Cloud Orchestration at the Level of Application

Project Acronym: **COLA**

Project Number: **731574**

Programme: **Information and Communication Technologies**
Advanced Computing and Cloud Computing

Topic: **ICT-06-2016 Cloud Computing**

Call Identifier: **H2020-ICT-2016-1**
Funding Scheme: **Innovation Action**

Start date of project: 01/01/2017 Duration: 30 months

Deliverable:
D5.2 COLA Application Templates

Due date of deliverable: 31/07/2017 Actual submission date: 26/07/2017

WPL: Gabriele Pierantoni

Dissemination Level: PU

Version: WIP

1. Table of Contents

Contents

2. List of Figures and Tables

Figures

Tables

3. Status, Change History and Glossary

Status:	Name:	Date:	Signature:
Draft:	Gabriele Pierantoni	20/07/17	Gabriele Pierantoni
Reviewed:	Jose Manual Rapun	24/07/17	Jose Manual Rapun
Approved:	Gabor Terstyanszky	26/07/17	Gabor Terstyanszky

Table 1, Status Change History

Version	Date	Pages	Author	Modification
V0.1	28/05	ALL	G. Pierantoni	First Policy Draft
V0.2	01/06	ALL	G. Pierantoni	Added relationship with other WPs and Some Policy Examples
V1.0	20/07	ALL	G. Pierantoni, G. Terstyanszky	Added Sections on multi-level application description and other corrections.
V1.1	24/07	ALL	G. Pierantoni, G. Terstyanszky	Addressed Internal and External Review Comments
V1.2	26/07	ALL	G. Pierantoni	Updated policy template and policy-related annexes.
V1.3	26/07	ALL	G. Pierantoni	Final Corrections and Formatting
final	26/07	ALL	G. Pierantoni	Final version

Table 2, Deliverable Change History

Glossary

API	Application Programming Interface
CAMP	Cloud Application Management for Platforms
COLA	Cloud Orchestration at the level of Application
CLI	Command Line Interface
DoW	Description of Work
GUI	Graphical User Interface
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
SaaS	Software as a Service
TOSCA	Topology Orchestration Specification for Cloud Application

Table 3, Glossary

4. Introduction

COLA DoW specifies the **D5.2 "*Specification of the Application Template Concept*"** as follows:

> "*This deliverable will outline the concept of the application template and its formal description*."

This deliverable aims at illustrating how COLA will describe the applications and how these description referred to in this document as Application Templates will be handled by the various components of the COLA architecture. This deliverable is related to other COLA Deliverables and work packages and such dependencies are detailed in section TODO. The Research Activities in this area are correlated to other Work packages (as detailed in Section 5) and is still ongoing; the proposed approach described may (and probably will) be amended and ameliorated depending on the results of the adoption of his design and guidelines

D5.2, is structured as follows:

- Section 5, defines the dependencies of this deliverable with other deliverables of COLA.
- Section 6, offers an overview of Application Descriptions and COLA design guidelines.
- Section 7, defines how COLA will describe the Application Topologies.
- Section 8, details the relationship of TOSCA with virtualization.
- Section 9, introduces the description of policies.
- Section 10, defines the assumption and limitations of the design for the policies description, defines the hierarchy for the definition of the policy types, defines the relationship between the policy description and the application template structure and defines the structure for the policy template.
- Section 11, concludes the Deliverable and lists some open issue that will be further investigated.
- Section 12 contains the references.

5. Relationship with other Work Packages and Deliverables

This deliverable is closely related to the following Work Packages and Deliverables:

WP4 – Cloud Access Layer. The Policy Extension uses information that originates from the Cloud Access Layer (e.g. the geographical location of the services). The policy extension may also dictate some characteristics of the IaaS Layer (e.g. the storage encryption or networking constraints), finally, the Microservices Performance benchmarking may be used directly in the Trigger Namespace; alternatively, another component will collect these measures and present them in another namespace.

WP5 – Application Definition Templates. Deliverable **D5.1** – "Analysis of existing application description approaches", analyses the existing applications descriptions. This state of the art is related to this deliverable as it defines TOSCA as description language and overall design approach is best suited to solve COLA's requirements.

WP6 – Microservices deployment and execution. The COLA policy extensions will be used by the Cloud Deployment Orchestrator Service (**T6.1**) through the TOSCA Translator. The policies will use information from the Measurements and Metrics (**T6.2**). Finally, the policies will be used by the MiCADO Policy Keeper (**T6.3**) and MiCADO Policy Optimiser (**T6.4**) to define high-level optimization policies. (See Fig. 2 Architecture of the MiCADO Cloud Orchestration Layer in D6.1.)

WP7 – Security, Privacy and Trust. The COLA policy extensions will be compatible with the design drafted in **T7.1** and will be related to the COLA Security Architecture Design. More specifically, the TOSCA policy extensions will be able to support and express the security policies to be defined in **D7.2**.

WP8 – Use Case Pilots. The TOSCA policy extensions will be able to support the use cases defined in **D8.1**

6. Application Descriptions

The description of the applications in COLA covers various interconnected aspects and it is fundamental for the good success of the project.

- Firstly, as it is common in related scientific and industrial research, we assume that each application will be composed by different services that are arranged in an “application topology”.
- Secondly, we assume that the application description must provide information on how each service is deployed, undeployed and run
- Thirdly, we assume that aspects on how services are deployed, undeployed and run are described by policies.

