployableServiceSpec` class has also an attribute to specify non-functional requirements. This attribute is called "resource impact", and it can be used by the `NodeSelector` to choose the node in which the service should be deployed. `NodeSelector` will try to choose a node that enables the service to fulfil such requirements.

This attribute is not described in this document because its structure is not fully defined yet. But it is expected to define, among others, required values of CPU, memory, and disk usage.

XML representation

Each class is mapped to and from an XML representation according to the default behaviour of the JAXB API[4]. To properly build and read these XML representations, you can rely on the schema definition (Section 2.5). We provide here an example of `ChorSpec` (Listing 2.1) and `Choreography` (Listing 2.2) XML representations to a little choreography with just two services (airline and travel-agency services).

Listing 2.1: ChorSpec XML representation example.

```
<choreographySpec>
    <deployableServiceSpecs>
        <name>airline</name>
        <roles>airline</roles>
        <serviceType>
            <type>SOAP</type>
        </serviceType>
        <endpointName>airline</endpointName>
        <numberOfInstances>1</numberOfInstances>
        <packageType>
            <type>COMMAND_LINE</type>
        </packageType>
        <packageUri>http://valinhos.ime.usp.br:54080/enact_test/v3/Airline-3.0.0.jar</
            packageUri>
        <port>1234</port>
        <resourceImpact/>
        <version>0.1</version>
    </deployableServiceSpecs>
    <deployableServiceSpecs>
        <dependencies>
            <serviceSpecName>airline</serviceSpecName>
            <serviceSpecRole>airline</serviceSpecRole>
        </dependencies>
        <name>travelagency</name>
        <roles>travelagency</roles>
        <serviceType>
            <type>SOAP</type>
        </serviceType>
        <endpointName>travelagency</endpointName>
        <numberOfInstances>1</numberOfInstances>
        <packageType>
            <type>COMMAND_LINE</type>
        </packageType>
        <packageUri>http://valinhos.ime.usp.br:54080/enact_test/v3/TravelAgency-3.0.0.jar
            </packageUri>
        <port>1235</port>
        <resourceImpact/>
        <version>0.1</version>
    </deployableServiceSpecs>
</choreographySpec>
```

Listing 2.2: Choreography XML representation example.

```
<choreography>
    <choreographySpec>
        <deployableServiceSpecs>
            <name>airline</name>
            <roles>airline</roles>
            <serviceType>
                <type>SOAP</type>
            </serviceType>
```

[4] Java Architecture for XML Binding (JAXB): allows Java developers to map Java classes to XML representations.

```
 9              <endpointName>airline</endpointName>
10              <numberOfInstances>1</numberOfInstances>
11              <packageType>
12                  <type>COMMAND_LINE</type>
13              </packageType>
14              <packageUri>http://valinhos.ime.usp.br:54080/enact_test/v3/Airline-3.0.0.jar<
                    /packageUri>
15              <port>1234</port>
16              <resourceImpact/>
17              <version>0.1</version>
18          </deployableServiceSpecs>
19          <deployableServiceSpecs>
20              <dependencies>
21                  <serviceSpecName>airline</serviceSpecName>
22                  <serviceSpecRole>airline</serviceSpecRole>
23              </dependencies>
24              <name>travelagency</name>
25              <roles>travelagency</roles>
26              <serviceType>
27                  <type>SOAP</type>
28              </serviceType>
29              <endpointName>travelagency</endpointName>
30              <numberOfInstances>1</numberOfInstances>
31              <packageType>
32                  <type>COMMAND_LINE</type>
33              </packageType>
34              <packageUri>http://valinhos.ime.usp.br:54080/enact_test/v3/TravelAgency
                    -3.0.0.jar</packageUri>
35              <port>1235</port>
36              <resourceImpact/>
37              <version>0.1</version>
38          </deployableServiceSpecs>
39      </choreographySpec>
40      <deployableServices>
41          <spec xsi:type="deployableServiceSpec" xmlns:xsi="http://www.w3.org/2001/
                XMLSchema-instance">
42              <dependencies>
43                  <serviceSpecName>airline</serviceSpecName>
44                  <serviceSpecRole>airline</serviceSpecRole>
45              </dependencies>
46              <name>travelagency</name>
47              <roles>travelagency</roles>
48              <serviceType>
49                  <type>SOAP</type>
50              </serviceType>
51              <endpointName>travelagency</endpointName>
52              <numberOfInstances>1</numberOfInstances>
53              <packageType>
54                  <type>COMMAND_LINE</type>
55              </packageType>
56              <packageUri>http://valinhos.ime.usp.br:54080/enact_test/v3/TravelAgency
                    -3.0.0.jar</packageUri>
57              <port>1235</port>
58              <resourceImpact/>
59              <version>0.1</version>
60          </spec>
61          <UUID>9fb5c93a-b4d1-4a56-9b4b-120e89681e31</UUID>
62          <selectedNodes>
63              <hostname>choreos-node</hostname>
64              <id>2</id>
65              <ip>192.168.56.102</ip>
66          </selectedNodes>
67          <serviceInstances>
68              <instanceId>travelagency1</instanceId>
69              <nativeUri>http://192.168.56.102:1235/travelagency/</nativeUri>
70              <node>
71                  <hostname>choreos-node</hostname>
72                  <id>2</id>
73                  <ip>192.168.56.102</ip>
74              </node>
75              <serviceSpec>
76                  <dependencies>
77                      <serviceSpecName>airline</serviceSpecName>
```

```
78                      <serviceSpecRole>airline</serviceSpecRole>
79                  </dependencies>
80                  <name>travelagency</name>
81                  <roles>travelagency</roles>
82                  <serviceType>
83                      <type>SOAP</type>
84                  </serviceType>
85                  <endpointName>travelagency</endpointName>
86                  <numberOfInstances>1</numberOfInstances>
87                  <packageType>
88                      <type>COMMAND_LINE</type>
89                  </packageType>
90                  <packageUri>http://valinhos.ime.usp.br:54080/enact_test/v3/TravelAgency
                        -3.0.0.jar</packageUri>
91                  <port>1235</port>
92                  <resourceImpact/>
93                  <version>0.1</version>
94              </serviceSpec>
95          </serviceInstances>
96      </deployableServices>
97      <deployableServices>
98          <spec xsi:type="deployableServiceSpec" xmlns:xsi="http://www.w3.org/2001/
                XMLSchema-instance">
99              <name>airline</name>
100             <roles>airline</roles>
101             <serviceType>
102                 <type>SOAP</type>
103             </serviceType>
104             <endpointName>airline</endpointName>
105             <numberOfInstances>1</numberOfInstances>
106             <packageType>
107                 <type>COMMAND_LINE</type>
108             </packageType>
109             <packageUri>http://valinhos.ime.usp.br:54080/enact_test/v3/Airline-3.0.0.jar<
                    /packageUri>
110             <port>1234</port>
111             <resourceImpact/>
112             <version>0.1</version>
113         </spec>
114         <UUID>68d8e82b-f6e6-4314-9415-d2a18f61edcf</UUID>
115         <selectedNodes>
116             <hostname>choreos-node</hostname>
117             <id>1</id>
118             <ip>192.168.56.101</ip>
119         </selectedNodes>
120         <serviceInstances>
121             <instanceId>airline0</instanceId>
122             <nativeUri>http://192.168.56.101:1234/airline/</nativeUri>
123             <node>
124                 <hostname>choreos-node</hostname>
125                 <id>1</id>
126                 <ip>192.168.56.101</ip>
127             </node>
128             <serviceSpec>
129                 <name>airline</name>
130                 <roles>airline</roles>
131                 <serviceType>
132                     <type>SOAP</type>
133                 </serviceType>
134                 <endpointName>airline</endpointName>
135                 <numberOfInstances>1</numberOfInstances>
136                 <packageType>
137                     <type>COMMAND_LINE</type>
138                 </packageType>
139                 <packageUri>http://valinhos.ime.usp.br:54080/enact_test/v3/Airline-3.0.0.
                        jar</packageUri>
140                 <port>1234</port>
141                 <resourceImpact/>
142                 <version>0.1</version>
143             </serviceSpec>
144         </serviceInstances>
145     </deployableServices>
146     <id>1</id>
```

```
147 </choreography>
```

2.3 REST API

The Enactment Engine clients access its features through the REST API that is described in this section.

