CHAPTER 4

Application Blueprints and Service Description

Ioan Dragan, Teodor-Florin Fortiș, Marian Neagul, Dana Petcu, Teodora Selea, and Adrian Spataru

Abstract In the context of creating a self-organising and self-managing cloud infrastructure we propose a set of extensions to the existing Service Description Languages (SDLs) and Application Blueprints in order to establish a common ground for the various CloudLightning components. By implementing this SDL and all the missing links one can assure that the CloudLightning system works in such a way that users can easily interact with it. In this chapter we present in detail the design decisions that were made during the development of various components alongside with their formal description.

I. Dragan (✉)
Victor Babeș University of Medicine and Pharmacy, Timișoara, Romania

Institute e-Austria Timisoara, Timișoara, Romania
e-mail: idragan@ieat.ro

T.-F. Fortiș • M. Neagul • D. Petcu • T. Selea • A. Spataru
Institute e-Austria Timisoara, Timișoara, Romania

West University of Timișoara, Timișoara, Romania
e-mail: florin.fortis@e-uvt.ro; marian.neagul@e-uvt.ro; Dana.Petcu@e-uvt.ro; adrian.spataru@e-uvt.ro

© The Author(s) 2018
T. Lynn et al. (eds.), *Heterogeneity, High Performance Computing, Self-Organization and the Cloud*, Palgrave Studies in Digital Business & Enabling Technologies,
https://doi.org/10.1007/978-3-319-76038-4_4

Keywords Service Description Language • CL Blueprints • CL Gateway Service • Lifecycle and resource management

4.1 Introduction

To deliver the quality of service (QoS) expected by end users on a distributed multi-tenant infrastructure requires careful management of computing resources. This is particularly the case where there is a rapid growth in usage such as cloud computing. Cloud service providers (CSPs) are faced with a myriad of challenges in meeting the needs of a large and diverse range of end users including, but not limited to, service transparency, automated service provisioning, efficiently managing workload segmentation and portability, and managing virtual services instances at one level, while optimising the utilisation of all resources at a different level (Sun et al. 2012). The issues can be resolved through specialised and precise cloud service specification models, Service Description Languages (SDLs), describing cloud services, their deployment specifications, and the required resources to run these cloud services. The majority of the existing SDLs and associated frameworks implement tools, Application Programming Interfaces (APIs), and strategies for managing the lifecycle of cloud applications and/or resources, and they are usually provided as a self-service interface to Enterprise Application Operators (EAOs). This self-service approach allows an EAO to have full control over the management of applications as well as the underlying resources such as virtual machines (VMs) and containers. It subsequently narrows down the opportunities for CSPs to improve resource utilisation and potentially the quality of services.

The CloudLightning architecture endeavours to create a service-oriented architecture for the evolving heterogeneous cloud. In this respect, it is imperative to maintain a separation between application lifecycle management and resource management. This separation of concerns implements a "what-how" approach where the user concentrates on "what" needs to be done, while the CSP concentrates on "how" it should be done. With such an approach, it will be possible to implement continuous improvements, in terms of resource utilisation and service delivery, at the resource level. From this perspective, SDLs facilitate both (a) application lifecycle management by the user and (b) resource management by the CSP. As such, they ensure a proper separation of concerns between stakeholders, a core design principle of CloudLightning introduced in Chap. 1. Particular service offerings are captured in blueprints to assist end users to discover and select from an increasing catalogue of services and determine an optimal, and potentially heterogeneous, set of resources to implement them. The remainder of this chapter is organised as follows. The next section provides an overview of two representative application lifecycle frameworks and one representative resource management framework. This is followed by an overview of the specific stakeholders whose concerns are of interest to CloudLightning. The CloudLightning approach to separation of concerns is then described followed by the Gateway Service and its functionalities. Formal definition of the CloudLightning Service Description Language (CL-SDL) is provided in Sect. 4.4 followed by an exemplar implementation. This chapter concludes with a summary and future work on the components and concepts presented in the chapter.

4.2 Representative Application Lifecycle and Resource Management Frameworks

In order to identify concerns about the classical, vertical management approach to cloud computing application lifecycle and resource management, three representative frameworks are used for illustrative purposes: OpenStack Solum, Apache Brooklyn, and OpenStack Heat.

The cloud application lifecycle management architecture is represented in Fig. 4.1, using OpenStack Solum and Apache Brooklyn frameworks for Platform as a Service (PaaS) cloud, and resource lifecycle management using OpenStack Heat mainly for Infrastructure as a Service (IaaS) cloud.

Project Solum and Apache Brooklyn allow the user to deploy a cloud application or a group of cloud applications previously described in a blueprint, using an SDL. The main purpose of such an SDL is to provide a way of expressing the management processes for cloud applications. Depending on the actual implementations, this may include providing the ability for describing the characteristics of the application components, deployments scripting, dependencies, locations, logging, policies, and so on.

In the case of OpenStack Solum, the engine takes a blueprint as an input and converts it to a Heat Orchestration Template (HOT) that can be understood by the application and resource management engine (OpenStack Heat). The Heat engine, thereafter, calls the corresponding service APIs that are offered by the cloud infrastructure framework such as OpenStack.

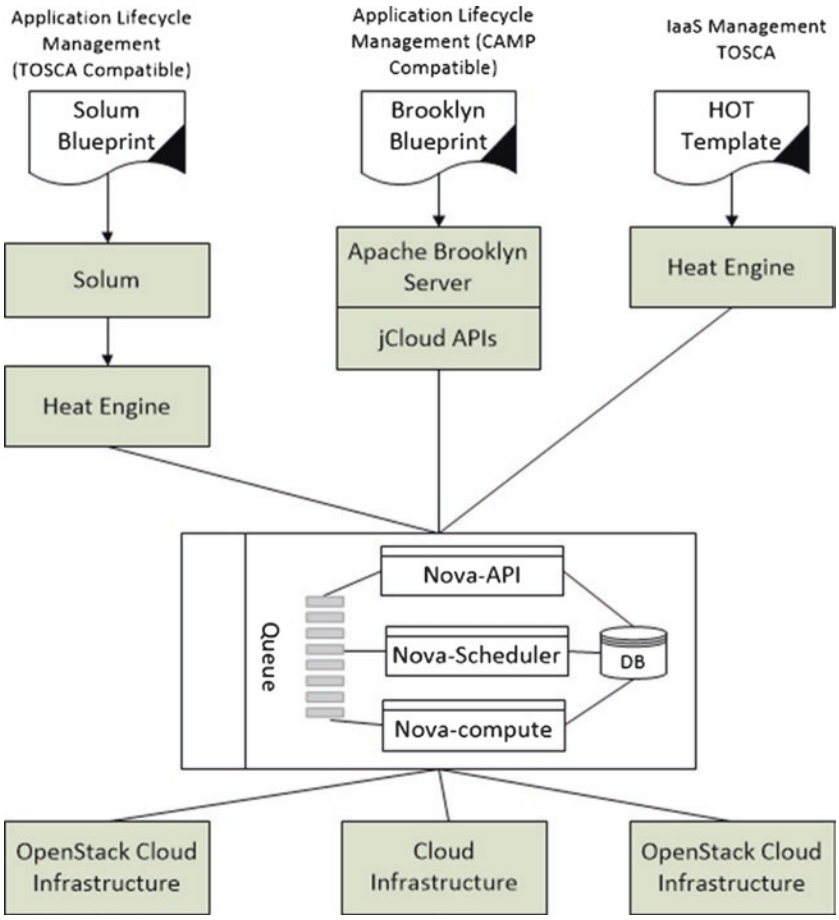


Fig. 4.1 Lifecycle management for OpenStack Solum, Apace Brooklyn, and OpenStack Heat

In contrast, Apache Brooklyn converts a blueprint into a series of API calls (specifically, jCloud APIs) that can be used to directly contact the underlying cloud infrastructure. For example, these calls may reach the cloud infrastructure with a request for creating a VM in OpenStack; the OpenStack Nova API service will capture the request and send it to nova-scheduler, which, in turn, decides on the physical server on which the VM should be started on. This approach is based on a request-response pattern, providing a simple, robust, and efficient implementation. However, as each request is processed independently, when blueprints are specifying, for example, placement constraints based on vicinity of resources, such a constraint is hard to be captured and fully implemented by APIs with a vertical approach.