In addition to these design requirements, we also define a set of design guidelines that inspire the design in this document. These guidelines are either derived from COLA’s DoW or have been defined to tackle specific aspects of the project implementation.

- The application description must be compatible and interoperable as much as possible with current industrial and scientific standards and practices.
- The application description must minimize the effort required to application developers, by maximizing re-usability.
- The application description must contain the definition of policies that regulate the deployment, execution and un-deployment of services.
- The application description must support an incremental development of the COLA architecture.

Deliverable D5.1, has provided an overview of the current status of Application Description approaches both in the academic and industrial environment. Given various considerations and the main COLA requirements, TOSCA [1] was selected as the most suitable solution for the Application Description. A State of the Art and the parameters that led to the decision to adopt TOSCA for the Application Description in COLA are detailed in Deliverable 5.1.

The concepts of **Applications, Services, Types and Templates** are important to this document and are detailed in Figure 1.

On the horizontal axis, we have Types and Templates. Types are akin to Object Oriented Hierarchies and define the structure of an application description while templates define the values defined within the type.

Within Types and Templates, we have Applications and Services: Applications are composed by set of Services along with the full software stacks that allow their deployment and executions. Applications are described by graphs that are also referred to as “Services Topologies”.

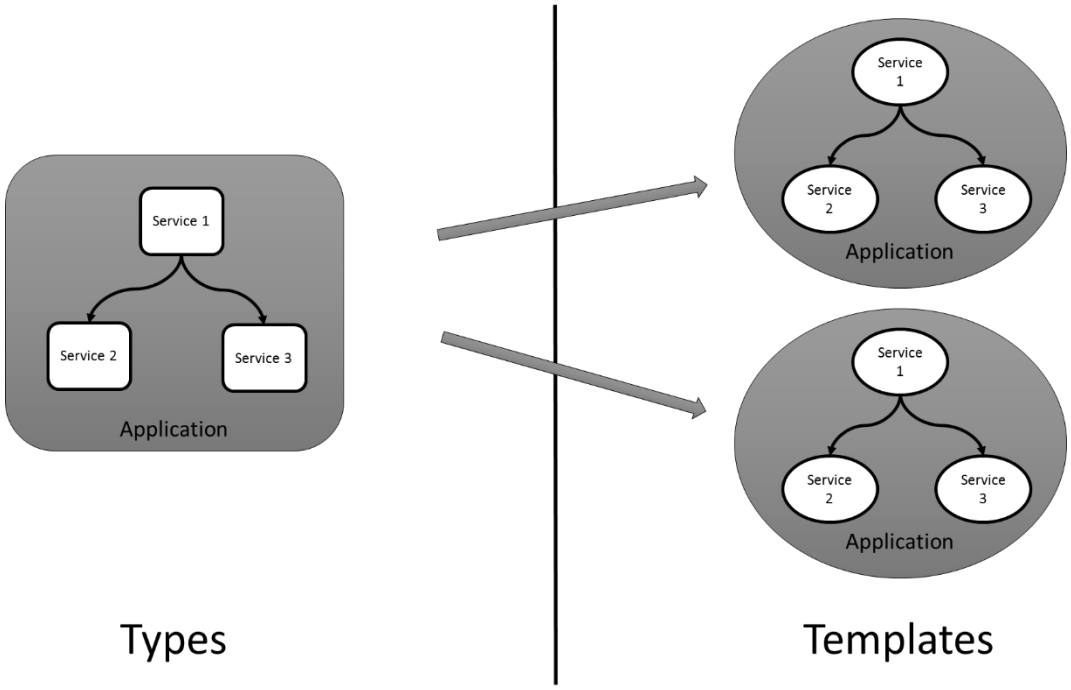

Figure 1, Applications, Services, Types and Templates

This Deliverable describes how COLA will use the TOSCA language specification[1][2] to implement a multi-layered application description and how to define policies that will define the modalities with which the lifetime cycle of the application will be managed.

7. COLA applications and their TOSCA based description

Typical COLA applications are web applications that contain three layers: proxy, application and database layer. These layers may run on a single or multiple servers considering user demands. Figure 2 presents the application architecture and its typical implementation running on multiple servers.

Figure 2, COLA application architecture

COLA describes applications using the TOSCA meta-model, presented in Figure 3. TOSCA defines generalized and base node types such as application server, web server, database, compute node, storage node etc. to describe PaaS and IaaS resources and services. WP5 will use these node types to describe the platform on which they run and resources they use.

