Orchestration for the Deployment of Distributed Applications with Geographical Constraints in Cloud Federation

Massimo Villari(✉), Giuseppe Tricomi, Antonio Celesti, and Maria Fazio

Department of Engineering, University of Messina, Messina, Italy
{mvillari,gtricomi,acelesti,mfazio}@unime.it

Abstract. This paper presents a system developed in the Horizon 2020 BEACON project enabling the deployment of distributed applications in an OpenStack-based federated Cloud networking environment. In such a scenario, we assume that a distributed application consists of several microservices that can be instantiated in different federated Cloud providers and that users can formalize advanced geolocation deployment constrains. In particular, we focus on an Orchestration Broker that is able to create ad-hoc manifest documents including application deployment instructions for the involved federated Cloud providers and users' requirements.

1 Introduction

Nowadays, federated Cloud networking [1] represents an interesting scenario for the deployment of distributed applications. In this paper, we describe the results obtained by the Horizon 2020 BEACON Project in terms of Cloud brokering for the deployment of distributed applications in federated OpenStack-based Cloud networking environments [2]. In such a scenario, we assume that a distributed application consists of several microservices that can be instantiated in different federated Cloud providers and that users can specify advanced geolocation deployment constrains. In particular, we present an Orchestration Broker that is able to create ad-hoc service manifest documents including application deployment instructions destined to selected federated Cloud providers and users' requirements. The Orchestration Broker interacts through RESTFUL communications with federated OpenStack Clouds through their own HEAT orchestration systems.

The purposes of this paper is not to define a new standard for addressing application deployment but to extend the Heat Orchestrator Template (HOT) [3] resource set in order to manage the federated deployment of distributed applications. The Orchestration Broker analyses the HOT service manifest of an application and automatically extracts the elements able to describe how microservices have to be deployed in federated OpenStack Clouds via their HEAT systems. An important feature of this approach is that the Orchestrator Broker is able to select target federated Clouds according to their geographic location. In fact, a “borrower”, i.e., a Cloud federation client, can exactly specify the application requirements along with the geographical locations where the microservices of a distributed application have to be deployed.

The rest of the paper is organized as follows: Sect. 2 describes related works. Section 3 presents the Orchestrator Broker design. Section 4 describes implementation highlights. Section 5 concludes the paper also providing lights to the future.

2 Related Work and Background

Cloud federation raises many challenges in different research fields as described in [4–7]. Most of scientific works focus on architectural models able to efficiently support the collaboration between different Cloud providers according to different points of view. The recent trend has been to use Cloud federation for new challenging scenarios including Internet of Things (IoT) [6], Fog, Edge, and Osmotic Computing [8]. In [9] it is proposed a mathematical algorithm to improve the automatic scaling capabilities of a Cloud provider, based on a brokering approach. The adopted approach focuses on the selection of Cloud providers in order to create a federation able to maximize profits. In [10], it is proposed a selection algorithm that allows federated Cloud providers to determine and choose the best destination where to migrate their VMs according to green computing policies. A parametric decisioning algorithms for the distribution of the computational workload in a federated Cloud environment was presented in [11]. An architecture for the setup of a Platform as a Service (PaaS) for the deployment of distributed application was described in [12]. The aforementioned initiatives focus on algorithms and architectures for the distribution of the computational workload of application s among different Cloud providers, but they lack of a concrete deployment orchestration mechanism able to consider geographical constrains.

3 Architectural Design

The federation management system requires to manage several OpenStack Clouds that cooperate each other according to specific federation agreements. In order to achieve such a goal, we designed a federation management component named OpenStack Federation Flow Manager (OSFFM) that acts as Orchestrator Broker. It is responsible to interact with OpenStack Clouds under specific assumptions and to lead the deployment and management of distributed applications. The OSFFM was designed according to the following assumptions:

– The system is composed by twin Clouds; this means that Virtual Machine (VM) images, networks, users, key pair, security groups and other configurations are the same in each Clouds.
– Each OpenStack Cloud interacts with a component called Federation Agent able to set up OVN tunnel between with other Clouds.