Create a choreography

HTTP Method	*URI*	*Request body*	*Responses*

POST	/chors	`ChorSpec` XML representation (see Listing 2.1)	`201 CREATED`
			`location = "/chors/{id}"`
			
			`400 BAD REQUEST`
			
			`500 ERROR`

Creates a specification of the choreography on the Enactment Engine. It does not deploy the choreography.

Obs: `application/xml` is the value to the `Content Type` header when XML representations are written in the request or response body.

Retrieve choreography information

HTTP Method	*URI*	*Request body*	*Responses*

GET	/chors/{id}	-	`200 OK`
			`location = "/chors/{id}"`
			
			`Body:`
			
			`Choreography` XML representation (see Listing 2.2)
			
			`400 BAD REQUEST`
			
			`404 NOT FOUND`
			
			`500 ERROR`

If this operation is invoked after the creation and before the deployment of a choreography, the body response will be a `Choreography` representation without any deployed service.

Deploy a choreography

HTTP Method	*URI*	*Request body*	*Responses*

POST	/chors/{id}/deployment	-	`200 OK`
			`location = "/chors/{id}"`
			`Body:`
			
			`Choreography` XML representation (see Listing 2.2)
			
			`400 BAD REQUEST`
			
			`404 NOT FOUND`
			
			`500 ERROR`

With this invocation, services will be finally deployed. The response arrives only after the deployment of all services, if no deployment fails. It is possible to parse the output to find out failed deployments, which will be the services without associated nodes.

Update a choreography (only partially implemented at this time)

HTTP Method	*URI*	*Request body*	*Responses*

PUT	/chors/{id}	`ChorSpec` XML representation (see Listing 2.1)	`200 OK`
			`location = "/chors/{id}"`
			`Body:`
			
			`Choreography` XML representation (see Listing 2.2)
			
			`400 BAD REQUEST`
			
			`404 NOT FOUND`
			
			`500 ERROR`

This operation has the same behavior of the create choreography operation. To apply the changes it is necessary to invoke the deployment operation again. When the new deployment is invoked, the Enactment Engine will detect the changes that have been inserted in the choreography and deploys new services, remove unneeded services and redeploy services where some aspect (such as version number, number of instances etc) has changed. Currently, the only detected changes are increased or decreased number of instances and increased or decreased memory consumption. Services on the old and new versions of the choreography are correlated by means of the `name` attribute of ServiceSpec.

2.4 Java client

In the `EnactmentEngineAPI` project there is the `EEClient` class, which implements the `EnactmentEngine` interface (Listing 2.3) and handles the REST communication with the Enactment Engine server. This means you can invoke the Enactment Engine by using a simple Java object, without worrying with XML details.

Listing 2.3: Enactment Engine Java interface.

```
1  package org.ow2.choreos.chors;
2
3  import org.ow2.choreos.chors.datamodel.Choreography;
4  import org.ow2.choreos.chors.datamodel.ChoreographySpec;
5
6  public interface EnactmentEngine {
7
8      public String createChoreography(ChoreographySpec chor);
9
10     public Choreography getChoreography(String chorId)
11         throws ChoreographyNotFoundException;
12
13     public Choreography deployChoreography(String chorId)
14         throws DeploymentException, ChoreographyNotFoundException;
15
16     public void updateChoreography(String chorId, ChoreographySpec spec)
17         throws EnactmentException, ChoreographyNotFoundException;
18
19 }
```

To use the Enactment Engine Java client in your code, it's enough to import the API project into your project. One way of doing this is using Maven: install the API project into your local maven repo (`EnactmentEngineAPI$mvn install`), add it as a dependency of your project by editing your pom.xml (Listing 2.4), and finally compile your project.

Listing 2.4: Adding EnactmentEngineAPI as a dependency of your project.

```
<dependency>
  <groupId>org.ow2.choreos</groupId>
  <artifactId>EnactmentEngineAPI</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency>
```

The Listing 2.5 shows an example of how to use the Java API to create a choreography specification. This example is equivalent to the XML in Listing 2.1.

Listing 2.5: Example of choreography specification using the Java API.

```
public class ChorSpecExample {

    public static final String AIRLINE = "airline";
    public static final String TRAVEL_AGENCY = "travelagency";
    public static final String AIRLINE_JAR =
            "http://valinhos.ime.usp.br:54080/airline.jar";
    public static final String TRAVEL_AGENCY_JAR =
            "http://valinhos.ime.usp.br:54080/travel.jar";
    public static final int AIRLINE_PORT = 1234;
    public static final int TRAVEL_AGENCY_PORT = 1235;

    private ChoreographySpec chorSpec;
    private DeployableServiceSpec airlineSpec;
    private DeployableServiceSpec travelSpec;

    public ChoreographySpec getChorSpec() {
        createAirlineSpec();
        createTravelAgencySpec();
        chorSpec = new ChoreographySpec(this.airlineSpec, this.travelSpec);
        return chorSpec;
    }

    private void createAirlineSpec() {
        airlineSpec = new DeployableServiceSpec();
        airlineSpec.setName(AIRLINE);
        airlineSpec.setServiceType(ServiceType.SOAP);
        airlineSpec.setPackageType(PackageType.COMMAND_LINE);
        airlineSpec.setPackageUri(AIRLINE_JAR);
        airlineSpec.setPort(AIRLINE_PORT);
        airlineSpec.setEndpointName(AIRLINE);
        airlineSpec.setRoles(Collections.singletonList(AIRLINE));
    }

    private void createTravelAgencySpec() {
        travelSpec = new DeployableServiceSpec();
        travelSpec.setName(TRAVEL_AGENCY);
        travelSpec.setServiceType(ServiceType.SOAP);
        travelSpec.setPackageType(PackageType.COMMAND_LINE);
        travelSpec.setPackageUri(TRAVEL_AGENCY_JAR);
        travelSpec.setPort(TRAVEL_AGENCY_PORT);
        travelSpec.setEndpointName(TRAVEL_AGENCY);
        travelSpec.setRoles(Collections.singletonList(TRAVEL_AGENCY));
        ServiceDependency dependency = new ServiceDependency();
        dependency.setServiceSpecName(AIRLINE);
        dependency.setServiceSpecRole(AIRLINE);
        travelSpec.addDependency(dependency);
    }
```

Finally, Listing 2.6 is an example of how to use the Java API to invoke the EE.