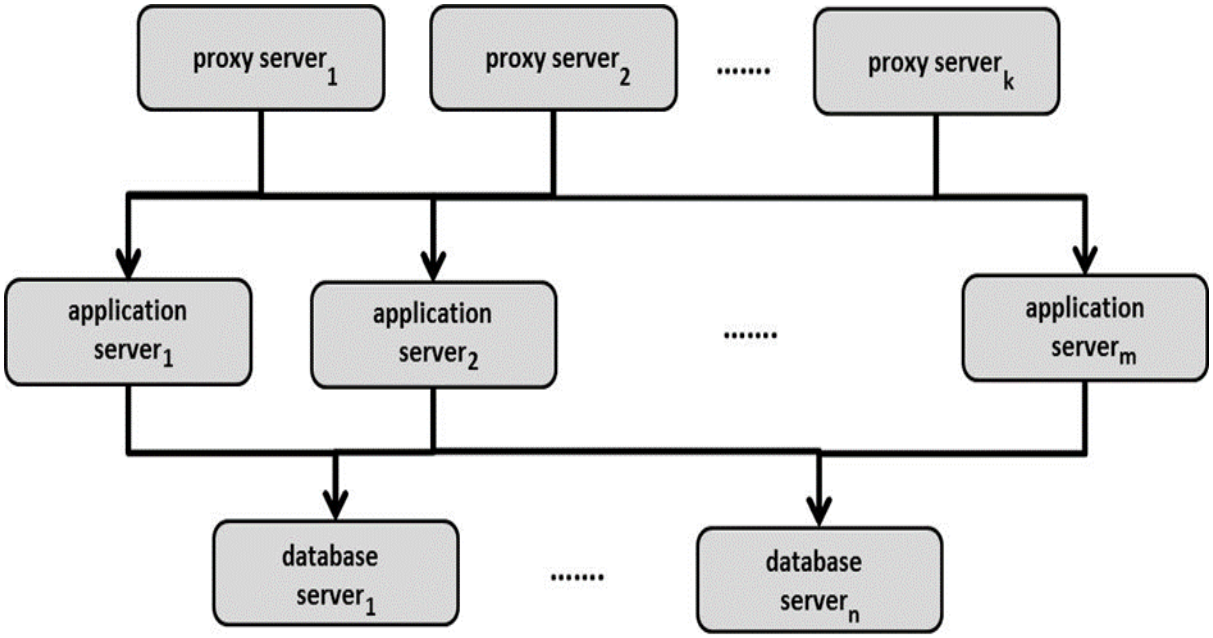

Figure 3, Implementation of the COLA application architecture

COLA applications using the TOSCA meta-model, (Figure 4) define generalized and base node types such as application server, web server, database, compute node, storage node etc. to describe PaaS and IaaS resources and services. WP5 will use these node types to describe the platform on which they run and resources they use.

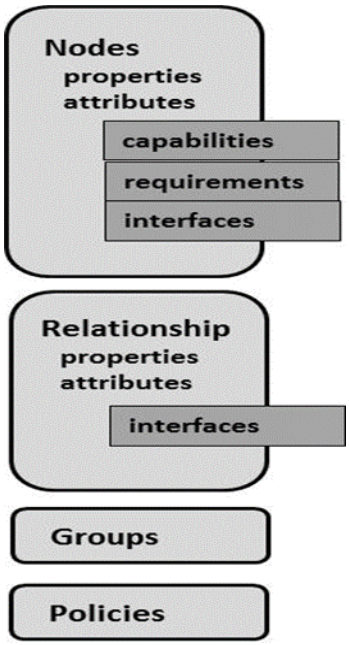

Figure 4, TOSCA meta-model

In COLA cloud applications, three layers can will be distinguished: application, platform, and resource layer (Figure 5).