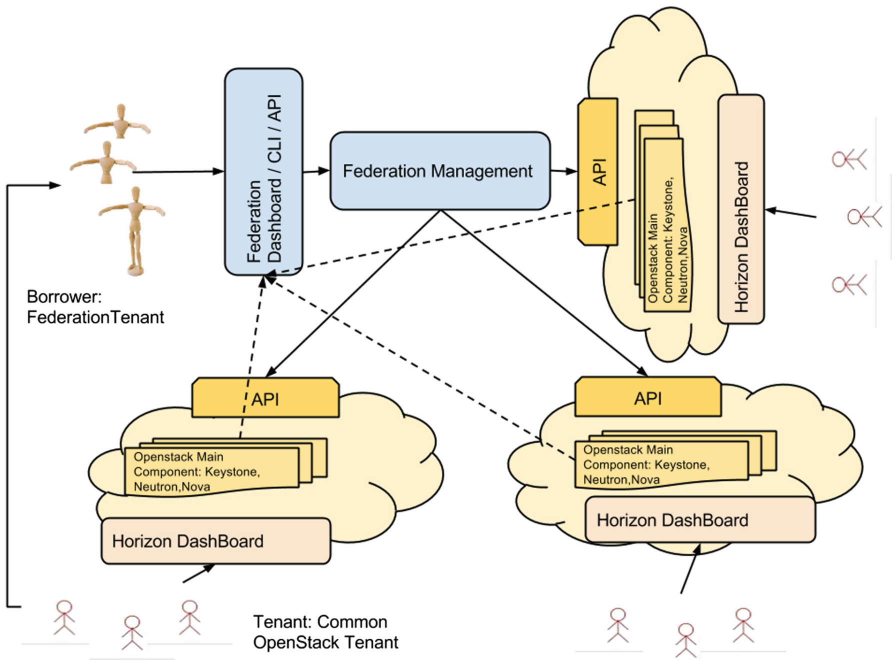

Federation Dashboard / CLI / API
Federation Management
API
Openstack Main Component: Keystone, Neutron,Nova
Horizon DashBoard
Borrower: FederationTenant
API
Openstack Main Component: Keystone, Neutron,Nova
Horizon DashBoard
API
Openstack Main Component: Keystone, Neutron,Nova
Horizon DashBoard
Tenant: Common OpenStack Tenant

Fig. 1. Overall high-level federation architecture.

Figure 1 shows the logical high-level Cloud federation architecture.

We can identify two types of actors: *tenant* of a Cloud involved in federation, *borrower* of a federation. The tenant is a subject (a person or a society) who/which uses the resource and services provided by a Cloud in order to provide, in turn, services to his/her/its clients. Instead, the borrower is considered as a tenant of the Cloud federation having an agreement with a particular Cloud for accessing federated services.

The federation management is implemented by means of the OSFFM component that represents the core of the Orchestrator Broker. It includes several Application Program Interfaces (APIs) for the interaction among borrowers, tenants and users of the Clouds involved in the federation. It carries out a federation coordinator task managing virtual machines and virtual networks. All produced big data for management and federation status are stored in a NoSQL database. Each OpenStack Cloud involved in the federation is supported by a VM running a software component named Federation Agent, that is responsible to create virtual networks and virtual paths among them. In order to achieve such a goal, each OpenStack Cloud was equipped with a OSFFM component whose architecture is shown in Fig. 2.

The OSFFM is connected with the other actors by means of the following interfaces:

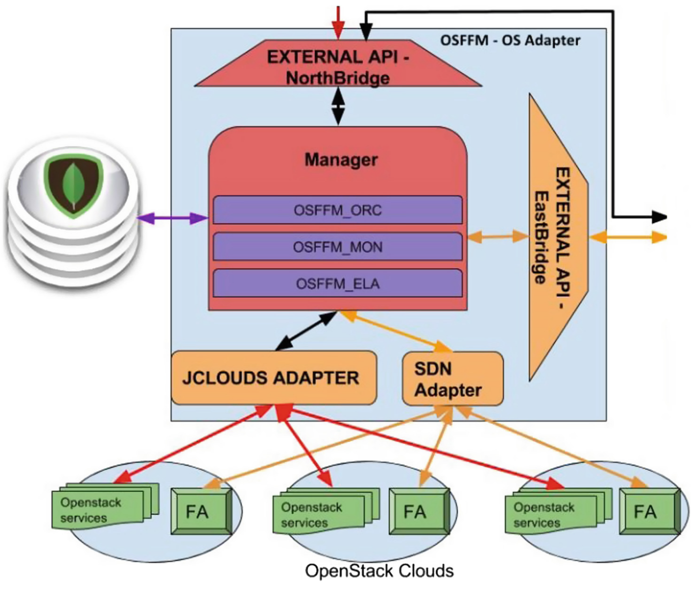


Fig. 2. OSFFM system architecture.