Listing 2.6: Deploying a choreography using the Java API.

```
public class Deployment {

    public static void main(String[] args) throws DeploymentException,
        ChoreographyNotFoundException {

        final String EE_URI = "http://localhost:9102/enactmentengine";
        EnactmentEngine ee = new EnactmentEngineClient(EE_URI);
        ChorSpecExample example = new ChorSpecExample();
        ChoreographySpec chorSpec = example.getChorSpec();

```

```
10          String chorId = ee.createChoreography(chorSpec);
11          Choreography chor = ee.deployChoreography(chorId);
12
13          System.out.println(chor); // checking output
14      }
15  }
```

2.5 Choreography XML Schema Definition (XSD file)

```
1   ChorSpec XSD:
2   <?xml version="1.0" encoding="UTF-8"?><xs:schema xmlns:xs="http://www.w3.org/2001/
        XMLSchema" version="1.0">
3   <xs:element name="choreographySpec" type="choreographySpec"/>
4   <xs:element name="deployableServiceSpec" type="deployableServiceSpec"/>
5   <xs:element name="legacyServiceSpec" type="legacyServiceSpec"/>
6   <xs:element name="resourceImpact" type="resourceImpact"/>
7   <xs:complexType name="choreographySpec">
8           <xs:sequence>
9               <xs:element maxOccurs="unbounded" minOccurs="0" name="deployableServiceSpecs"
                     nillable="true" type="deployableServiceSpec"/>
10              <xs:element maxOccurs="unbounded" minOccurs="0" name="legacyServiceSpecs"
                    nillable="true" type="legacyServiceSpec"/>
11          </xs:sequence>
12      </xs:complexType>
13  <xs:complexType name="deployableServiceSpec">
14          <xs:complexContent>
15              <xs:extension base="serviceSpec">
16                  <xs:sequence>
17                      <xs:element minOccurs="0" name="cloudAccount" type="xs:string"/>
18                      <xs:element minOccurs="0" name="desiredQoS" type="desiredQoS"/>
19                      <xs:element minOccurs="0" name="endpointName" type="xs:string"/>
20                      <xs:element name="numberOfInstances" type="xs:int"/>
21                      <xs:element minOccurs="0" name="packageType" type="packageType"/>
22                      <xs:element minOccurs="0" name="packageUri" type="xs:string"/>
23                      <xs:element name="port" type="xs:int"/>
24                      <xs:element minOccurs="0" ref="resourceImpact"/>
25                      <xs:element minOccurs="0" name="version" type="xs:string"/>
26                  </xs:sequence>
27              </xs:extension>
28          </xs:complexContent>
29      </xs:complexType>
30  <xs:complexType abstract="true" name="serviceSpec">
31          <xs:sequence>
32              <xs:element maxOccurs="unbounded" minOccurs="0" name="dependencies" nillable=
                    "true" type="serviceDependency"/>
33              <xs:element minOccurs="0" name="name" type="xs:string"/>
34              <xs:element maxOccurs="unbounded" minOccurs="0" name="roles" nillable="true"
                    type="xs:string"/>
35              <xs:element minOccurs="0" name="serviceType" type="serviceType"/>
36          </xs:sequence>
37      </xs:complexType>
38  <xs:complexType name="desiredQoS">
39          <xs:sequence>
40              <xs:element minOccurs="0" name="responseTimeMetric" type="responseTimeMetric"
                    />
41          </xs:sequence>
42      </xs:complexType>
43  <xs:complexType name="responseTimeMetric">
44          <xs:sequence>
45              <xs:element name="acceptablePercentage" type="xs:float"/>
46              <xs:element name="maxDesiredResponseTime" type="xs:float"/>
47          </xs:sequence>
48      </xs:complexType>
49  <xs:complexType name="packageType">
50          <xs:sequence>
51              <xs:element minOccurs="0" name="type" type="xs:string"/>
52          </xs:sequence>
53      </xs:complexType>
54  <xs:complexType name="resourceImpact">
55          <xs:sequence>
56              <xs:element minOccurs="0" name="memory" type="memoryType"/>
```

```
57              <xs:element minOccurs="0" name="cpu" type="xs:string"/>
58              <xs:element minOccurs="0" name="storage" type="xs:string"/>
59              <xs:element minOccurs="0" name="network" type="xs:string"/>
60          </xs:sequence>
61      </xs:complexType>
62  <xs:complexType name="serviceDependency">
63          <xs:sequence>
64              <xs:element minOccurs="0" name="serviceSpecName" type="xs:string"/>
65              <xs:element minOccurs="0" name="serviceSpecRole" type="xs:string"/>
66          </xs:sequence>
67      </xs:complexType>
68  <xs:complexType name="serviceType">
69          <xs:sequence>
70              <xs:element minOccurs="0" name="type" type="xs:string"/>
71          </xs:sequence>
72      </xs:complexType>
73  <xs:complexType name="legacyServiceSpec">
74          <xs:complexContent>
75              <xs:extension base="serviceSpec">
76                  <xs:sequence>
77                      <xs:element maxOccurs="unbounded" minOccurs="0" name="nativeURIs"
                            nillable="true" type="xs:string"/>
78                  </xs:sequence>
79              </xs:extension>
80          </xs:complexContent>
81      </xs:complexType>
82  <xs:simpleType name="memoryType">
83          <xs:restriction base="xs:string">
84              <xs:enumeration value="SMALL"/>
85              <xs:enumeration value="MEDIUM"/>
86              <xs:enumeration value="LARGE"/>
87          </xs:restriction>
88      </xs:simpleType>
89  </xs:schema>
90
91  Choreography XSD:
92  <?xml version="1.0" encoding="UTF-8"?><xs:schema xmlns:xs="http://www.w3.org/2001/
        XMLSchema" version="1.0">
93  <xs:element name="choreography" type="choreography"/>
94  <xs:element name="choreographySpec" type="choreographySpec"/>
95  <xs:element name="cloudNode" type="cloudNode"/>
96  <xs:element name="deployableService" type="deployableService"/>
97  <xs:element name="deployableServiceSpec" type="deployableServiceSpec"/>
98  <xs:element name="legacyServiceSpec" type="legacyServiceSpec"/>
99  <xs:element name="resourceImpact" type="resourceImpact"/>
100 <xs:complexType name="choreography">
101         <xs:sequence>
102             <xs:element minOccurs="0" ref="choreographySpec"/>
103             <xs:element maxOccurs="unbounded" minOccurs="0" name="deployableServices"
                    nillable="true" type="deployableService"/>
104             <xs:element minOccurs="0" name="id" type="xs:string"/>
105             <xs:element maxOccurs="unbounded" minOccurs="0" name="legacyServices"
                    nillable="true" type="legacyService"/>
106         </xs:sequence>
107     </xs:complexType>
108 <xs:complexType name="choreographySpec">
109         <xs:sequence>
110             <xs:element maxOccurs="unbounded" minOccurs="0" name="deployableServiceSpecs"
                    nillable="true" type="deployableServiceSpec"/>
111             <xs:element maxOccurs="unbounded" minOccurs="0" name="legacyServiceSpecs"
                    nillable="true" type="legacyServiceSpec"/>
112         </xs:sequence>
113     </xs:complexType>
114 <xs:complexType name="deployableServiceSpec">
115         <xs:complexContent>
116             <xs:extension base="serviceSpec">
117                 <xs:sequence>
118                     <xs:element minOccurs="0" name="cloudAccount" type="xs:string"/>
119                     <xs:element minOccurs="0" name="desiredQoS" type="desiredQoS"/>
120                     <xs:element minOccurs="0" name="endpointName" type="xs:string"/>
121                     <xs:element name="numberOfInstances" type="xs:int"/>
122                     <xs:element minOccurs="0" name="packageType" type="packageType"/>
123                     <xs:element minOccurs="0" name="packageUri" type="xs:string"/>
```

```
124                     <xs:element name="port" type="xs:int"/>
125                     <xs:element minOccurs="0" ref="resourceImpact"/>
126                     <xs:element minOccurs="0" name="version" type="xs:string"/>
127                 </xs:sequence>
128             </xs:extension>
129         </xs:complexContent>
130     </xs:complexType>
131 <xs:complexType abstract="true" name="serviceSpec">
132         <xs:sequence>
133             <xs:element maxOccurs="unbounded" minOccurs="0" name="dependencies" nillable=
                    "true" type="serviceDependency"/>
134             <xs:element minOccurs="0" name="name" type="xs:string"/>
135             <xs:element maxOccurs="unbounded" minOccurs="0" name="roles" nillable="true"
                    type="xs:string"/>
136             <xs:element minOccurs="0" name="serviceType" type="serviceType"/>
137         </xs:sequence>
138     </xs:complexType>
139 <xs:complexType name="desiredQoS">
140         <xs:sequence>
141             <xs:element minOccurs="0" name="responseTimeMetric" type="responseTimeMetric"
                    />
142         </xs:sequence>
143     </xs:complexType>
144 <xs:complexType name="responseTimeMetric">
145         <xs:sequence>
146             <xs:element name="acceptablePercentage" type="xs:float"/>
147             <xs:element name="maxDesiredResponseTime" type="xs:float"/>
148         </xs:sequence>
149     </xs:complexType>
150 <xs:complexType name="packageType">
151         <xs:sequence>
152             <xs:element minOccurs="0" name="type" type="xs:string"/>
153         </xs:sequence>
154     </xs:complexType>
155 <xs:complexType name="resourceImpact">
156         <xs:sequence>
157             <xs:element minOccurs="0" name="memory" type="memoryType"/>
158             <xs:element minOccurs="0" name="cpu" type="xs:string"/>
159             <xs:element minOccurs="0" name="storage" type="xs:string"/>
160             <xs:element minOccurs="0" name="network" type="xs:string"/>
161         </xs:sequence>
162     </xs:complexType>
163 <xs:complexType name="serviceDependency">
164         <xs:sequence>
165             <xs:element minOccurs="0" name="serviceSpecName" type="xs:string"/>
166             <xs:element minOccurs="0" name="serviceSpecRole" type="xs:string"/>
167         </xs:sequence>
168     </xs:complexType>
169 <xs:complexType name="serviceType">
170         <xs:sequence>
171             <xs:element minOccurs="0" name="type" type="xs:string"/>
172         </xs:sequence>
173     </xs:complexType>
174 <xs:complexType name="legacyServiceSpec">
175         <xs:complexContent>
176             <xs:extension base="serviceSpec">
177                 <xs:sequence>
178                     <xs:element maxOccurs="unbounded" minOccurs="0" name="nativeURIs"
                            nillable="true" type="xs:string"/>
179                 </xs:sequence>
180             </xs:extension>
181         </xs:complexContent>
182     </xs:complexType>
183 <xs:complexType name="deployableService">
184         <xs:complexContent>
185             <xs:extension base="service">
186                 <xs:sequence>
187                     <xs:element maxOccurs="unbounded" minOccurs="0" name="selectedNodes"
                            nillable="true" type="cloudNode"/>
188                     <xs:element maxOccurs="unbounded" minOccurs="0" name="
                            serviceInstances" nillable="true" type="serviceInstance"/>
189                 </xs:sequence>
190             </xs:extension>
```

```
191         </xs:complexContent>
192     </xs:complexType>
193 <xs:complexType abstract="true" name="service">
194         <xs:sequence>
195             <xs:element minOccurs="0" name="spec" type="serviceSpec"/>
196             <xs:element minOccurs="0" name="UUID" type="xs:string"/>
197         </xs:sequence>
198     </xs:complexType>
199 <xs:complexType name="cloudNode">
200         <xs:sequence>
201             <xs:element minOccurs="0" name="cpus" type="xs:int"/>
202             <xs:element minOccurs="0" name="hostname" type="xs:string"/>
203             <xs:element minOccurs="0" name="id" type="xs:string"/>
204             <xs:element minOccurs="0" name="image" type="xs:string"/>
205             <xs:element minOccurs="0" name="ip" type="xs:string"/>
206             <xs:element minOccurs="0" name="privateKeyFile" type="xs:string"/>
207             <xs:element minOccurs="0" name="ram" type="xs:int"/>
208             <xs:element minOccurs="0" name="so" type="xs:string"/>
209             <xs:element minOccurs="0" name="state" type="xs:int"/>
210             <xs:element minOccurs="0" name="storage" type="xs:int"/>
211             <xs:element minOccurs="0" name="user" type="xs:string"/>
212             <xs:element minOccurs="0" name="zone" type="xs:string"/>
213         </xs:sequence>
214     </xs:complexType>
215 <xs:complexType name="serviceInstance">
216         <xs:sequence>
217             <xs:element minOccurs="0" name="instanceId" type="xs:string"/>
218             <xs:element minOccurs="0" name="nativeUri" type="xs:string"/>
219             <xs:element minOccurs="0" name="node" type="cloudNode"/>
220             <xs:element minOccurs="0" name="serviceSpec" type="deployableServiceSpec"/>
221         </xs:sequence>
222     </xs:complexType>
223 <xs:complexType name="legacyService">
224         <xs:complexContent>
225             <xs:extension base="service">
226                 <xs:sequence>
227                     <xs:element maxOccurs="unbounded" minOccurs="0" name="
                            legacyServiceInstances" nillable="true" type="
                            legacyServiceInstance"/>
228                 </xs:sequence>
229             </xs:extension>
230         </xs:complexContent>
231     </xs:complexType>
232 <xs:complexType name="legacyServiceInstance">
233         <xs:sequence>
234             <xs:element minOccurs="0" name="spec" type="legacyServiceSpec"/>
235             <xs:element minOccurs="0" name="uri" type="xs:string"/>
236         </xs:sequence>
237     </xs:complexType>
238 <xs:simpleType name="memoryType">
239         <xs:restriction base="xs:string">
240             <xs:enumeration value="SMALL"/>
241             <xs:enumeration value="MEDIUM"/>
242             <xs:enumeration value="LARGE"/>
243         </xs:restriction>
244     </xs:simpleType>
245 </xs:schema>
```