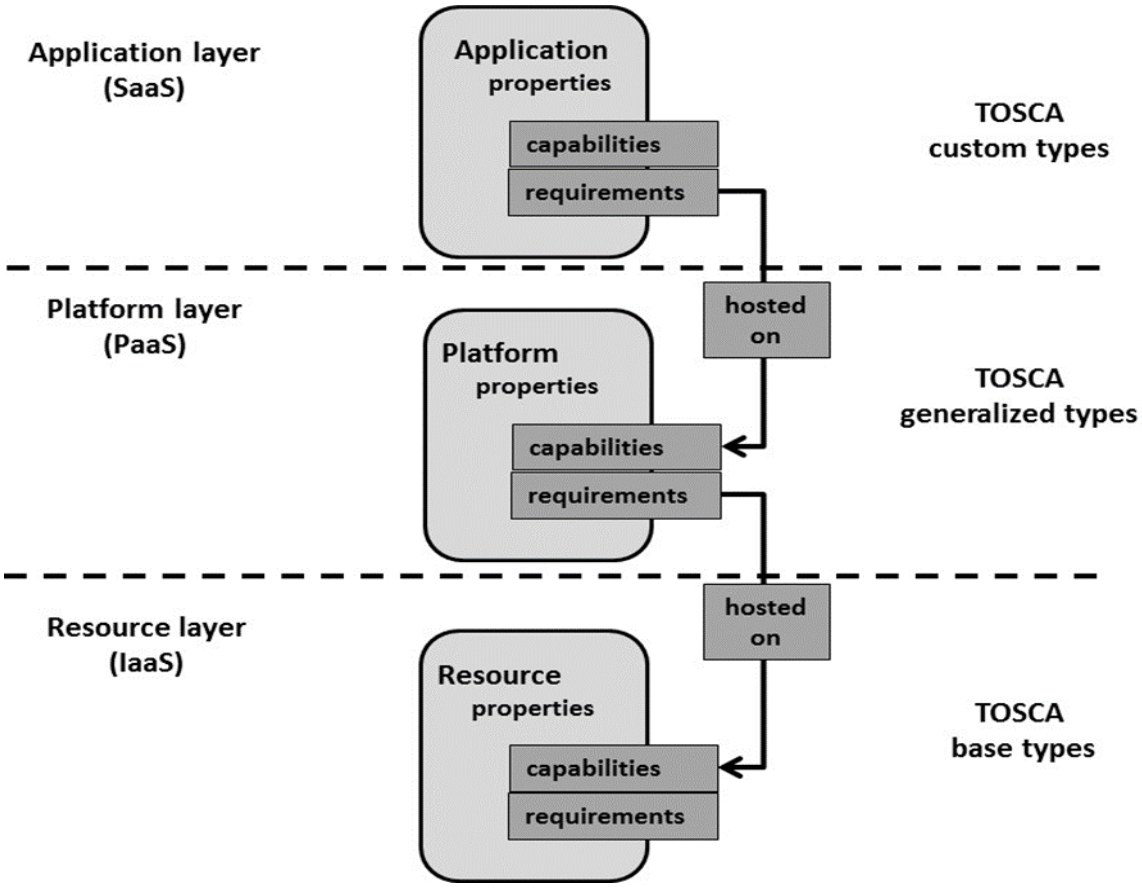

Figure 5, COLA application layers and their representation in TOSCA

The **application layer** contains applications such as proxy services, databases (for example MongoDB, MySQL, etc.), web applications (for example COLA user applications). They are managed as SaaS and executed on PaaSs and IaaSs. This layer defines the application architecture as a service template. This template describes the structure, including services as components and their relation, and a basic set of requirements of one particular application type. Applications can be described by custom node types to support their re-use in other TOSCA applications. These custom node types will be derived from TOSCA Container, SoftwareComponent and WebApplication generalized node type. The **platform layer** provides platforms as containers, for example proxy server, application server, web server, database server, etc., to host and run applications. The servers are described as generalized node types. For example BlockStorage, Container, Database, WebServer are generalized node types. The **resource layer** consists of compute, network and storage resources available in the Cloud. The resources are represented in TOSCA as base node types, such as Compute, Network and Storage base node type. Fig. 5 depicts the COLA applications layers and their representation in TOSCA.

The application architecture, depicted in Figure 2, can be specified in TOSCA using this three layer approach. This specification is presented in Figure 6.

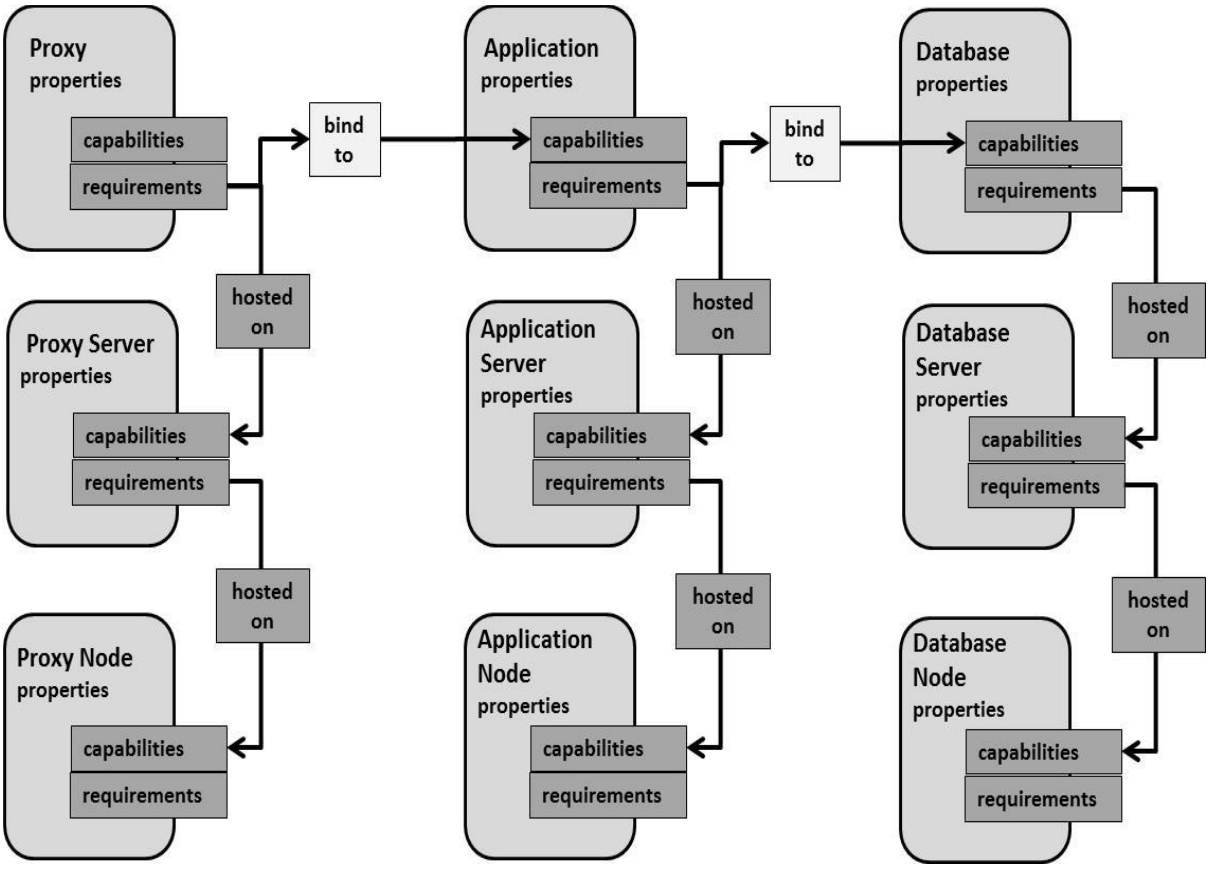

Figure 6, TOSCA specification of the COLA application architecture

8. TOSCA and virtualisation

As shown in Figure 7, COLA applications can run either on Linux or on Windows operating system. Windows applications could be too heavy for Docker containers. As a result, the MiCADO platform has to support running Linux based applications in containers and Windows based application in virtual machines