4.3 CloudLightning Stakeholders and Associated Concerns

Separation of concerns requires the identification of stakeholders and their associated concerns. For illustrative purposes, three distinct entities are identified—end users, Enterprise Application Operators and Developers (EAO/EAD), and IaaS resource providers (CSPs) each with differing concerns. The end user is the consumer of an application and/or service. As such, their concerns are primarily related to cloud application continuity, availability, performance, security, and business logic correctness. The EAO/EAD has traditional enterprise concerns, for example, cloud application configuration management, performance, load balancing, security, availability, and the deployment environment. As discussed in Chap. 1, the CSP's business model is driven by cost effectiveness and scalability while at the same time delivering the contracted service level. As such, their concerns are primarily related to optimisation including resource availability, operating costs (including power consumption), resource provisioning, resource organisation, and partitioning (if applicable).

Under separation of concerns, each entity manages their own concerns, to the extent that they can. Notwithstanding this, some concerns exist across the entities. For example, in order to realise high availability, an EAO may need to configure a load-balancer, while at the same time a CSP must implement a host-affinity policy.

4.4 The CloudLightning Approach Based on Separation of Concerns

4.4.1 CloudLightning Requirements

As discussed, the CloudLightning service delivery model depicted in Fig. 4.2 is a blueprint-based one. In contrast to existing frameworks, this service delivery model provides facilities for blueprint developers to specify

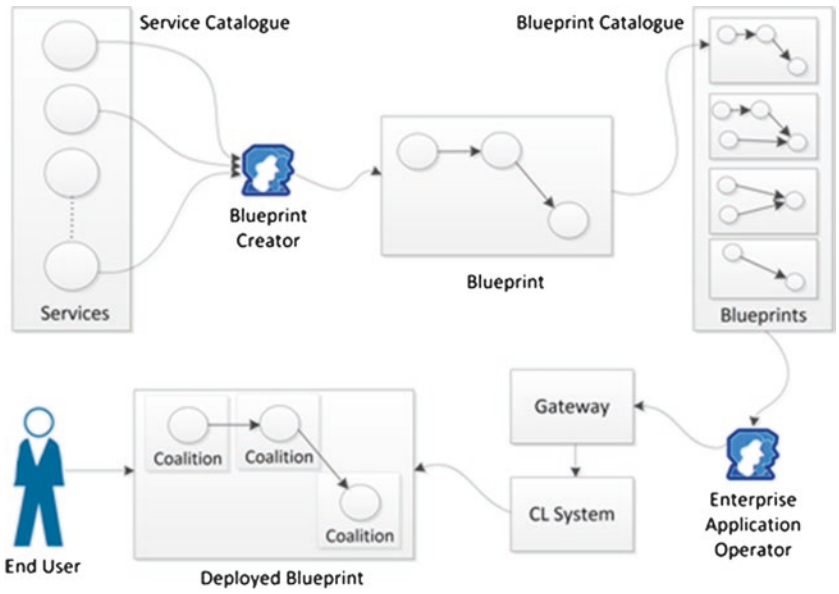


Fig. 4.2 CloudLightning service delivery model

comprehensive constraints and quality of service parameters for services and/or resources in the scope of a blueprint, by means of a specific SDL (the CL-SDL). Based on the specified constraints and parameters, it is then possible to provide an initial optimal deployment of the resources, a capability which has not been accomplished by previous solutions: for example, by placing resources (such as VMs) on the adjacent physical servers to minimise communication delay or allocating containers that have Graphical Processing Units (GPUs) or Xeon Phis attached to them to balance between performance and cost.

More importantly, in order to separate the concerns of cloud application lifecycle management and the resource lifecycle management, a CloudLightning-specific blueprint (CL-Blueprint) must be decomposed into two separate and interrelated blueprints, the first one for resource management (offering the Resource Template) and the other one for application/workflow management (defining framework-specific templates). This process is shown in Fig. 4.3. It also implies that the

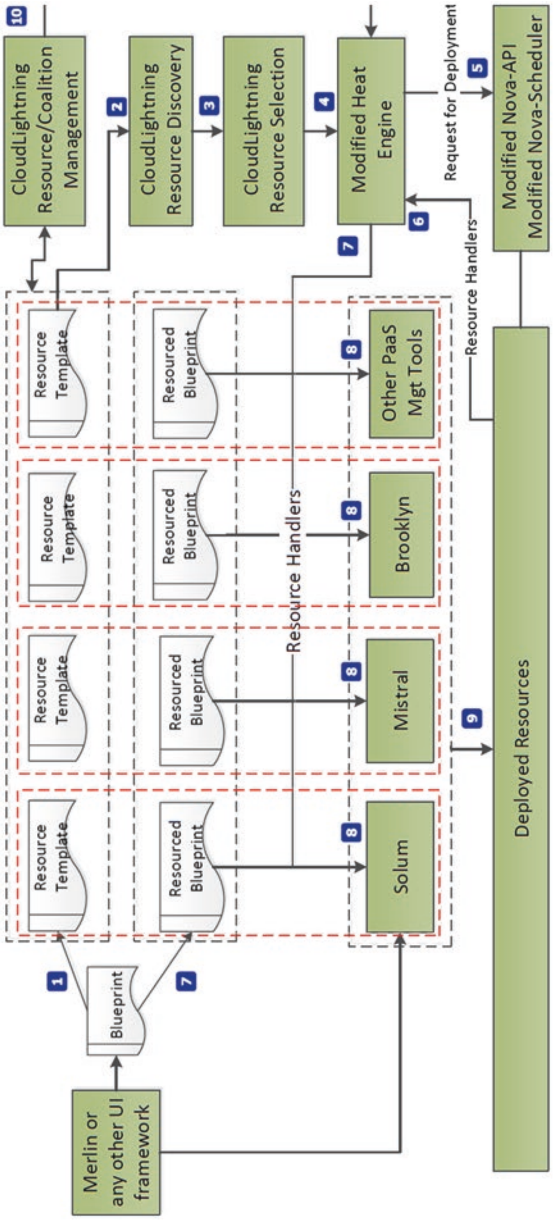


Fig. 4.3 Architecture for CloudLightning service delivery

CL-SDL shall be developed in such a way that a CL-Blueprint described in the CL-SDL can be transformed to framework-specific blueprints without losing generality.