Chapter 3

How to package services to be deployed by the Enactment Engine

3.1 Introduction

There are two main attributes in the DeployableServiceSpec class that define constraints on how a service must be coded and packaged. The `packageType` attribute defines which kind of deployable package is expected by the Enactment Engine and, therefore, the process of deploying and running the specified service. Examples of package types are COMMAND_LINE and TOMCAT. The `serviceType` attribute defines the process of invoking the specified service, which is used by the Enactment Engine to properly invoke the `setInvocationAddress` operation during deployment. Examples of service types are SOAP and REST types, although only SOAP services are currently supported by the Enactment Engine.

This chapter is targeted to service *developers* that intend to develop EE compatible services. Descriptions and hints encompass coding and packaging phases. This guide considers only the service and package types currently supported by EE.

3.2 COMMAND_LINE package type

Services whose package type are specified as COMMAND_LINE must be provided as JAR packages. The JAR must contain all the dependencies and resources within it. When using this package type, it is mandatory to fill the attributes `port` and `endpointName` on DeployableServiceSpec.

It must be possible to run the JAR by typing the command `java -jar <file_name>`, where `<file_name>` must be replaced with the name of the JAR file. Every JAR file contains a file called `MANIFEST.MF` within the `META-INF` folder, which is in the root of JAR file. A runnable JAR contains the following entry in its `MANIFEST` file: "`Main-Class: <class>`", where `<class>` must be replaces by the full qualified name of the class containing the main method, as for example `org.ow2.choreos.AirlineStarter`.

The Listing 3.1 provides an example of main class within a JAR file. The Airline interface is the business interface, and the AirlineService is the implementing class that uses the JAX-WS framework to expose SOAP services. In this example, the used TCP port and the endpoint name are defined in the SERVICE_ADDRESS assignment (line 7). The used port is the 1234, and the endpoint name is "airline". *Attention!* The use of "0.0.0.0" instead of "localhost" is necessary to make the service accessible from outside the machine where it is running.