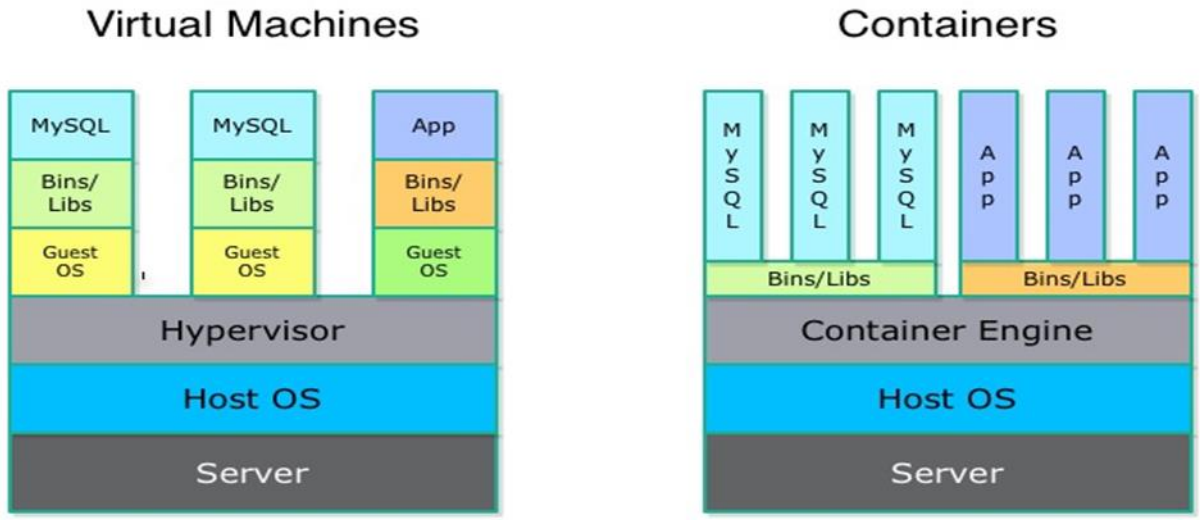

Figure 7, Containers and Virtual machines

Virtual machine (**VM**) emulates system providing functionality of a physical computer. Their implementations may involve specialized hardware, software, or a combination. VM requires some sort of emulation layer or hypervisor complete with an OS installation for each VM. The end user has the same experience on a virtual machine as they would have on dedicated hardware. The hypervisor emulates hardware resources, such as client, or server's CPU, memory, hard disk, network and other enabling virtual machines to share the resources. The hypervisor can emulate multiple virtual hardware platforms that are isolated from each other, allowing virtual machines to run Linux and Windows operating systems on the same physical host. Virtualization limits costs by reducing the need for physical hardware systems. Virtual machines more efficiently use hardware, which lowers the quantities of hardware and associated maintenance costs, and reduces power and cooling demand.

Container is an operating-system level virtualization method that provides a completely isolated environment, simulating a closed system running on a single host. It gives the user the ability to have an environment to do whatever is needed; in particular to develop and run applications with the necessary resources and environment configuration. Container uses features in the host operating system's kernel to provide an isolated virtual environment– disk, memory, networking, etc., on the same OS. Container does not require an install of anything other than the files required for applications. The container is a bit like virtual machines, but less demanding over the host computer. Unlike a virtual machine, where an application and the OS are tied together, a container splits these by making the OS a shared asset among containers. C**ontainers are isolated.** They may share the virtual machine with other containers, but they may never know about each other.

D6.2 identified **Docker** as a container technology (Figure 8) to be used to run COLA applications. Docker wraps up a software in a complete file system that contains everything it needs to run: code, run-time, system tools, and system libraries. Docker enables deploying

applications to a **container.**

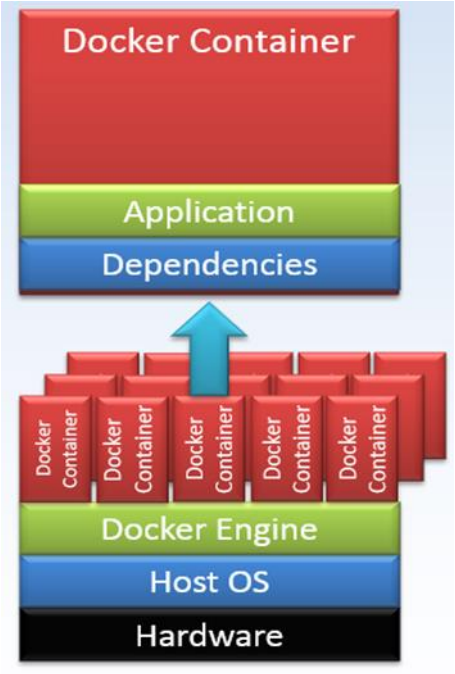

Figure 8, Docker Containers[1]

Docker has the following major components (Figure 9):

- **Docker images** - contain all the dependencies of applications. You using a file called
- **Docker file** - contains series of commands that specifies how to build an image.
- **Docker containers** - are instances of images holding everything that is needed for an application to run. Each container can be run, started, stopped, moved and deleted.
- **Docker registries -** repositories that hold base images, and they can be public (Docker Hub) or private.