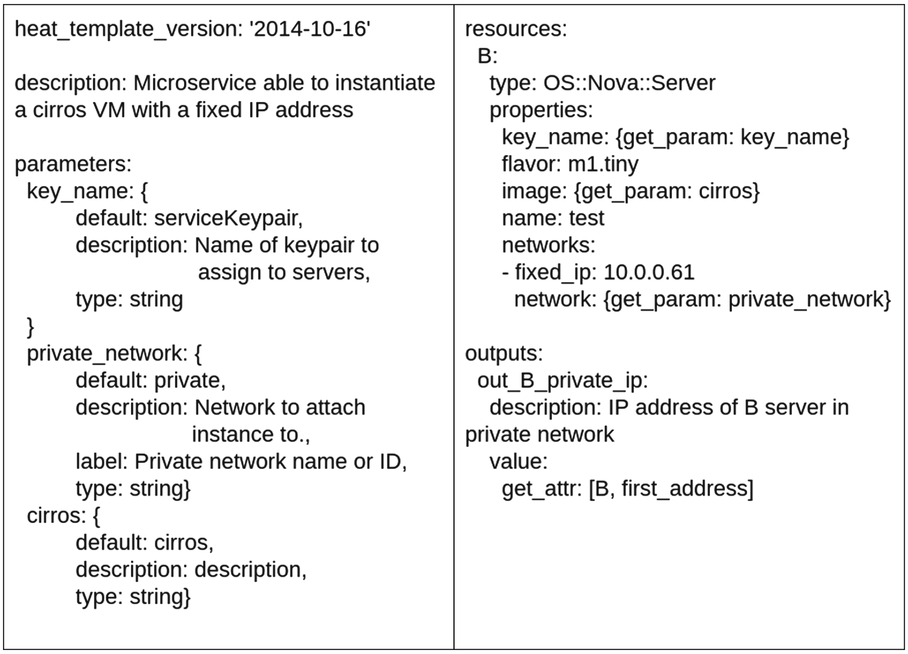
- **Southbridge.** It is used to send instructions to federated OpenStack Clouds. It includes two different adapters: jClouds and Software Defined Networking (SDN). The first one interacts with the OpenStack common module of each involved Cloud in order to retrieve information useful for the federation establishment process. Instead, the second one is used to provide information to the Federation Agent responsible to setup the federated network in which applications will be deployed.
- **Eastbridge.** It is used to send and receive instructions for the network management of the federated element involved in the application deployed. This interface acts both as REST server and client.
- **Northbridge.** It is used to receive application deployment instructions included in the service manifest.
- **Westbridge.** It is used to store status and management data, inside a NoSql database. In our example, it is represented by a MongoDB connector class.

The manager component includes three sub-modules that are OSFF Orchestrator (OSFFM_ORC), Monitoring (OSFFM_MON) and Elasticity Location Aware (OSFFM_ELA). All actions are performed by such sub-modules through the jClouds adapter that interacts with OpenStack Clouds.

OSFFM_ORC. This module is responsible to orchestrate all the tasks required for deploying a distributed application. In particular it receives as an input the HOT service manifest including guidelines for the deployment of a distributed application and builds several HOT microservice manifests that are sent to the heat modules of specific OpenStack Clouds for the instantiation of virtual resources in which microservices are deployed in. In order to do this it:

- pre-processes the pieces of information coming from others components and stores them in MongoDB,
- post-processes the pieces of information retrieved from MongoDB,
- creates connections between NorthBridge and SouthBridge interfaces in order to expose federated Clouds' data through a single point of inquiry;
- coordinates of the activity of the other OSFFM modules.