Listing 3.1: Example of a class with the main method within a JAR file.

```
1  package org.ow2.choreos;
2
3  import javax.xml.ws.Endpoint;
4
5  public class AirlineStarter {
6
7    public static final String SERVICE_ADDRESS = ``http://0.0.0.0:1234/airline'';
8    private static Endpoint endpoint;
9
10   public static void start() {
11     Airline service = new AirlineService();
12     endpoint = Endpoint.create(service);
```

```
13      endpoint.publish(SERVICE_ADDRESS);
14    }
15
16    public static void main(String[] args) {
17      start();
18    }
19  }
```

Runnable JARs can be easily generated by the export menu in Eclipse or by Maven. To generate a runnable JAR using Maven 3, add the excerpt in the Listing 3.2 in your `pom.xml` file, properly replacing the content of the `mainClass` element. In this way, when generating the JAR file (`mvn package`), Maven will be in charge of properly generating the `MANIFEST` file.

Listing 3.2: Excerpt of pom file to generate a runnable JAR using Maven 3.

```
1  <build>
2    <finalName>airline-service</finalName>
3    <!-- <finalName>travel-agency-service</finalName> -->
4    <plugins>
5      <plugin>
6        <groupId>org.apache.maven.plugins</groupId>
7        <artifactId>maven-shade-plugin</artifactId>
8        <version>2.0</version>
9        <executions>
10         <execution>
11           <phase>package</phase>
12           <goals>
13             <goal>shade</goal>
14           </goals>
15           <configuration>
16             <shadedArtifactId>airline-service</shadedArtifactId>
17             <shadeSourcesContent>true</shadeSourcesContent>
18             <transformers>
19                 <transformer
20 implementation=''org.apache.maven.plugins.shade.resource.ManifestResourceTransformer''>
21                 <mainClass>org.ow2.choreos.AirlineStarter</mainClass>
22               </transformer>
23             </transformers>
24           </configuration>
25         </execution>
26       </executions>
27     </plugin>
28   </plugins>
29 </build>
```

An important issue about using JAR packages is that the Enactment Engine is not able to prevent TCP port conflicts. Therefore, try to avoid the use of the same port in multiple services. Also, be sure that the required ports are not blocked by the cloud infrastructure. However, a better way to prevent such port issues is not to use JAR packages, but use WAR packages instead.

3.3 TOMCAT package type

Services whose package type are specified as TOMCAT must be provided as WAR packages. The Enactment Engine will be in charge of deploying and starting a Tomcat instance, or select an already running instance, and then deploying the WAR package onto the selected instance. If the service port is not specified in the service specification, the middleware assumes the port as 8080, the Tomcat default port. If the endpoint name is not specified, it is assumed as the WAR file name, without the "war" extension, as it is the default behavior in Tomcat.

Dependencies (JAR libraries) must be packaged within the WAR file. However, we provide a set of libraries in our Tomcat installation that are usually used in Java projects, specially projects using the JAX-WS framework. If some of these JARs are used by your service, they are not required to be packaged within your WAR file, since they are already on the Tomcat class path. This strategy helps in decreasing the size of WAR files and, therefore, decreasing the deployment time. The provided libraries are the following:

- `activation-1.1.jar`
- `ecj-3.7.1.jar`

- gmbal-api-only-3.1.0-b001.jar
- ha-api-3.1.8.jar
- istack-commons-runtime-2.2.1.jar
- javax.annotation-3.1.1-b06.jar
- jaxb-api-2.2.3.jar
- jaxb-impl-2.2.4-1.jar
- jaxws-api-2.2.5.jar
- jaxws-rt-2.2.5.jar
- jsr181-api-1.0-MR1.jar
- management-api-3.0.0-b012.jar
- mimepull-1.6.jar
- policy-2.2.2.jar
- resolver-20050927.jar
- saaj-api-1.3.3.jar
- saaj-impl-1.3.10.jar
- stax-api-1.0-2.jar
- stax-api-1.0.1.jar
- stax-ex-1.4.jar
- stax2-api-3.1.1.jar
- streambuffer-1.2.jar
- tomcat-api.jar
- tomcat-jdbc.jar
- tomcat-util.jar
- tomcat_libs.tar.gz txw2-20090102.jar
- woodstox-core-asl-4.1.1.jar
- wstx-asl-3.2.3.jar

One way to be sure that you are using the required versions is making your project depending on JAX-WS by adding the fragment of Listing 3.3 in your Maven's pom:

Listing 3.3: Making your project depending on JAX-WS using Maven.

```
1  <dependency>
2      <groupId>com.sun.xml.ws</groupId>
3      <artifactId>jaxws-rt</artifactId>
4      <version>2.1.4</version>
5  </dependency>
```

If you write your web service using JAX-WS, your WAR file must also package a `sun-jaxws.xml` file, as Listing 3.4. As any other WAR file, it must also contain a `web.xml` file. If your service was built with JAX-WS, your `web.xml` file must be similar to the one presented in the Listing 3.5. Be aware that besides the usual definition of `servlet` and `servelet-mapping` elements, it is also necessary to declare the `listener` element exactly as in the example (lines 8, 9, and 10)[1].

[1]The instructions about the `sun-jaxws.xml` and `web.xml` files were retrieved from `http://www.mkyong.com/webservices/jax-ws/deploy-jax-ws-web-services-on-tomcat/`

Listing 3.4: Example of `sun-jaxws.xml` file.

```
<?xml version=''1.0'' encoding=''UTF-8''?>
<endpoints
  xmlns=''http://java.sun.com/xml/ns/jax-ws/ri/runtime''
  version=''2.0''>
  <endpoint
      name=''Airline''
      implementation=''org.ow2.choreos.AirlineService''
      url-pattern=''/airline''/>
</endpoints>
```

Listing 3.5: Example of `web.xml` file.

```
<?xml version=''1.0'' encoding=''UTF-8''?>
<!DOCTYPE web-app PUBLIC ''-//Sun Microsystems,
Inc.//DTD Web Application 2.3//EN''
''http://java.sun.com/dtd/web-app_2_3.dtd''>

<web-app>
    <listener>
        <listener-class>
                com.sun.xml.ws.transport.http.servlet.WSServletContextListener
        </listener-class>
    </listener>
    <servlet>
        <servlet-name>airline</servlet-name>
        <servlet-class>
                com.sun.xml.ws.transport.http.servlet.WSServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>airline</servlet-name>
        <url-pattern>/airline</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>120</session-timeout>
    </session-config>
</web-app>
```

3.4 EASY_ESB package type

The Enactment Engine is also responsible for the coordination delegates deployment, that are executed by the EasyESB service bus. To enable this functionality, we have created the EASY_ESB package type, that is a tar.gz package containing a `config.xml` file with instructions for the bus. In the package, some resources needed for the deployment can be added. This process enables not only the deployment of coordination delegates, but actually any interaction with EasyESB.

The `config.xml` file must be built according to the schema in the Listing 3.6. `Configuration` is the root element. `Service` is an element containing a set of `Actions` made on a particular EasyESB node. These actions can be:

Deploy: deploys an artifact in EasyESB (BPEL, CD, etc.). It must contain the `MainResource` element and can have additional `Resource` elements.

Bind: binds a running web service onto an EasyESB; this action receives as parameter the web service URL and the web service WSDL location.

Proxify: binds a running web service onto an EasyESB node and re-export it using the same parameters used in the `Bind` action.

Expose: exposes an EasyESB internal service as a web service. Parameters are `ServiceNamespace` and `ServiceName`, that correspond to the `QName` of the service defined in the WSDL (usually it is the WSDL target namespace plus the name attribute of the service element).