A CL-Blueprint deployment starts from sending the raw Resource Template to a Resource Discovery component and a Resource Selection component, which are the two main components of a complementary system (in this situation, the CloudLightning Self-Organising and Self-Management [SOSM] framework), for optimal resource identification in the scope of a blueprint, as indicated in Fig. 4.3. Once the optimal resource identification process has finished, the initially received Resource Template must be reconstructed in order to embed the received resource optimisation information and consequently send it to the resource lifecycle management engine, which will carry out the actual resource deployment on the infrastructure it manages.

In addition, some of the optimisation information (e.g., on which physical server should this VM be allocated) must be embedded into resource requests (API calls), and this special information must be captured by the lower infrastructure management components.

The returns from the deployment process are the resource handlers (e.g., a resource handler can be a login account with username, access key, and Internet Protocol address to a VM, a container, a bare metal machine with pre-installed operating system, or an existing High Performance Computing [HPC] cluster). These resource handlers will then be returned to the Gateway Service, which will reformulate the original workflow/application blueprint along with the resource handlers.

The newly formulated workflow/application blueprint will then be submitted to the corresponding workflow/application lifecycle management framework to carry out the deployment of the cloud applications on these pre-provisioned resources. This process is shown in Fig. 4.3. To this end, a CL-Blueprint deployment process is complete.

Notice that this service delivery model is much more sophisticated when compared to the current self-service model using a vertical management approach, as the cloud application management and the resource management operate independently. Moreover, the cloud application management layer constantly needs to exchange information with resource management layers in certain circumstances (e.g., when ending the lifetime of a CL-Blueprint, a notification needs to be sent to the resource management layer so that the underlying resources can be reused or decommissioned).

In order to align with the design of the bespoke service delivery model, and implement the separation of concerns, the specific SDL shall be developed with following capabilities:

1. To describe characteristics of a cloud application
2. To describe cloud application execution environment and dependencies
3. To specify cloud application deployment processes
4. To specify resource type and resource requirements
5. To express constraints between blueprint service elements
6. To express quality of service parameters for each individual blueprint service element
7. To accommodate extensions for supporting specific/non-traditional cloud applications such as HPC applications
8. To fulfil above requirements without losing generality

4.4.2 Separation of Concerns

During the lifetime of the CL-Blueprint, the EADs/EAOs are responsible for managing the cloud applications through specific frameworks, such as Apache Brooklyn and OpenStack Solum, while the CloudLightning SOSM system manages the underlying resources. A series of advantages of this approach may be then highlighted:

1. continuous improvement on quality of CL-Blueprint services
2. improving service delivery and user experience by reusing resources that have already been provisioned
3. resource optimisations and energy efficiency optimisation
4. flexible and extensible when integrating other management system such as the OpenStack Mistral (Openstack.org 2017) workflow management system

In CloudLightning, the functional components that realise the concept of the "separation of concerns" are shown in Fig. 4.4 with the following description.

4.4.2.1 Application Lifecycle Management

- *Abstract Blueprint*: used to represent specific application requirements, constraints, and metrics defined by users, and describe

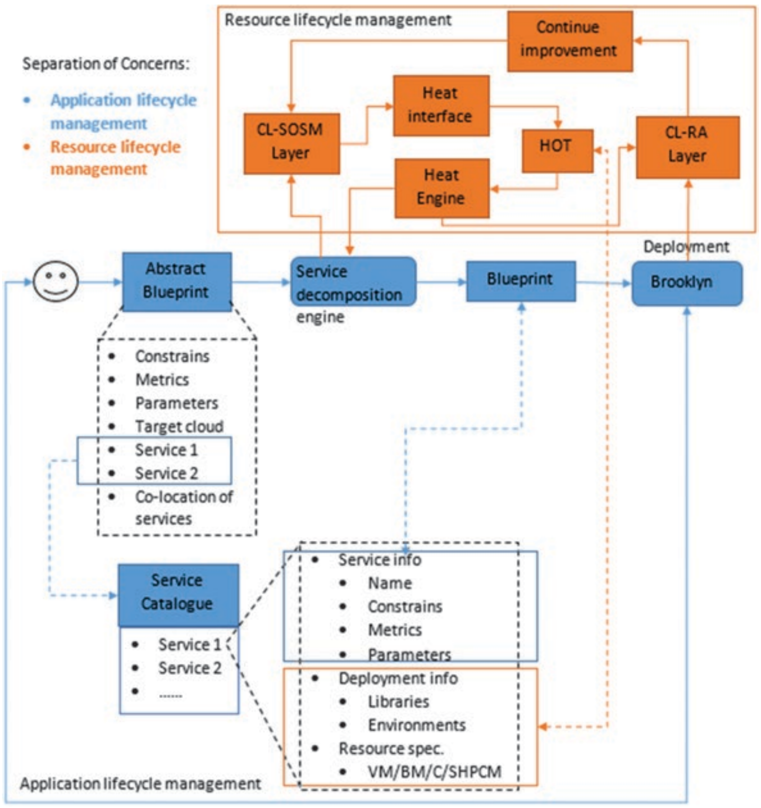


Fig. 4.4 CloudLightning implementation of the "separation of concerns"

the concrete and abstract services (referenced only by identification) alongside with the collocation of the services.

- *Blueprint*: represents a fully qualified Cloud Application Management for Platforms (CAMP) (Organization for the Advancement of Structured Information Standards [OASIS] CAMP TC, 2014) Document containing references to real resource types, resource locations, and deployment mechanisms, which are fully understood and handled by a CAMP-compliant implementation.
- *Service Catalogue*: it is a persistent collection of versioned services, each of which includes service information, deployment information, and CL-Resource specification.

- *Service Decomposition Engine (SDE)*: handles the transformation of Abstract Blueprints to concrete Blueprints according to provided requirements.
- *Brooklyn*: used for deploying and managing the applications via Blueprints.

4.4.2.2 *Resource Lifecycle Management*

- *CL-SOSM Layer*: CloudLightning SOSM Layer aims to identify and create/allocate the optimal CL-Resource for applications using principles of SOSM.
- *CL-RA Layer*: CloudLightning Resource Abstraction Layer is used for abstracting the CL-Resources in different ways (such as Bare Metal, Virtualisation, Containerisation, and Direct Access) from various hardware types (such as Central Processing Unit [CPU], GPU, Data Flow Engine, and Many Integrated Core [MIC]).
- *Heat Orchestration Template (HOT)*: describes the infrastructure resource (such as servers, networks, routers, floating IPs, and volume) for a cloud application, as well as the relationships between resources.
- *Heat Interface*: automatically generates HOTs in terms of the results from SOSM Layer or dynamically modifies HOTs based on the results from the Continued Improvement component.
- *Heat Engine*: manages the whole lifecycle of the provisioning process.
- *Continued Improvement*: this management component together with Heat and telemetry does the continued improvement for the deployed blueprint during the lifetime.

4.5 The CloudLightning Gateway Architecture

Integration of the use cases provided in CloudLightning with the Gateway Service will be done by following the CL-SDL (Xiong et al. 2016). The proposed CL-SDL specification is built on top of the OASIS CAMP specification and introduces new concepts suitable for expressing the requirements of HPC applications.

The syntax of the CL-SDL is based on the Brooklyn blueprint YAML (Yet Another Markup Language) and is used to describe the Resource Template and the Resourced Blueprint. Both of these offer support for CloudLightning Blueprint lifecycle management.