The OSFFM_ORC sub-module uses hash tables in order to store the result of its "manifest parsing" sub-elaboration activities; these activities are focused on retrieving pieces of federated deployment information and pieces information required to apply elasticity policies on the VM instantiated for the distributed application deployment. After such parsing activities of the HOT service manifest, OSFFM_ORC builds several HOT microservice manifests that are provided to specific federated Clouds. Figure 3 shows an example of HOT microservice manifest. The HOT standard has been extended in order to add new parameters that specify deployment requirements defined as resources. In this way, it is possible to compose a complex HOT service manifest by defining resources of



```
heat_template_version: '2014-10-16'

description: Microservice able to instantiate
a cirros VM with a fixed IP address

parameters:
  key_name: {
        default: serviceKeypair,
        description: Name of keypair to
                      assign to servers,
        type: string
  }
  private_network: {
        default: private,
        description: Network to attach
                     instance to.,
        label: Private network name or ID,
        type: string}
  cirros: {
        default: cirros,
        description: description,
        type: string}
```

```
resources:
  B:
    type: OS::Nova::Server
    properties:
      key_name: {get_param: key_name}
      flavor: m1.tiny
      image: {get_param: cirros}
      name: test
      networks:
      - fixed_ip: 10.0.0.61
        network: {get_param: private_network}

outputs:
  out_B_private_ip:
    description: IP address of B server in
private network
    value:
      get_attr: [B, first_address]
```

Fig. 3. Example of HOT microservice manifest.

other resources that can be queried by means of a simple "get_resource" function call. In particular, the following deployment requirements have been added: *(i)* service placement policies according to location constraints; *(ii)* location-aware elasticity rules; *(iii)* network reachability rules. The main methods involved for the HOT service manifest instantiation process are:

– *ManifestInstantiation.* This method is used to create an instance of a ManifestManager thread starting from an existing HOT service manifest. All ManifestManager threads are stored inside a hash map indexed through a service Manifest Unique IDentifier (UID).
– *ManagementGeoPolygon & ManagementRetrieveCredential.* These methods are used to retrieve from MongoDB, the credentials that borrowers hold in the Cloud datacenters placed particular geographical locations. These actions are complex because the credentials stored in the data model are stratified and are associate to a three-dimensional matrix whose dimension represents: Geographical Areas, Datacenters in Geographical Areas and Credentials valid in Datacenters. More specifically the first two levels of this structure are retrieved from the managementGeoPolygon method.
– *DeployManifest & SendShutSignalStack4DeployAction.* These methods are used to deploy a stack, i.e., a group of resources, in several federated Clouds belonging to a particular service group. The instantiated resources are all twins among them for fault-tolerance purposes. In fact, after the instantiation of VMs only a few of them are maintained in an active status, whereas the others ones are shut down.

OSFFM_ELA. This module is designed to control and maintain the performance of applications deployed in the federation. This goal is achieved by providing functions that allow to horizontally scale of resources in the federation. This module interacts with the target Cloud when a particular condition occurs in order to trigger a specific action. By interfacing this component with the monitoring one, it is possible to have the parameters needed to verify both the Cloud infrastructure and VMs internal states. When the previous information is correlated with other Clouds information, the module becomes able to make the required scalability decisions. The OSFFM_ELA interacts directly with OSFFM_ORC to accomplish the decision made and interacts with OSFFM_MON to receive status notifications about monitored condition.

OSFFM_MON. This module is designed as a collector of various monitoring flows coming from several monitoring components inside the federated clouds. The OSFFM_MON is interconnected with clouds via the SouthBridge interfaces and makes request in order to discover information needed to monitor resources state instantiated via the stacks. For OSFFM architecture, the monitoring solution is achieved by adapting the monitoring module used in OpenStack, that is the Ceilometer component, at a federation level and interconnected with the federated Ceilometer creating a hierarchical level structure.