Listing 3.6: `config.xml` schema.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <schema targetNamespace="http://ebmwebsourcing.com/cli/schema"
3   elementFormDefault="qualified" xmlns="http://www.w3.org/2001/XMLSchema"
4   xmlns:tns="http://ebmwebsourcing.com/cli/schema">
5
6    <element name="Configuration" type="tns:ConfigurationType" />
7
8    <complexType name="ConfigurationType">
9      <sequence>
10       <element maxOccurs="unbounded" name="Service"
11         type="tns:ServiceConfigurationType" />
12     </sequence>
13   </complexType>
14
15   <complexType name="ServiceConfigurationType">
16     <sequence>
17       <element maxOccurs="unbounded" name="Action" type="tns:ActionType" />
18     </sequence>
19     <attribute name="url" type="string" />
20   </complexType>
21
22   <complexType name="ActionType">
23     <choice>
24       <element name="Deploy" type="tns:DeployType" />
25       <element name="Bind" type="tns:BindType" />
26       <element name="Expose" type="tns:ExposeType" />
27       <element name="Proxify" type="tns:ProxifyType" />
28       <element name="AddNeighbourNode" type="tns:AddNeighbourNodeType" />
29     </choice>
30   </complexType>
31
32   <complexType name="AddNeighbourNodeType">
33     <sequence>
34       <element name="NeighbourAdminAddress" type="string" />
35     </sequence>
36   </complexType>
37
38   <complexType name="DeployType">
39     <sequence>
40       <element name="MainResource" type="string" />
41       <element maxOccurs="unbounded" name="Resource" type="string" />
42     </sequence>
43   </complexType>
44
45   <complexType name="BindType">
46     <sequence>
47       <element name="Url" type="string" />
48       <element name="Wsdl" type="string" />
49     </sequence>
50   </complexType>
51
52   <complexType name="ExposeType">
53     <sequence>
54       <element name="ServiceNamespace" type="string" />
55       <element name="ServiceName" type="string" />
56       <element name="EndpointName" type="string" />
57     </sequence>
58   </complexType>
59
60   <complexType name="ProxifyType">
61     <sequence>
62       <element name="Url" type="string" />
63       <element name="Wsdl" type="string" />
64     </sequence>
65   </complexType>
66
67 </schema>
```

The Listing 3.7 shows an example of `config.xml` file that makes the deployment of a coordination delegate in a scenario where the correspondent business service is already running and available[2]. The

[2]Indeed, the main scenario envisioned by CHOReOS is that there are already a lot of the services running "on the wild",

`weatherforecastservice.lts` and `cdweatherforecastservice.wsdl` files, referenced in lines 8 and 9, are provided within the tar.gz package. The use of the "../../" in these lines is mandatory. The url `http://192.168.56.101:8192/weatherforecastservice` provided in line 15, as well as the correspondent WSDL indicated in line 15, are references to a service already running and accessible. The `ServiceNamespace` (line 20) is the `targetNamespace` defined in the service WSDL. The value of the `ServiceName` element (line 21) must correspond to the value of the `name` attribute of the `service` element in the service WSDL. The value of the `EndpointName` element (line 22) must correspond to the `name` of the `portType` element in the coordination delegate WSDL. The lts file pointed by the `config.xml` is provided in the Listing 3.8, and the value of its `endpoint` attribute must correspond to the `name` of the `portType` element in the already-running service WSDL.

Listing 3.7: Example of `config.xml` that deploys a coordination delegate.

```
1  <?xml version=''1.0'' enc\begin{lstlisting}oding=''UTF-8''?>
2  <Configuration xmlns=''http://ebmwebsourcing.com/cli/schema''
3                 xmlns:xsi=''http://www.w3.org/2001/XMLSchema-instance''
4                 xsi:schemaLocation=''http://ebmwebsourcing.com/cli/schema conf-schema.xsd''>
5    <Service url="http://localhost:8180/services/adminExternalEndpoint">
6      <Action>
7        <Deploy>
8          <MainResource>../../weatherforecastservice.lts</MainResource>
9          <Resource>../../cdweatherforecastservice.wsdl</Resource>
10       </Deploy>
11     </Action>
12     <Action>
13       <Bind>
14         <Url>http://192.168.56.101:8192/weatherforecastservice</Url>
15         <Wsdl>http://192.168.56.101:8192/weatherforecastservice?wsdl</Wsdl>
16       </Bind>
17     </Action>
18     <Action>
19       <Expose>
20         <ServiceNamespace>http://services.choreos.org/</ServiceNamespace>
21         <ServiceName>WeatherForecastServiceService</ServiceName>
22         <EndpointName>CDWeatherForecastServicePort</EndpointName>
23       </Expose>
24     </Action>
25   </Service>
26 </Configuration>
```

Listing 3.8: LTS file of a simple coordination delegate that acts as a proxy.

```
1  endpoint=WeatherForecastServicePort
```

3.5 Choreographing services that are already running

A service choreography can be composed of services that are running before the choreography deployment. Although a service like this do not need to be deployed by the middleware, it must be declared on the choreography specification with the LegacyServiceSpec class. The `URIs` attribute must contain the list of URIs where the multiple replicas of the service are accessible, which will be used by the middleware to invoke the `setInvocationAddress` operation of other services.

3.6 SOAP service type

By convention, services of this type must provide an operation named "`setInvocationAddress`"; this operation is used to inform the service about the remote service endpoints that implement the various *roles* it depends on. The `setInvocationAddress` arguments are the following:

dependency role: defines the operations provided by the dependency. A service may depend on multiple services with different roles, so this argument is necessary to the service know how to use the received dependency. It is a requirement that the service must to know the available operations of each role from which it depends. The role of each service must be also defined in the choreography specification, that is the Enactment Engine input.

and choreographies are made just to compose these already-running services, situation in which only coordels are actually deployed.

dependency name: the name of the dependency that implements the role above. It works as a label that the service may use to distinguish different available services playing the same role. These different services are actually different implementations, possibly belonging to different organizations.

dependency endpoints: the list of alternative URIs to access the dependency. It has several URIs because a service may has multiple instances to improve its scalability. It is expected, but not required, from the dependent service to implement some load balancing between the different URIs. However, the dependent service may simply pick up anyone of the received endpoints. URIs passed to the `setInvocationAddress` operation do not contain the '`?wsdl`' suffix.

The expected interface of the `setInvocationAddress` operation is formally expressed by the WSDL elements presented in the Listing 3.9.

Listing 3.9: Parts of the service WSDL that define the `setInvocationAddress` operation.

```
<xs:schema version=''1.0'' targetNamespace=''http://services.choreos.org/''>
  ...
  <xs:complexType name=''setInvocationAddress''>
    <xs:sequence>
      <xs:element name=''arg0'' type=''xs:string'' minOccurs=''0''/>
      <xs:element name=''arg1'' type=''xs:string'' minOccurs=''0''/>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="arg2" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name=''setInvocationAddressResponse''>
    <xs:sequence/>
  </xs:complexType>
  ...
</xs:schema>

<message name=''setInvocationAddress''>
  <part name=''parameters'' element=''tns:setInvocationAddress''/>
</message>
<message name=''setInvocationAddressResponse''>
  <part name=''parameters'' element=''tns:setInvocationAddressResponse''/>
</message>

<portType ... >
...
  <operation name=''setInvocationAddress''>
    <input message=''tns:setInvocationAddress''/>
    <output message=''tns:setInvocationAddressResponse''/>
  </operation>
</portType>
```

If you are using the JAX-WS framework, you can easily create a compatible `setInvocationAddress` operation by using the code provided in the Listing 3.10.

Listing 3.10: Implementing `setInvocationAddress` with JAX-WS.