To run an application inside a Docker container the following steps must be completed:

1. Create a **Docker file** that describes what application needs to run (i.e., how to define an image).
2. Connect to the **Docker Host**.
3. Using the Docker file, create an **image** in the Host.
4. Create a new **container** using the image.
5. Start the container with **"Dockerized"** application**.**
6. Take a snapshot of this container. This will create a new image which can be uploaded to a **Docker Registry**.

[1] A. Farkas, MTA SZTAKI, Docker Clustering Tools Comparison

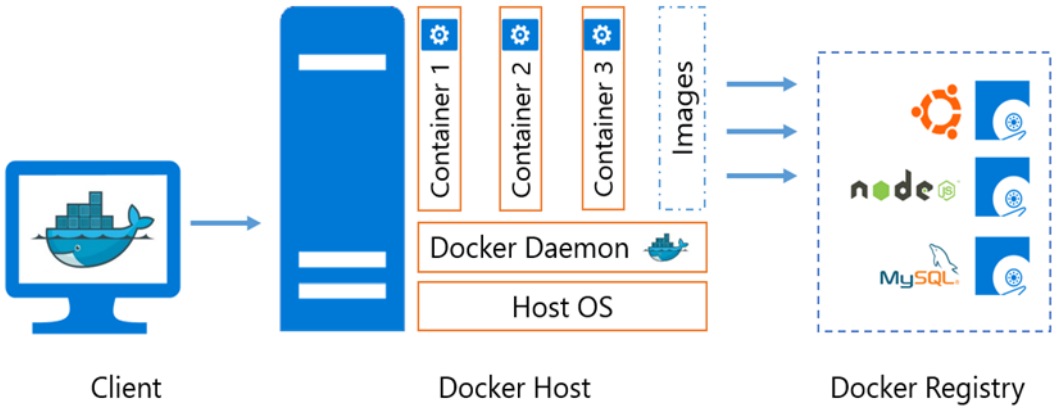

Figure 9, Docker environment[2]

COLA will use Docker Swarm in the MiCADO platform to manage Docker containers in the Cloud.

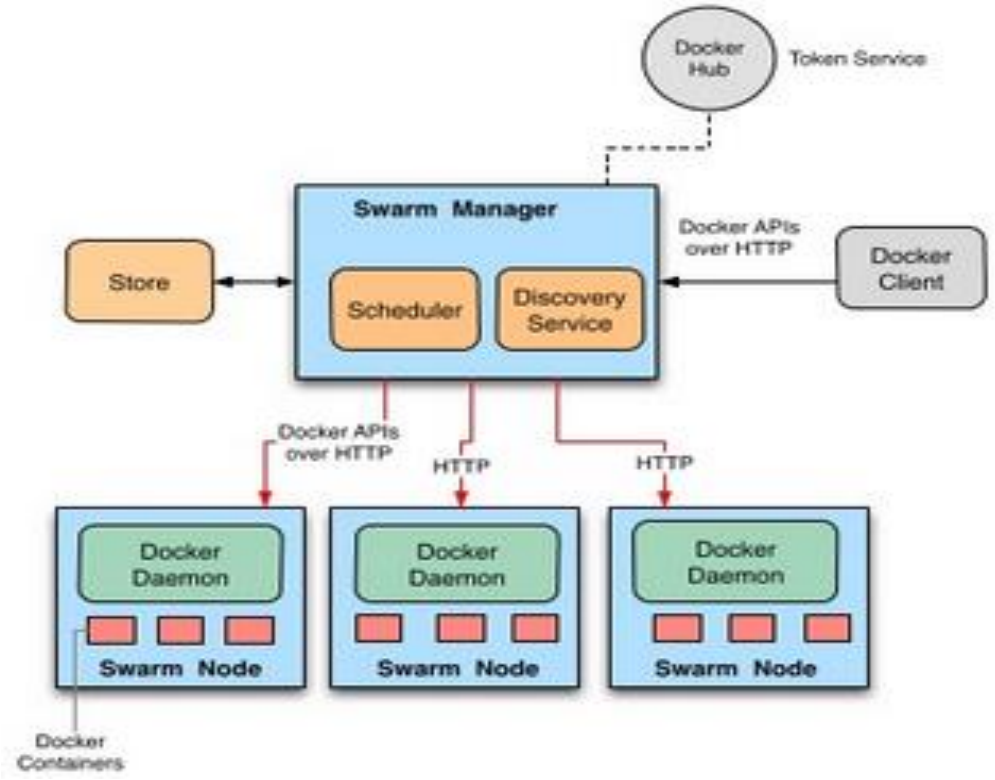

Figure 10, Docker Swarm architecture

Figure 10 presents the architecture of Docker Swarm used to manage containerized applications in the Cloud. Docker Swarm is an orchestration tool providing clustering and scheduling capabilities for applications. Docker Engines are clustered into a single "virtual engine" that pools their resources together and communicate with a single Swarm master to execute commands. It offers flexible scheduling policies to improve management of host resources available to run applications.