The Blueprint is used to represent specific application requirements, constraints, and metrics defined by either the EAD or the EAO, and describe services by name and their relationships. As depicted in Fig. 4.5, service definitions are predefined by EADs in special catalogues that follow the Cloud Service Archive (CSAR) specifications (Breiter et al. 2012), a subset of rules defined by the Topology and Orchestration Specification for Cloud Applications (TOSCA) standard (OASIS Open 2013).

The Resourced Blueprint is obtained from the SDE. This operation effectively invokes the underlying CL-SOSM subsystem that is responsible for resource management, for available resources and resource definitions. The resulting Resourced Blueprint is completely supported by a CAMP-compliant CAMP Provider (Carlson et al. 2012).[1]

In the CL-Blueprint all references to CloudLightning-defined artefacts are removed, except for specific CloudLightning handles (opaque to the CAMP Provider). These handles are used for the creation of a session between the resource scheduling (self-organisation) layer and the deployed resources. This CL-Blueprint represents a fully qualified CAMP Document containing reference to real resource types, resource locations, and deployment mechanisms, which are fully understood and handled by a CAMP-compliant implementation.

4.5.1 Gateway Service Architecture

The CloudLightning Gateway Service builds upon the capabilities of the Apache Brooklyn solution, providing “service decomposition” capabilities. The Gateway Service completely reuses the rest of the features provided by Apache Brooklyn, facilitating the reuse of existing Blueprints and integration. Of particular interest is the integration with various Configuration Management Systems like Puppet, Chef, or Ansible (Fig. 4.6).

The Gateway Service has several roles, as follows:

1. Receive/create abstract[2] Blueprint definitions from EAO.
2. Decompose the received Abstract Blueprint into individual services. For each of the services check if it is a fully qualified service or has to be further processed. This operation further discussed in Sect. 4.5.2 (Service Decomposition).
3. Once the Blueprint is fully qualified (it does not contain any abstract service definitions), the Gateway Service triggers the services deployment and further execution.

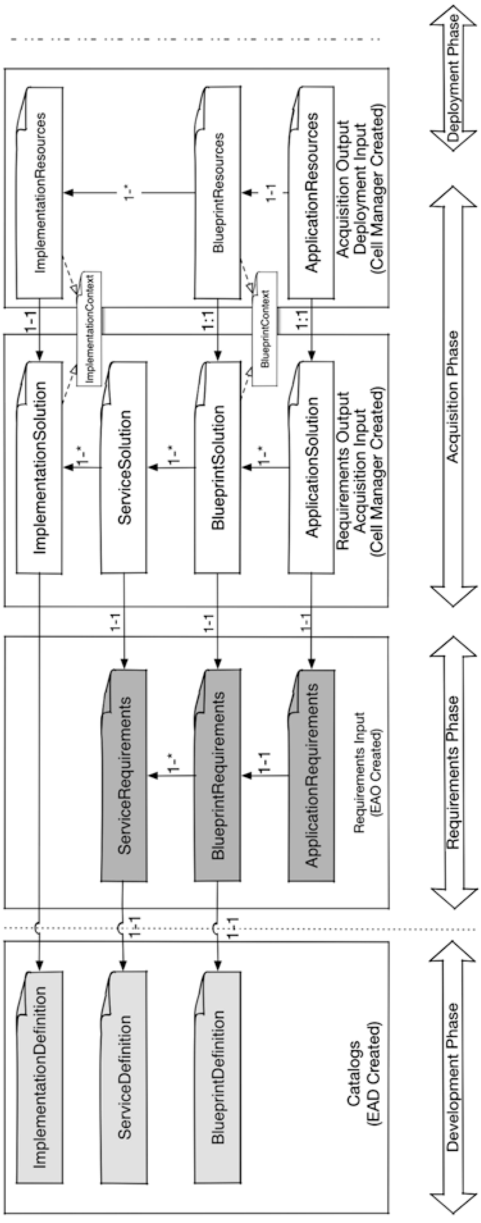


Fig. 4.5 API Message relationships

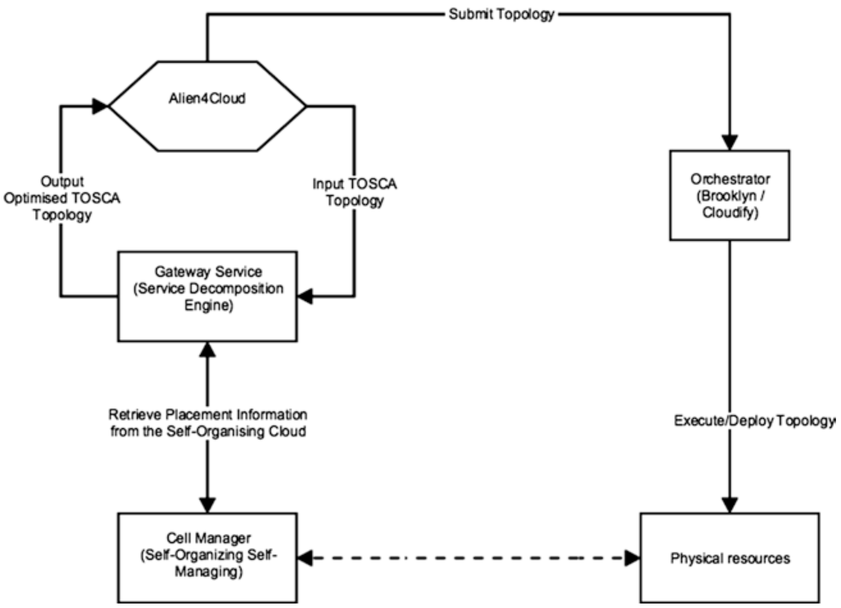

Submit Topology
Alien4Cloud
Output Optimised TOSCA Topology
Input TOSCA Topology
Orchestrator (Brooklyn / Cloudify)
Gateway Service (Service Decomposition Engine)
Retrieve Placement Information from the Self-Organising Cloud
Execute/Deploy Topology
Cell Manager (Self-Organizing Self-Managing)
Physical resources

Fig. 4.6 Gateway Service overall architecture

The Gateway Service exposes a series of APIs usable by consumers (EAOs and EADs) for controlling the application lifecycle.

4.5.2 Service Decomposition

The operation of Service Decomposition is implemented by the SDE and represents one of the core CloudLightning contributions in the Gateway Service. The SDE is responsible for the interaction with the SOSM subsystem. The overall operation of the SDE can be summarised as follows:

1. For each service, check if it can be instantiated directly (there exists a single implementation of the service, and that implementation is well known to the Gateway Service) or that it is an abstract service (a service interface that could be implemented by several implementations).
2. If the service is an abstract service the SDE contacts the backend SOSM system for selecting the proper implementations for the service.

3. In order to facilitate the selection of the proper implementation, the SDE transmits the user-provided requirements (in the form of ClassAd [Solomon 2003] definitions). These requirements are used by the SOSM subsystem for properly selecting the right implementations.
4. The selection of concrete implementations results in modifying the original Blueprint, by replacing the abstract definition with the resourced one (eventually after a user interaction for validating the right solution) and submitting the Blueprint to the next stage.