```
1  @WebService
2  public class SomeWebServiceClass {
3
4      ...
5
6      @WebMethod
7      public void setInvocationAddress(String role, String name, List<String> endpoints) {
8                  ...
9      }
10 }
```

3.7 COORDEL service type

This service type must be used when declaring service specifications to CHOReOS Coordination Delegates (CDs). Although a coordination delegate proxifies all the operations of a SOAP service, the proxified `setInvocationAddress` operation will be not invoked. If a coordination delegate cdA is declared to depend on a coordination delegate cdB, the Enactment Engine will link the EasyESB nodes hosting the coordination delegates by invoking the `addNeighbour` operation of the EasyESB hosting cdA passing as neighbor the EasyESB node hosting cdB. Such operation enables coordination delegates to communicate directly among them by using the primitives (UPDATE_STATE(), WAIT(), NOTIFY()) provided by the CD-component running on EasyESB nodes. Every CD depend on some business service (SOAP service). But this dependency must be not declared on the service specification. It is a implicit dependency that must be declared on the config.xml file of the correspondent CD. Finally, when some SOAP service is declared to depend on some CD, the SOAP service will receive through `setInvocationAddress` the CD endpoint exposed by the bus. Such CD endpoint proxifies the service related to the CD.

3.8 Coding guidelines

Here are just some few important reminders:

- Do not forget to unblock the TCP ports used by your services. Often, this may be accomplished by the usage of management tools of your cloud environment.
- WAR packages are preferable to JAR packages. WAR packages avoid port conflict issues, and it is easier to manage the life-cycle of services distributed in WAR packages thanks to the management utilities of Tomcat. Life-cycle management matters, for example, when debugging if a service is actually running or not. Handling the life-cycle of JAR packaged services requires directly handling Unix processes.
- Never use absolute paths to retrieve resources, since the service will not run in the same machine where it was compiled. A good way to access resources is using the `getResourceAsStream` method of the current class loader[3].
- When starting a JAR packaged service, do not use the "localhost" address to create the endpoint, since the service will be not remotely accessible. Instead, use the address "0.0.0.0" that will make your service listen in every possible address in the machine, including the localhost. This practice makes the service accessible from other machines.
- Do not use `System.out.println`, use a log tool instead. Since the services are deployed in an automated way, it might be impossible to retrieve the console output, which will make debugging harder. Using a logger, as Log4j for example, makes the service record its messages in a file, which helps developers and operators in debugging.
- Do not forget to validate the WSDL files of your web services, specially if there is some manual edition applied on them.

[3] `http://docs.oracle.com/javase/6/docs/api/java/lang/ClassLoader.html#getResourceAsStream(java.lang.String)`

- Ensure that the service port address in the WSDL file (see Listing 3.11), when seen from remote locations, do not use "localhost", "0.0.0.0", or other unsuitable addresses, as lan private IPs for example. Remember the client that sees the WSDL needs an accessible endpoint.

Listing 3.11: Good example of service port address on a WSDL file.

```
1 <service name="AirlineServiceService">
2   <port name="AirlineServicePort" binding="tns:AirlineServicePortBinding">
3     <soap:address location="http://200.221.3.47:1234/airline"/>
4   </port>
5 </service>
```

- If service A needs to invoke service B, there is no problem if service A is compiled with classes used to build service B. Actually, usually it is very useful to A having access to B interfaces. Nonetheless, this class dependency must be only static. There is no point in service A trying to access objects states of service B, or access resources, as configuration files, bundled in service B package.

- The `packageUri` attribute defines the URL from where the Enactment Engine retrieves the package to be deployed. Therefore, all the services packages need to be already Internet accessible at deployment time. This can be accomplished, for example, by hosting the packages on a web server.

- Packages cannot be downloaded from https URLs. This restricts using some services, such as Dropbox, to host the packages.

Chapter 4

Extending Enactment Engine

4.1 Introduction

Current PaaS solutions available on market are well known for their low flexibility. Some of them work only on a specific cloud environment, others only with a few development frameworks. Enactment Engine tries to overcome this issue by providing an extensible architecture. Although the out-of-box version of Enactment Engine is quite limited, with some programming is possible to extend it to provide support to new *i)* cloud providers, *ii)* package types, *iii)* service types, and *iv)* node selection policies. By "extending" we mean no current Enactment Engine code need to be changed, and that each new extension can be implemented by the means of a well-defined process, which are now described in this section.

4.2 Supporting new cloud providers

In Enactment Engine, cloud providers are just a source of virtual machines provisioning. Any technology able to create new virtual machines may be used as "cloud provider".

To implement a new cloud provider, it is necessary to implement the CloudProvider interface (Listing 4.1). Current implementations are AWSCloudProvider (that uses EC2 service), OpenStackKeystoneCloudProvider, and FixedCloudProvider (that always points to the same user-defined VMs). An example of new cloud provider implementation could be the VirtualBoxCloudProvider, that would use VirtualBox on the developer machine to create new VMs (this is an example more suited to test environments).

Listing 4.1: CloudProvider interface.

```
1  public interface CloudProvider {
2
3      public String getCloudProviderName();
4
5      public CloudNode createNode(NodeSpec nodeSpec)
6          throws NodeNotCreatedException;
7
8      public CloudNode getNode(String nodeId)
9          throws NodeNotFoundException;
10
11     public List<CloudNode> getNodes();
12
13     public void destroyNode(String id)
14         throws NodeNotDestroyed, NodeNotFoundException;
15
16     public CloudNode createOrUseExistingNode(NodeSpec nodeSpec)
17         throws NodeNotCreatedException;
18
19     public void setCloudConfiguration(CloudConfiguration cloudConfiguration);
20
21 }
```

Implementations should use the `cloudConfiguration` object to retrieve configuration properties supplied by EE administrators. Such properties usually encompass user credentials to access the infrastructure provider service, and options such as VM types or images. The `cloudConfiguration` object is injected into the cloud provider instance by the EE.

Important note: in the current implementation, Enactment Engine is tailored to work with nodes running *Ubuntu 12.04*. Therefore, CloudProvider implementors should provide Ubuntu 12.04 nodes.

The next step is to edit the `extensible/cloud_providers.properties` file, located on EnactmentEngine resources folder. You must add a line in the format `NAME=full.qualified.class.name`, where the key is just an alias that you can freely define (since it does not conflict with other existing keys on the same file), and the value is the full qualified name of the CloudProvider implementing class. It is also necessary to recompile the EnactmentEngine project in such way it can access the implementing class. One suggestion is by adding your class in your local maven repository and edit the EnactmentEngine project's pom to make EnactmentEngine dependent on your project holding the new cloud provider

Finally, to use your new cloud provider, it is necessary to configure the `clouds.properties`, adding a cloud account whose `CLOUD_PROVIDER` property values the `NAME` defined in the `cloud_providers.properties` file.

4.3 Supporting new package types

Services may be delivered in different package types, such as JARs, WARs, etc. Each package type has its own specific deployment procedures, as well its specific process to start the service. When using different technologies, such as Python, to write new services, you will need to define a new package type, as well the deployment procedure associated with it. Such procedure is specified in Chef recipe.

So, the first step is to create a new Chef cookbook similar to the "jar" and "war" recipes already provided by Enactment Engine. These cookbooks are actually templates that EE will use to create specific cookbooks to each service to be deployed. You can use the the constants $PACKAGE_URL and $NAME within your cookbook recipe and attributes files. These constants will be injected by Enactment Engine to each specific recipe. You can have an idea about how to use them by looking to the WAR cookbook implementation, in Listing 4.2 and Listing 4.3. After writing the new recipe, you must associate this recipe to the new package type by editing the `extending/cookbooks.properties` file.

Listing 4.2: Recipe template for WAR deployment.