4 OSFFM Orchestration Implementation

```java
public void manifestinstatiation(String manName,String tenant){
   JSONObject manifest=null;
   this.addManifestToWorkf(manName, manifest);
   ManifestManager mm=(ManifestManager)OrchestrationManager.mapManifestThr.get(manName);
   this.manageYAMLcreation(mm, manName,tenant);
```

Listing 1. Function manifestinstatiation

The OSFFM was developed in JAVA, and exposes REST-API interfaces in order to avoid platform constraint during its usage. According to HEAT-API, also service manifest used in our solution are based on YAML, this makes OSFFM able to manipulate the manifest disrupting, and manifest composing, without problems.

```java
public HashMap<String,ArrayList<ArrayList<String>>> managementgeoPolygon(
    String manName,MDBInt.DBMongo db,String tenant)
    {
      HashMap<String,ArrayList<ArrayList<String>>> tmp=new
      HashMap<String,ArrayList<ArrayList<String>>>();
      ManifestManager mm=(ManifestManager)OrchestrationManager.mapManifestThr.get(manName);
      Set s=mm.table_resourceset.get("OS::Beacon::ServiceGroupManagement").keySet();
      Iterator it=s.iterator();
      boolean foundone =false;
      while(it.hasNext()){
        String serName=(String)it.next();
        SerGrManager sgm=(SerGrManager)mm.serGr_table.get(serName);
        ArrayList<MultiPolygon> ar=null;
        try{
            ar=(ArrayList<MultiPolygon>)mm.geo_man.retrievegeoref(
                sgm.getGeoreference());
        }catch(NotFoundGeoRefException ngrf){...}
        ArrayList dcInfoes=new ArrayList();
        for(int index=0;index<ar.size();index++){
          try{
            ArrayList<String> dcInfo=
              db.getDatacenters(tenant,ar.get(index).toJSONString());
              if(dcInfo.size()!=0){
                dcInfoes.add(dcInfo);
                foundone=true;
              }
          }
          catch(org.json.JSONException je){...}
        }
        tmp.put(serName, dcInfoes);
        if(!foundone) return null;
      }
      return tmp;
}
```

Listing 2. Implementation of the *managementGeoPolygon* method.

```
public ArrayList<ArrayList<HashMap<String, ArrayList<Port>>>> deployManifest(
  String template,
  String stack,
  HashMap<String, ArrayList<ArrayList<OpenstackInfoContainer>>> tmpMapcred,
  HashMap<String,ArrayList<ArrayList<String>>> tmpMap,
  DBMongo m){
    String stackName = stack.substring(stack.lastIndexOf("_") + 1 > 0 ?
    stack.lastIndexOf("_") + 1 : 0, stack.lastIndexOf(".yaml") >= 0 ?
    stack.lastIndexOf(".yaml") : stack.length());
    ArrayList arDC=(ArrayList<ArrayList<String>>)tmpMap.get(stackName);
    ArrayList arCr=(ArrayList<ArrayList<OpenstackInfoContainer>>)
      tmpMapcred.get(stackName);
    ArrayList<ArrayList<HashMap<String, ArrayList<Port>>>> arMapRes =
      new ArrayList<>();
    boolean skip = false, first = true;
    int arindex = 0;
    while (!skip){
      ArrayList tmpArDC = (ArrayList<String>) arDC.get(arindex);
      ArrayList tmpArCr = (ArrayList<OpenstackInfoContainer>) arCr.get(arindex);
      ArrayList<HashMap<String, ArrayList<Port>>> arRes = new
        ArrayList<HashMap<String, ArrayList<Port>>>();
      for (Object tmpArCrob : tmpArCr) {
        boolean result = this.stackInstantiate(template, (OpenstackInfoContainer)
          tmpArCrob, m, stackName);
        String region = "RegionOne";
        ((OpenstackInfoContainer) tmpArCrob).setRegion(region);
        HashMap<String, ArrayList<Port>> map_res_port =
          this.sendShutSignalStack4DeployAction(stackName,
          (OpenstackInfoContainer) tmpArCrob, first, m);
        if(result){
          first = false;
          arRes.add(map_res_port);
        }
        arindex++;
        arMapRes.add(arRes);
        if(arindex > tmpArCr.size()) skip = true;
     }
     return arMapRes;
   }
}
```

Listing 3. Implementation of the *deployManifest* method.