4.5.3 Interaction with the SOSM System

After the successful query of available implementations for each abstract service definition, the SDE component constructs a Resource Template containing information about the specific requirements of each implementation. An example of such Resource Template is given in Listing 4.1

Consider a Blueprint containing a single service in order to maintain better readability of the listing. Such a document contains a **blueprint ID** that is unique for each request, a **timestamp** representing the request time, a **cost** limit for the entire Blueprint, and the **callback endpoint** used by the SOSM system to communicate back results of the optimisation steps.

The sample service has two implementation options between which the SOSM will choose depending on their constraints and the overall cost of the blueprint. The first one refers to the need for a single VM with a single core (expressed by a computation range between 1 and 1), 1000 MB of memory, 50 GB of storage, bandwidth between 100 Mbps and 1 Gbps, and no accelerators.

The second implementation is of type **MIC-CONTAINER**, requiring the CellManager to find or create a container, which has access to an **MIC** accelerator. This service requires one container with one CPU core, memory between 100 and 1000 MB, storage between 10 and 50 GB, the same bandwidth as the other implementation, and one MIC accelerator.

4.5.3.1 Resource Discovery

The Gateway Service and the SOSM system exchange information for two operations: *resource discovery* and *resource release.*

- *Resource discovery* is the operation by which the SOSM system chooses the most suitable service implementation and the resources on which to deploy it, according to user constraints and system state.

```
"blueprintId": "{bpId}",
"timestamp": 1929292,
"cost": 0.0,
"callbackEndpoint": "http://10.0.0.1/sde/rest/blueprints/{bpId}",
"serviceElements": [
        {
                "serviceElementId": "service-elem-1",
                "implementations": [
                        {
                        "implementationType": "CPU-VM",
                        "requiredResourceUnit": 1,
                        "computationRange": [1, 1],
                        "memoryRange": [1000, 1000],
                        "storageRange": [50, 50],
                        "bandwidthRange": [100, 1000],
                        "acceleratorRange": [0, 0]
                        },

                        {
                        "implementationType": "MIC-CONTAINER",
                        "requiredResourceUnit": 1,
                        "computationRange": [1, 1],
                        "memoryRange": [100, 1000],
                        "storageRange": [10, 50],
                        "bandwidthRange": [100, 1000],
                        "acceleratorRange": [1, 1]
                        }
                ]
        },
        ...
        ]
}
```

Listing 4.1 Resource template

- *Resource release* is the operation by which the SOSM system is informed that the services have been terminated, so the underlying resources may be reallocated.

The aforementioned operations are modelled by Hypertext Transfer Protocol (HTTP) Representational State Transfer (REST) methods, both the *Cell Manager* and the SDE acting as REST servers.

Figure 4.7 describes the protocol for resource discovery and a POST request with the body containing a ResourceTemplate of the structure, as illustrated in Listing 4.1. If the *Cell Manager* encounters any problems during the parsing of the body, the status code of the response will be **409 Conflict**. Otherwise, the status code will be **201 Created** and the resource

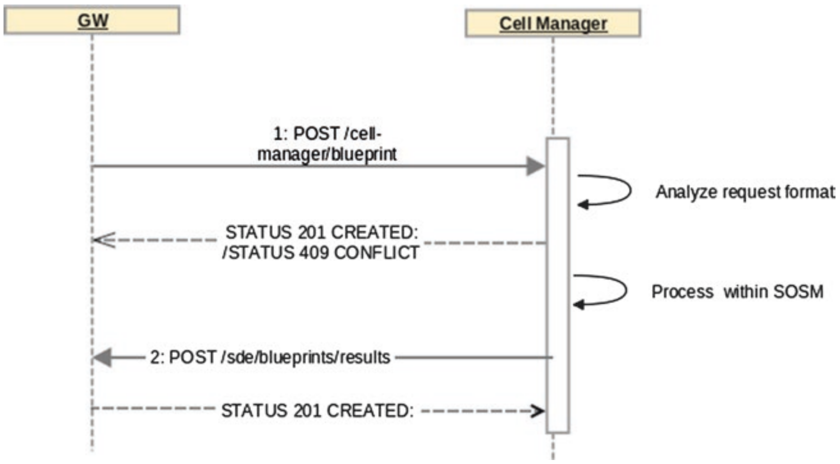

GW
Cell Manager
1: POST /cell-manager/blueprint
Analyze request format
STATUS 201 CREATED: /STATUS 409 CONFLICT
Process within SOSM
2: POST /sde/blueprints/results
STATUS 201 CREATED:

Fig. 4.7 Resource discovery sequence diagram

discovery process will start. The *Cell Manager* is in charge of informing the SDE when the result is ready.

When resources have been identified for all services, the *Cell Manager* will use a POST request with the body containing the information about the placement and implementation of each service, referred as a *Resourced Template*. This will trigger the SDE to instantiate each abstract service and update the Blueprint with concrete services and resource access information. An example result is shown in Listing 4.2. The chosen implementation is **CPU-VM**, and the resource type is **OPENSTACK ACCOUNT**, meaning that the SOSM is managing an OpenStack cluster as a resource. In this case, access information consists of credentials for accessing the OpenStack Nova API in order to create the VM.

4.5.3.2 Resource Release

The protocol for releasing the resources associated to a Blueprint is depicted in Fig. 4.8. A **DELETE** request is made to the *Cell Manager* at a path referencing the Blueprint ID. In case of successful resource release, the response will have the status **204 No Content**. Otherwise, the response will have status **400 Bad Request** and the body should provide useful information that will be propagated to the user interface (UI).

```
{
"blueprintId": "{bpId}",
"timestamp": 1929392,
"status": "SUCCESSFUL",
"resourcedServiceElements": [
        {
                "serviceElementId": "service-elem-1",
                "implementationType": "CPU-VM",
                "creatorId": "vrm-1",
                "status": "COMPLETED",
                "resourceType": "OPENSTACK_ACCOUNT",
                "resources": [
                {
                "resourceCreationId": "1234-5567-82929",
                "resourceDescriptor": "{\"platform\": \"OPENSTACK\",
                 \"domain\": \"SOSM\", \"project\": \"CL-SOSM\",
                 \"username\": \"cl-admin\", \"password\": \"s3cret\",
                \"authEndpoint\": \"http://10.0.2.19:5000/v3.0\"}"
                }
                ]
        },

  ]
}
```

Listing 4.2 Resourced template

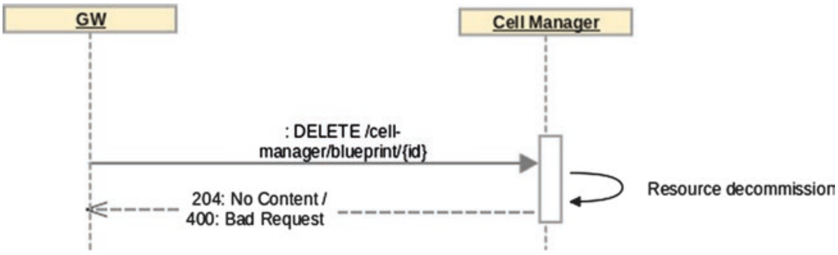

GW
Cell Manager
: DELETE /cell-manager/blueprint/{id}
204: No Content / 400: Bad Request
Resource decommission

Fig. 4.8 Resource release sequence diagram

4.6 The CloudLightning Blueprint Extensions

Below is a summary of the technologies upon which the CloudLightning Blueprints were developed.