Several cloud orchestration solutions, for example Cloudify[3], Mesos[4], OpenTosca[5][6][7],

[2] Diagram source - http://southworks.com/blog/2015/07/03/introduction-to-docker/

etc. defined a new generalized node type called “DockerEngine” to create and run Docker[8] containers in the Cloud. This node type provides PaaS and IaaS support to run Docker containers in the Cloud. Using OpenTosca Winery[9], WP5 developed the TOSCA descriptions, presented in Figure 11, of the application architecture given in Figure 2.

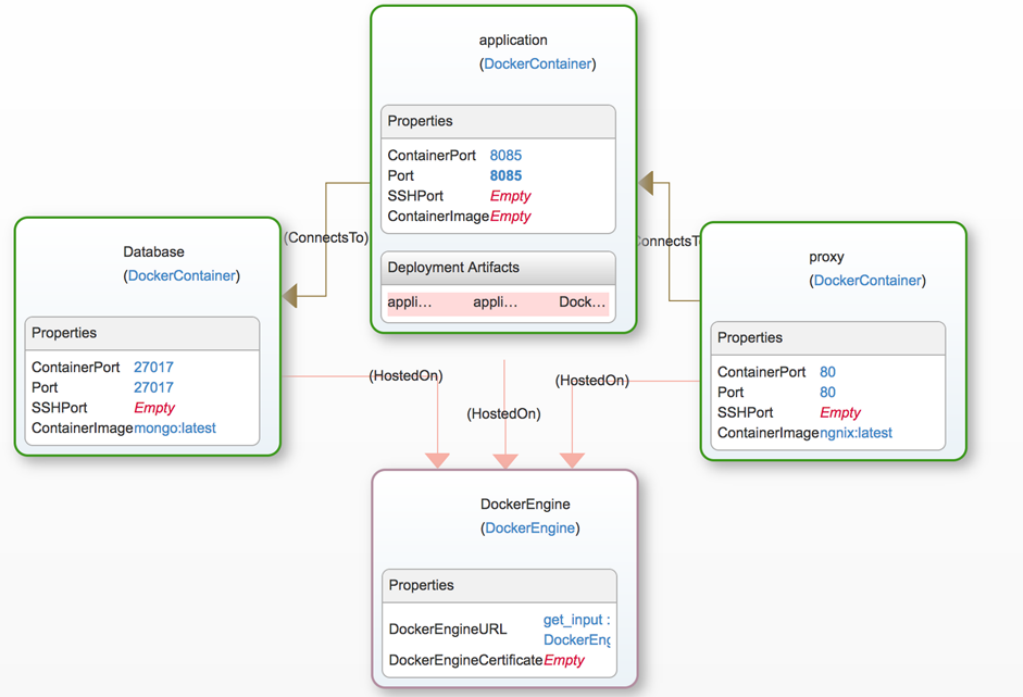

Figure 11, TOSCA description of a Docker containerized application

The TOSCA description of Linux based applications contains three custom nodes type: application, database and proxy type embedded in a Docker container. Each Docker container is executed (or hosted) on a Docker Engine that deploys, configures, starts, manages and deletes them. In contrary the TOSCA description of Windows based applications run inside virtual machines. This description incorporates all three layers presented in Fig. 4. (See Figure 13) presents the three layer description of a Windows application.

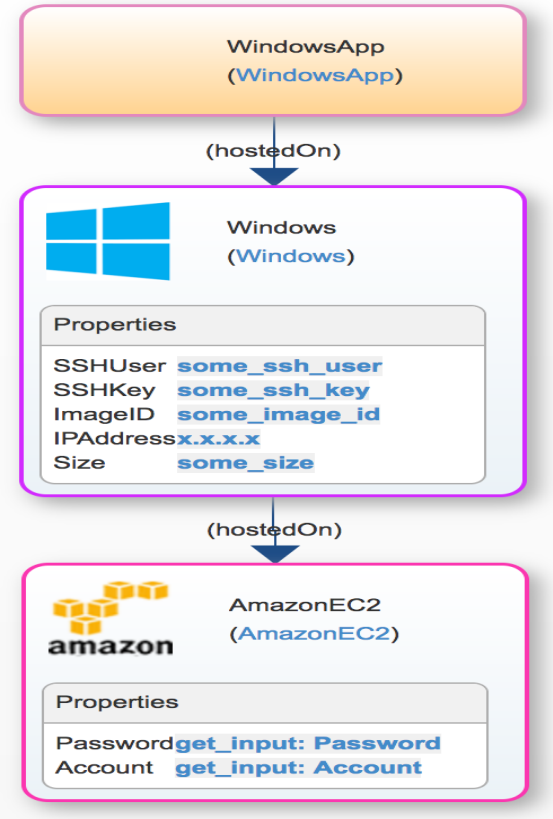

Figure 12, TOSCA description of a Windows based application

9. Policies Overview and Related Work

The problem of the description and enforcement of policies in TOSCA is an additional dimension to the description of the application topologies and their implementation that have been described in the previous sections. They define and enforce the modalities with which the services are deployed, configured and executed.

The definition and enforcement of policies is an active field of research and various solutions both generic and more specific have been proposed[10]. TOSCA offers non-normative extensions that can be used to define policies [11] and COLA will follow this approach.

The Application Descriptions Templates are shared among different components of the COLA architecture and the information flow can be drafter as illustrated in Figure 13.