In our implementation we considered the OpenStack Mitaka release along with the OVN integration for Neutron. In the following, we provide a few implementation highlights regarding the main previously described OSFFM_ORC methods. The *manifestInstantiation* method is defined in the OrchestrationManager class and it is the manifest analysis workflow entry point. It allows to create a new object of ManifestManager moved inside a support HashMap used to bind Manifest with its ManifestManager thread. The manifest instantiation code is shown in Listing 1. After this Manifest splitting process, the ManifestManager starts the reconstruction in order to create the temporary template following the service groups directives written inside the service manifest. The *management-GeoPolygon* method is used to create a Hash Map storing three-dimensional parameters for each service group that has been found in the service manifest. Its implementation is shown in Listing 2. The *managementRetrieveCredential* method retrieves from MongoDB the access credentials related to the borrower that are valid in the Cloud selected by the previous function. Listings 3 and 4 respectively show the java code of the *deployManifest* and *sendShutSignal-*

Stack4DeployAction methods that are strictly linked. The first one prepares all information needed by the second one, the real executor of the deployment task. These methods are used to deploy a stack in the federation according to the service replication purposes. The performed actions are used for the resources instantiation on all Clouds selected in the service manifest for a particular service group.

```
public HashMap<String, ArrayList<Port>> sendShutSignalStack4DeployAction(
        String stackName, OpenstackInfoContainer credential,boolean first, DBMongo m) {
    try {
       Registry myRegistry = LocateRegistry.getRegistry(ip,port);
       RMIServerInterface impl = (RMIServerInterface) myRegistry.lookup("myMessage");
       ArrayList resources =impl.getListResource(credential.getEndpoint(),
          credential.getUser(),credential.getTenant(),credential.getPassword(),stackName);
       boolean continua=true;
       NovaTest nova = new NovaTest(credential.getEndpoint(), credential.getTenant(),
                credential.getUser(), credential.getPassword(), credential.getRegion());
       NeutronTest neutron = new NeutronTest(credential.getEndpoint(),
          credential.getTenant(), credential.getUser(), credential.getPassword(),
          credential.getRegion());
       HashMap<String, ArrayList<Port>> mapResNet = new HashMap<String, ArrayList<Port>>();
       Iterator it_res = resources.iterator();
       while (it_res.hasNext()) {
          String id_res = (String) it_res.next();
          if(!first){
              nova.stopVm(id_res);
              m.updateStateRunTimeInfo(credential.getTenant(), id_res, first);
          }
          ArrayList<Port> arPort = neutron.getPortFromDeviceId(id_res);
          mapResNet.put(id_res, arPort);
          Iterator it_po = arPort.iterator();
          while (it_po.hasNext()) {
            m.insertPortInfo(credential.getTenant(),
            neutron.portToString((Port)it_po.next()));
          }
        }
        return mapResNet;
    }catch (Exception e){
      ...
      return null;
    }
}
```

Listing 4. Implementation of the *sendshutsignalstack4deployaction* method.

4.1 The New HOT Manifest

```
federation:
    type: OS::Beacon::ServiceGroupManagement
    properties:
      name: GroupName
      geo_deploy: { get_resource: geoshape_1}
      resource:
        groups:  {get_resource:  A}
```

Listing 5. New Model of resource OS::Beacon::ServiceGroupManagement

The HOT service manifest is typically provided by the borrower and it is an advanced version of HOT, because it is enriched with a set of new resource types that are extracted by the Orchestrator Broker and processed separately. These resources extend the HOT capabilities and are used by the OSFFM to compose a series of HOT microservice manifests. In addition, it is possible to use the HOT syntax in order to formalize the requirements for the deployment of a borrowers' distributed application. As previously discussed, such requirements include: *(i)* Service placement policies according to location constraints; *(ii)* Location-aware elasticity rules; *(iii)* Network reachability rules. In order to address the aforementioned requirements, in the context of the BEACON H2020 project we added to the HOT-based service manifest the following new definition of resources: OS::Beacon::ServiceGroupManagement and OS::Beacon::Georeferenced_deploy. *OS::Beacon::ServiceGroupManagement* is related requirements *(i)* and *(ii)*. This resource type allows to specify the geographical information for the deployment of a specific group by means of the geo_deploy field. Listing 5 shows an example of such a resource. Instead, *"OS::Beacon::Georeferenced_deploy"* allows to define an array of polygon (defined in GeoJSON format as MultiPolygon) that identifies the areas where a resource could be allocated. Listing 6 shows an example of such a resource.

```
geoshape_1:
    type: OS::Beacon::Georeferenced_deploy
    properties:
      label: Shape label
      description: descripition
      shapes: [{"type":"Feature","id":"BEL","properties":{"name":"Belgium"},
      "geometry":{"type":"Polygon","coordinates":[[[3.314971,51.345781],
      [4.047071,51.267259],[4.973991,51.475024],[5.606976,51.037298],[6.156658,50.803721],
      [6.043073,50.128052],[5.782417,50.090328],[5.674052,49.529484],[4.799222,49.985373],
      [4.286023,49.907497],[3.588184,50.378992],[3.123252,50.780363],[2.658422,50.796848],
      [2.513573,51.148506],[3.314971,51.345781]]]}}]
```