4.6.1 CloudLightning Brooklyn Extensions

As part of CloudLightning project, Apache Brooklyn was adopted and extended as the underlying platform for achieving the project's ultimate goal of both supporting HPC applications and adoption of modern cloud technologies, thus creating a bridge between the HPC and Cloud end user communities.

The decision to use the Apache Brooklyn framework is motivated by the design decisions established in the conceptualisation of the CloudLightning architecture (Morrison et al. 2016), the CloudLightning protocol specification and APIs (Neagul et al. 2016), and the Gateway Service (Dragan et al. 2017).

The main advantages of using Apache Brooklyn include:

1. It provides the building blocks needed for developing the necessary functionality expected from the Gateway Service.
2. It offers support for "automatic blueprints" based on OASIS CAMP, an extensible specification that can serve as the core specification for the CloudLightning Blueprints.
3. The Apache Project plans to support TOSCA in the near future.[3] This could potentially allow further developments in the CloudLightning SDL, supporting the TOSCA standard (OASIS Open 2013).
4. The harnessing of existing Apache Blueprints, providing HPC vendors more choices without requiring more development effort.

The purpose of this section is to discuss how the adoption of the Brooklyn Blueprints, particularly the expected additions to the Blueprint YAML, is envisioned in CloudLightning. As previously noted, two different kinds of blueprints are identified for use in CloudLightning: Abstract Blueprints and Concrete Blueprints (referred further as "blueprints"). Both types of Blueprints are built on top of Apache Brooklyn blueprints.

The translation between the Abstract Blueprint and Runnable Blueprints is performed by means of a specialised component residing inside the Gateway Service, component named "Service Decomposition Engine." The decomposition engine is responsible for interacting with the SOSM infrastructure (Fig. 4.9).

Each of the two types of Blueprints is discussed in the following sections, outlining the changes to the vanilla (plain) Brooklyn Blueprints. Note that the proposed extensions are subject to change as other parts of the CloudLightning Project evolve and might also be influenced by outside changes in the Apache Brooklyn project, as, for example, the addition of new functionality or deprecation of a current one.

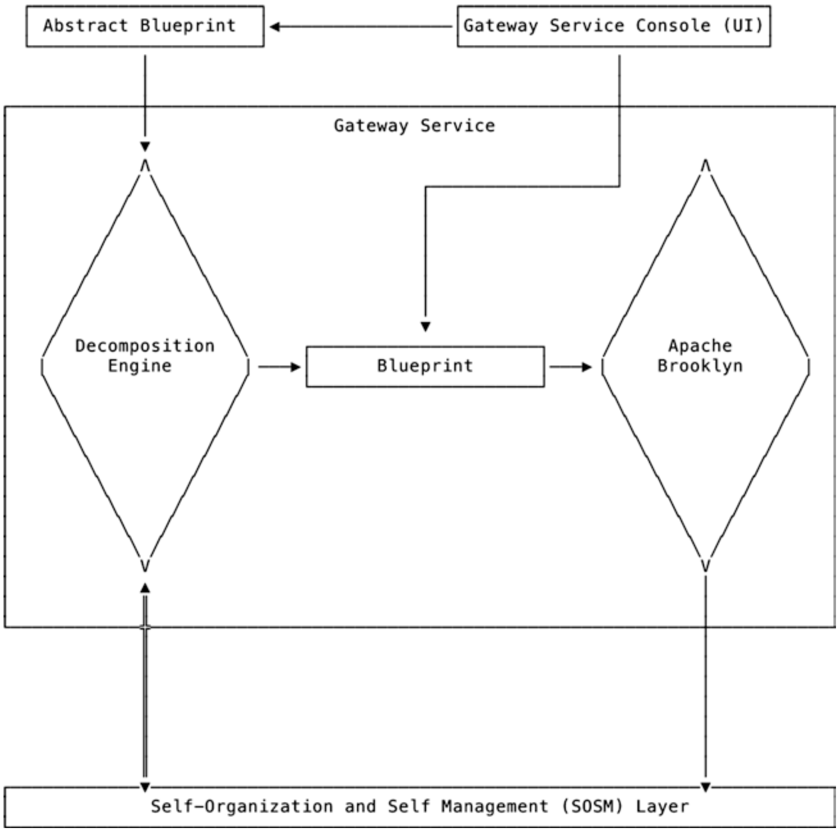

Abstract Blueprint
Gateway Service Console (UI)
Gateway Service
Decomposition Engine
Blueprint
Apache Brooklyn
Self-Organization and Self Management (SOSM) Layer

Fig. 4.9 CloudLightning Blueprint decomposition process

4.6.2 CloudLightning Abstract Blueprint

The Abstract Blueprint is represented by an extended version of the Apache Brooklyn Blueprint, containing attributes holding CloudLightning-specific entries, as described in Listing 4.3.

In this example, the Abstract Blueprint requires the deployment of a Java web application and a computing resource providing raytracing capabilities. Of interest in this case is the abstract computing service identified by the name “RaytracingApplicationId”: the service cannot be directly handled by the Apache Brooklyn framework as it does not

```
id: jetty-node-with-raytracing-compute
name: "Jetty␣Application␣With␣Raytracing␣Computing␣Resource"
origin: http://cloudlightning.io/
locations:
- cloudlightning-openstack
services:
- type: cloudlightning.entity.meta.RaytracingApp
name: RaytracingApplicationId
location: cloudlightning-openstack
cloudlightning.config:
service-requirements:
- type: classad
requirement: 'Arch=="Intel"␣&&␣CoProcesor=="IntelPhi"'
rank: 'TARGET.Mips'
- type: brooklyn.entity.webapp.ControlledDynamicWebAppCluster
name: webApp
location: cloudlightning-openstack
cloudlightning.config:
service-requirements:
- type: classad
requirement: 'Arch=="Intel"'
brooklyn.config:
wars.root: http://example.cloudlightning.io/webapp/webapp.war
http.port: 9280+
proxy.http.port: 9210+
java.sysprops:
cloudlightning.example.ray.url:
          $brooklyn:formatString("drmaa://%s",
                    component("RaytracingApplicationId").
                    attributeWhenReady("drmaa.url"))
```

Listing 4.3 An Abstract Blueprint

provide the required information (the **cloudlightning.entity.meta.RaytracingApp** type is not known to Brooklyn).

This service is handled by the CloudLightning SDE by interpreting the provided application information (in this case, the type) and the corresponding matching information. The information needed for the normal SDE operation is defined at the service level, under the **cloudlightning.config** attribute.

The relevant attributes handled by the SDE at the "service-requirements" level are:

- *Type*: this field defines the syntax used for expressing this requirement. Currently the only defined syntax is based on the ClassAds system[4].

- *Requirements*: this field defines the expression interpreted by the SOSM system to identify the appropriate resource required for this service.
- *Rank*: this field defines the way of ranking the possible solutions obtained from the underlying SOSM infrastructure; this expression might be used to prefer resources by various attributes, eventually based on power consumption or computing power.

The "requirements" attribute is aimed at restricting the resources that the SOSM subsystem can consider for choosing the proper implementation for the user-requested service. This attribute is expected to be used by HPC application to express their performance requirements, and it is complemented by the "rank" attribute, used for expressing preference regarding the available and matching resources.

4.6.3 CloudLightning Blueprint

The CloudLightning Blueprint represents the outcome of the Service Decomposition Operation and basically represents a fully qualified Blueprint document that can be handled by the CAMP framework (in our case, Brooklyn).