```
1  include_recipe "apt"
2  include_recipe "tomcat::choreos"
3
4  remote_file "war_file" do
5          source "#{node['CHOReOSData']['serviceData']['$NAME']['PackageURL']}"
6          path "#{node['tomcat']['webapp_dir']}/$NAME.war"
7          mode "0755"
8          action :create_if_missing
9  end
10
11 file "#{node['tomcat']['webapp_dir']}/$NAME.war" do
12         action :nothing
13 end
```

Listing 4.3: Attributes template for WAR deployment.

```
1  default['CHOReOSData']['serviceData']['$NAME']['PackageURL'] = "$PACKAGE_URL"
```

It is up to EE to "guess" the service URI too. A service URI follows the format `http://IP:PORT/CONTEXT`. And the CONTEXT formation rule is package type dependent. Therefore, when extending package type, it is necessary to create a new URIContextRetriever implementation and to link this implementation to its package type in the URIContextRetrieverFactory class. To make this relationship, it is enough to add a single line in the factory, by adding a new entry in the `classMap` variable. Both URIContextRetriever and URIContextRetrieverFactory classes are in the org.ow2.choreos.services.datamodel.uri package, on the EnactmentEngineAPI project.

Hint: if your package type is based on some kind of container to run the services, such as Tomcat, it may be a good idea to prepare a new image with this container already installed and running. So, you can configure EE to create VMs with an already running instance of your chosen container (e.g. JBoss). This strategy helps in achieving a faster deployment.

4.4 Supporting new service types

Although web services came to tackle the interoperability issue, today we have a couple of technologies implementing the concept of services. The main standards in this context are SOAP and REST, but other technologies could be used to implement services, such as JMS.

In the Enactment Engine context, the service type affects only how the `setInvocationAddress` is invoked. Therefore, to support a new service type, you have only to write a new ContextSender (Listing 4.4) implementation.

Listing 4.4: ContextSender interface.

```
1 public interface ContextSender {
2
3     public void sendContext(String serviceEndpoint,
4                             String partnerRole,
5                             String partnerName,
6                       List<String> partnerEndpoints) throws ContextNotSentException;
7 }
```

The final step is to edit the `extensible/context_sender.properties` file, located on EnactmentEngine resources folder. You must add a line in the format `SERVICE_TYPE=full.qualified.class.name`, where the key is the name of the new service type, and the value is the full qualified name of the ContextSender implementing class. It is also necessary to recompile the EnactmentEngine project in such way it can access the implementing class. Now you can create new service specs using the just-created service type! But be sure services implementation are prepared to receive the `setInvocationAddress` invocation.

4.5 Supporting new node selection policies

A node selection policy defines the *mapping* of services to cloud nodes. Since cloud nodes are dynamically created, node selection policies must be flexible, and not rely on hard-coded IPs. A node selection policy may define nodes to be used based on services non-functional properties. To create a new node selector, you must create a new NodeSelector (Listing 4.5) implementation. Pay attention that such implementation must be thread-safe, since multiple threads will invoke concurrently the method `select`.

Listing 4.5: NodeSelector interface.

```
1 public interface NodeSelector extends Selector<CloudNode, DeployableServiceSpec> {
2
3 }
4
5 public interface Selector<T, R> {
6
7     public List<T> select(R requirements, int objectsQuantity) throws NotSelectedException;
8
9 }
```

After writing the new node selector, you must associate this selector to a label by editing the `extensible/node_selector.properties` file, at EnactmentEngine resources folder. To use the new selector, finally, you must attribute the defined label to the NODE_SELECTOR property on the `ee.properties` file.

Chapter 5

Elasticity and QoS management

As mentioned in Chapter 2, the Enactment Engine is able to modify a choreography that is currently running. For instance, one may decide to switch from using a service offered by one provider to a compatible service by a different provider, or to increase/decrease the number of deployed replicas of a given service in order to adapt to fluctuations in usage load. To do this, the user simply uses the API again to submit an updated version of the choreography specification to the Enactment Engine and requests the redeployment of the choreography. The Enactment Engine, in turn, detects the modifications made to the choreography and performs the requested modifications, by deploying new versions of services, removing service replicas etc.

This capability, together with the flexibility offered by the CHOReOS monitoring subsystem, presents the user with the framework necessary to adjust the run-time environment of the choreography according to QoS parameters and constraints, such as response time or cost. In order to accomplish this, the user needs to create a separate daemon that acts both as a monitoring client and as a client for the Enactment Engine. As a client for the monitoring system, this daemon uploads rules to the Glimpse Monitoring CEP and awaits for notifications from it whenever such rules are triggered; as a client for the Enactment Engine, it requests modifications to running choreographies by sumitting updated Choreography Deployers when notifications are received from the monitoring subsystem.

An example of such a daemon is available in the Enactment Engine source code repository, in the "reconfiguration" directory. We show below some code snippets from this example daemon and explain its general mechanism.

During startup, the daemon loads predefined rules from a static file and submits them to the Glimpse monitor:

```
public static void main(String[] args) {
    String rules = Manager.ReadTextFromFile(
        this.getClass().getClassLoader().getResource("rules/SLAViolations.xml").getFile());
    new EnactmentEngineGlimpseConsumer([... properties ...], rules);
}
```

Whenever a rule is triggered, the daemon is notified and runs the code from a class whose name is contained in the notification event:

```
public void messageReceived(Message arg0) {
    ObjectMessage responseFromMonitoring = (ObjectMessage) arg0;
    response = (ComplexEventResponse) responseFromMonitoring.getObject();
    event = new HandlingEvent(response.getResponseValue(), response.getRuleName());
    Class<ComplexEventHandler> theClass;
    theClass = (Class<ComplexEventHandler>) Class.forName(
        "org.ow2.choreos.chors.reconfiguration.events." + event.getEventData());
    handler = theClass.newInstance();
    handler.handleEvent(event);
}
```

The submitted example rules define that whenever more than 5% of requests during the last 2 minutes have a response time above 120 miliseconds, new replicas of the service should be created (the class AddReplica should be run):

```
    when
        $ev : ResponseTimeEvent() over window:time(2m);

        Number( $eventSum : doubleValue ) from accumulate(
            $event : ResponseTimeEvent($ev.service == service, $ev.chor == chor, $ev.ip == ip)
                over window:time(2m), count($event)
        );

        Number( intValue > $eventSum*0.05 ) from accumulate(
            $sEvent : ResponseTimeEvent(
                value > 120, $ev.service == service, $ev.chor == chor, $ev.ip == ip)
                over window:time(2m), count($sEvent)
        );

    then
        ResponseDispatcher.NotifyMeValue("AddReplica",
            "eeConsumer", (String) $ev.ip, (String) $ev.service);
    end
```

Finally, the AddReplica class then interacts with the Enactment Engine to update the number of replicas of the service:

```
public void handleEvent(HandlingEvent event) {
    List<DeployableService> services = registryHelper.getServicesHostedOn(event.getNode());
    List<DeployableServiceSpec> serviceSpecs = registryHelper.getServiceSpecsForServices(services);

    Choreography c = registryHelper.getChor(event.getNode());
    ChoreographySpec cSpec = c.getChoreographySpec();

    for (DeployableServiceSpec spec : serviceSpecs) {
        for (DeployableServiceSpec s : cSpec.getDeployableServiceSpecs()) {
            if (s.getName().equals(spec.getName())) {
                s.setNumberOfInstances(s.getNumberOfInstances() + 1);
                break;
            }
        }
    }

    registryHelper.getChorClient().updateChoreography([... id ...], cSpec);
    registryHelper.getChorClient().enactChoreography([... id ...]);
}
```