Listing 6. New Model of resource OS::Beacon::Georeferenced_deploy

5 Conclusion and Future Work

In this paper, we proposed an approach for the orchestration deployment of distributed applications in federation Cloud environments. In particular, an Orchestration Broker is presented. It is able to process a HOT service manifest and to produce different corresponding microservice manifests destined to different federated OpenStack-based Clouds providers. An important feature of this approach is the ability to select target federated Clouds as function of their geographical position. In fact, a borrower can select exactly the geographical area where a distributed application has to be deployed. In future works, we will focus on the enhancement of the maintenance tasks performed by the OSFFM_ELA and OSFFM_MON modules on the virtual resources in which a distributed application is deployed.

Acknowledgment. This work was supported by the European Union Horizon 2020 BEACON project under grant agreement number 644048.

References

1. Moreno-Vozmediano, R., et al.: BEACON: a cloud network federation framework. In: Celesti, A., Leitner, P. (eds.) ESOCC Workshops 2015. CCIS, vol. 567, pp. 325–337. Springer, Cham (2016). doi:10.1007/978-3-319-33313-7_25
2. Celesti, A., Levin, A., Massonet, P., Schour, L., Villari, M.: Federated networking services in multiple OpenStack clouds. In: Celesti, A., Leitner, P. (eds.) ESOCC Workshops 2015. CCIS, vol. 567, pp. 338–352. Springer, Cham (2016). doi:10.1007/978-3-319-33313-7_26
3. Heat Orchestration Template (HOT) specification. http://docs.openstack.org/developer/heat/template_guide/hot_spec.html
4. Vernik, G., Shulman-Peleg, A., Dippl, S., Formisano, C., Jaeger, M., Kolodner, E., Villari, M.: Data on-boarding in federated storage clouds. In: 2013 IEEE Sixth International Conference on Cloud Computing (CLOUD), pp. 244–251 (2013)
5. Azodolmolky, S., Wieder, P., Yahyapour, R.: Cloud computing networking: challenges and opportunities for innovations. IEEE Commun. Mag. **51**, 54–62 (2013)
6. Celesti, A., Fazio, M., Villari, M.: Enabling secure XMPP communications in federated IoT clouds through XEP 0027 and SAML/SASL SSO. Sensors **17**, 1–21 (2017)
7. Celesti, A., Celesti, F., Fazio, M., Bramanti, P., Villari, M.: Are next-generation sequencing tools ready for the cloud? Trends Biotechnol. **35**, 486–489 (2017)
8. Villari, M., Fazio, M., Dustdar, S., Rana, O., Ranjan, R.: Osmotic computing: a new paradigm for edge/cloud integration. IEEE Cloud Comput. **3**, 76–83 (2016)
9. Mashayekhy, L., Nejad, M.M., Grosu, D.: Cloud federations in the sky: formation game and mechanism. IEEE Trans. Cloud Comput. **3**, 14–27 (2015)
10. Giacobbe, M., Celesti, A., Fazio, M., Villari, M., Puliafito, A.: An approach to reduce carbon dioxide emissions through virtual machine migrations in a sustainable cloud federation. In: 2015 Sustainable Internet and ICT for Sustainability (SustainIT), Institute of Electrical & Electronics Engineers (IEEE) (2015)
11. Panarello, A., Breitenbcher, U., Leymann, F., Puliafito, A., Zimmermann, M.: Automating the deployment of multi-cloud application in federated cloud environments. In: Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools (2017)
12. Celesti, A., Peditto, N., Verboso, F., Villari, M., Puliafito, A.: DRACO PaaS: a distributed resilient adaptable cloud oriented platform. In: IEEE 27th International Parallel and Distributed Processing Symposium (2013)