As seen in Listing 4.4, all "abstract" specifications have been replaced with concrete ones. For example, the **cloudlightning.entity.meta.RaytracingApp** type has been replaced with another type understood by Brooklyn (**cloudlightning.entity.impl.HPCCluster**). This new type is complemented by a new set of attributes that provide deployment-specific information.

It is important to note that the "location" attribute has been customised to provide CloudLightning-specific information; particularly in this case, it contains a handle provided by the underlying SOSM subsystem that can be used at deployment time for synchronising information between the various subsystems. Notice that the **cloudlightning.entity.impl.HPCCluster** is known to Brooklyn due to the fact that it is registered by the EAO in the corresponding catalogue.

4.7 Example of Application Creation and Deployment

The architecture of the CloudLightning Gateway Service was presented previously in Sect. 4.5. This section demonstrates, using an example of a raytracing application, the ease with which the application topology can be created and deployed using the CloudLightning Gateway Service. This

```
id: jetty-node-with-raytracing-compute
name: "Web␣Application␣With␣Raytracing␣Computing␣Resource"
origin: http://cloudlightning.io/
locations:
- cloudlightning-openstack
services:
- type: cloudlightning.entity.impl.HPCCluster
name: RaytracingApplicationId
location:
cloudlightning-openstack:
session:handle: "b4cfc054-b760-4d82-a2ce-96a65b3d72d0"
brooklyn.config:
cloudlightning.deployment:
puppet.manifests.location: "http://p.cloudlightning.io/m/intelphicluster"
- type: brooklyn.entity.webapp.ControlledDynamicWebAppCluster
name: webApp
location:
cloudlightning-openstack:
session:handle: "26670286-a0ad-499e-9fef-f665d156e27e"
brooklyn.config:
wars.root: http://example.cloudlightning.io/
        webapp/webapp.war
http.port: 9280+
proxy.http.port: 9210+
java.sysprops:
cloudlightning.example.ray.url:
        $brooklyn:formatString("drmaa://%s",
                component("RaytracingApplicationId").
                attributeWhenReady("drmaa.url"))
```

Listing 4.4 The CloudLightning Blueprint

use case is used to illustrate a user's interactions with the Gateway Service, enhancing the resource optimisation feature. The remainder of this section provides a brief overview of the steps to be taken to safely create, optimise, and deploy the raytracing application on the CloudLightning environment. Some of the essential steps are also depicted in screenshots taken from the actual system.

The process is as follows:

Step 1: To initialise the system, start Alien4Cloud service.

Step 2: Add the plugin to the desired orchestrator (CloudLightning uses Brooklyn-TOSCA as the underlying orchestrator). After the plugin is loaded, Alien4Cloud will present the orchestrator in the list of available plugins.

Step 3: Create a new orchestrator from the UI and link it to the newly added plugin.

Step 5: Before one can connect the orchestrator instance from Alien4Cloud to the underlying orchestrator (basically, the SOSM subsystem), one has to ensure that the Gateway Service Orchestrator is running. This step is not a mandatory step to be taken but it is advised.

Step 6: From the web console one can connect to the bespoke orchestrator. Before any further steps can be taken, wait until the orchestrator state is CONNECTED.

Step 7: After the orchestrator is connected, download the CSAR archive from a remote *git* repository.[5]

The orchestrator comes with *git* integration functionalities, and the only requirement is to have stored all custom CSAR files in such a repository. In case of the raytracing example, one has to enter the predefined *git* credentials and URL. The download process of the CSAR archive starts only after one clicks the Import button.

Step 8: Add the CloudLightning plugin to have access to the CloudLightning functionalities.

Step 9: For the creation of new applications one has to use the functionalities exposed by Alien4Cloud, more precisely the New Application panel. The CSAR archive may contain already defined application templates, and one can select some of those for the intended application design.

Step 10: As soon as the application creation step is finished, one can view the design and application in its home panel.

Step 11: The previously defined topology contains four types of nodes, which can be viewed in the Topology tab (see Fig. 4.10). It is also possible to view the newly created topology in YAML format by pressing the YAML tab in the designer.

Step 12: Next, enter the CloudLightning Optimisation Panel and start the optimisation process from the SOSM Optimiser button (see Fig. 4.11). On the left-hand side, one can view the endpoint for the SDE.

Step 13: Check that the SDE is up and running, and when the optimisation process is finished, one can notice that the abstract nodes have been replaced with concrete ones also in the application designer.

Step 14: As a final step prepare for the deployment of application by entering into the Deployment Panel. The orchestrator has already sent information about locations to Alien4Cloud and one has only to select the desired location.

Step 15: By moving to Deploy tab one can trigger the actual deployment of the application. This step is performed by pressing the Deploy button and wait until it finishes. Once pressed one can follow the explicit progress of the deployment also in the orchestrator console.

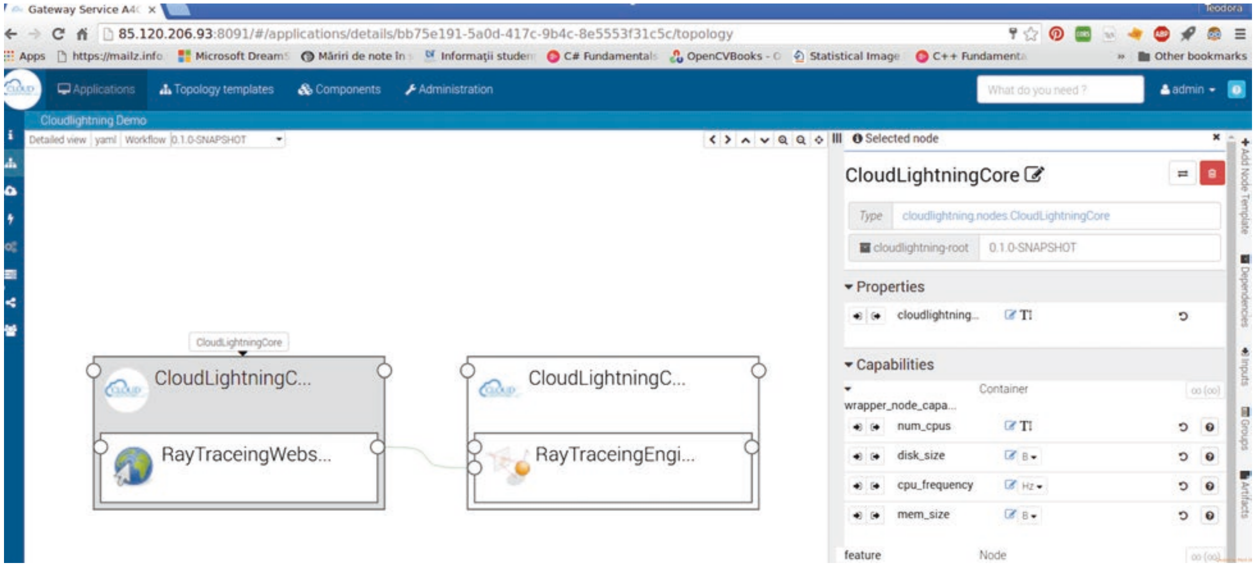
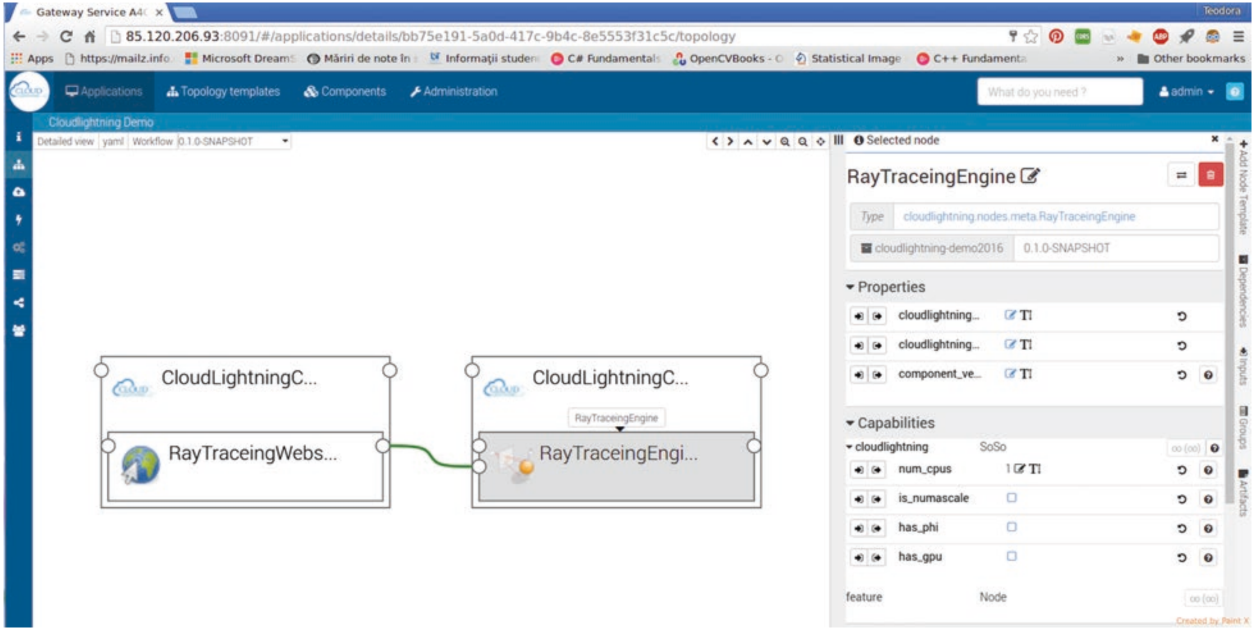


Fig. 4.10 Application topology: CloudLightning Core 1 node



RayTraceingEngine

Type	cloudlightning.nodes.meta.RayTraceingEngine
cloudlightning-demo2016	0.1.0-SNAPSHOT

Properties

cloudlightning...
cloudlightning...
component_ve...

Capabilities

cloudlightning	SoSo	∞ (∞)
num_cpus	1	
is_numascale	☐	
has_phi	☐	
has_gpu	☐	
feature	Node	∞ (∞)

Fig. 4.11 Start of the optimisation process

4.8 Conclusion

This chapter presented the CloudLightning Gateway Service, a user-friendly interface that enables users to create and deploy applications with minimum knowledge regarding the resource selection process. The Gateway Service is a key component of the CloudLightning system that facilitates application lifecycle management in the context of a cloud environment. Users can design the application topology using the *Drag & Drop* mechanism of the Gateway Service UI and link together the components of their application. From here, the topology is sent to the *SDE*, which is responsible for interacting with the *SOSM* system. The SDE translates the information from the application topology, into a specific CloudLightning Blueprint, using the *CloudLightning Service Description Language*. Next, *SOSM* handles the resource discovery process, assigning the most suitable set of resources for a user application, based on the received CloudLightning blueprint. In the following step, the *SOSM* sends back to the *SDE* a CloudLightning blueprint, with a proposed resource for each component of the application topology. In the end, the user may review the final version of its application topology, with the assigned resources, and start the process of application deployment.

4.9 Chapter 4 Related CloudLightning Readings

1. Dragan, I., Fortis, T. F., & Neagul, M. (2016). Exposing HPC services in the cloud: The CloudLightning approach. *Scalable Computing: Practice and Experience, 17*(4), 323–330.
2. Selea, T., Dragan, I., & Fortiş, T. F. (2017, April). The CloudLightning approach to cloud-user interaction. In *Proceedings of the 1st International Workshop on Next generation of Cloud Architectures*, Vol. 4, ACM.

Notes

1. The term CAMP provider is used in the sense as defined by the CAMP specification, basically "an implementation of the service aspects of this specification."
2. Abstract Blueprints are those blueprints that will be later on filled with concrete resources by the CL-System.
3. https://brooklyn.apache.org/learnmore/theory.html
4. https://research.cs.wisc.edu/htcondor/classad/classad.html
5. One keeps definitions of services in CSAR format in a remote repository.

References

Apache Software Foundation. (n.d.). Apache Brooklyn. Retrieved October 15, 2017, from https://brooklyn.apache.org/

Apache Software Foundation. (n.d.). Apache jClouds®. Retrieved October 15, 2017, from https://jclouds.apache.org/start

Breiter, G., Leymann, F., & Spatzier, T. (2012, May). *Topology and orchestration specification for cloud applications (TOSCA): Cloud service archive (CSAR)*. International Business Machines Corporation.

Carlson, M., Chapman, M., Heneveld, A., Hinkelman, S., Johnston-Watt, D., Karmarkar, A., et al. (2012). OASIS, Tech. Rep. *Cloud Application Management for Platforms*. Retrieved October 10, 2017, from http://cloudspecs.org/CAMP/CAMP_v1-0.pdf

Dragan, I., Selea, T., & Fortis, T.-F. (2017). *D5.2.1 Gateway Service*. CloudLightning consortium. Retrieved October 15, 2017, from.

Morrison, J., Xiong, H., Dong, D., & Momani, B. (2016). *D3.1.2 Architecture*. CloudLightning Consortium. Retrieved October 15, 2017, from.

Neagul, M., Dragan, I., & Craciun, C. (2016). *D4.1.1 protocol specification and API*. CloudLightning Consortium. Retrieved October 15, 2017, from.

OASIS Open. (2013). *Topology and orchestration specification for cloud applications version 1.0*. Retrieved October 10, 2017, from http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html

Openstack.org. (2017). OpenStack Mistral. Retrieved October 18, 2017, from https://docs.openstack.org/mistral/latest/

Openstack.org. OpenStack Solum. Retrieved October 18, 2017, from https://docs.openstack.org/solum/latest/

Openstack.org. Heat—OpenStack. Retrieved October 15, 2017, from https://docs.openstack.org/heat/pike/

Solomon, M. (2003). *The ClassAd Language Reference Manual, Version 2.1*. Computer Sciences Department, University of Wisconsin, Madison, WI, USA.

Sun, L., Dong, H., & Ashraf, J. (2012, October). Survey of service description languages and their issues in cloud computing. In *Eighth International Conference on Semantics, Knowledge and Grids (SKG)* (pp. 128–135). IEEE.

Xiong, H., Dong, D., Morrison, J., Antoniadis, I., Neagul, M., Giannoutakis, K., et al. (2016). *D5.1.1 service description format*. CloudLightning Consortium. Retrieved 15 October, 2017, from https://cloudlightning.eu/blog/service-description-format/d5-1-1-service-description-format-3/