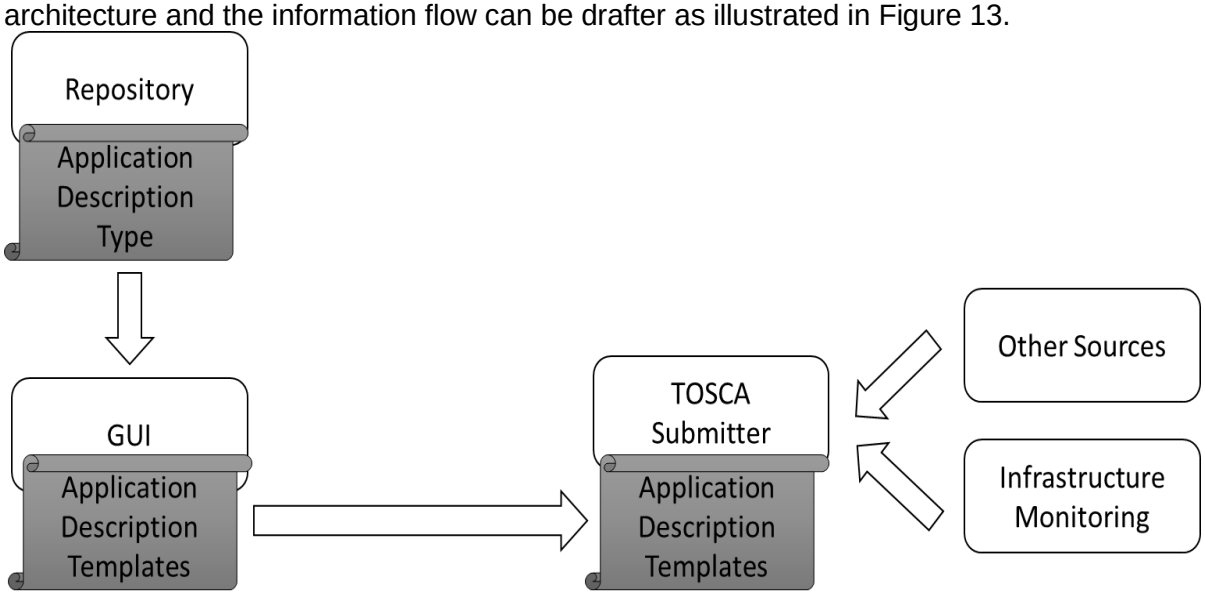

Figure 13, Application Description Types and Templates and the COLA Architecture

A **TOSCA Repository** will contain the TOSCA Application Descriptions with Policy Templates that are generic definition of policies that define the structure of the policy with default values set. The **User Interface**, (either a GUI or a CLI) will allow the Application Description Developers to define some of the values of the Policy Types, to create the Policy Template. The combined roles of the Repository and the User Interface will implement the guidelines of re-usability and the minimization of the Application Developer Intervention. Once completed, the TOSCA Application Description with the final version Policy Template (e.g. with all the values either set by the user of left to their default setting) will be parsed and interpreted by the **TOSCA Submitter.** The TOSCA Submitter will evaluate the values and thresholds defined by the policies and will compare them to the various measures obtained from the monitoring system(s) and other infrastructure information sources to enforce the related policies.

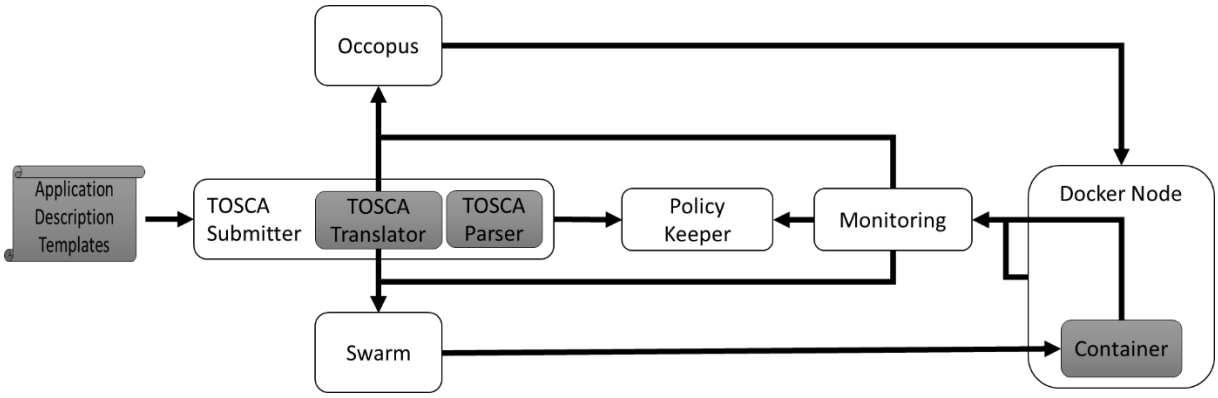

Figure 14, Application Description Templates and the COLA Architecture (Detail)

The interaction between the Application Description Templates expressed in TOSCA and the components of the COLA architecture that will enforce the policies is detailed in Figure 14.

The **TOSCA Submitter** will contain a TOSCA Parser and TOSCA Translator. that the TOSCA Parser will check the validity of the YAML[12] code of the Application Description Templates. It will pass the TOSCA descriptions to the TOSCA Translator that will instruct three components (Occopus, Swarm and the COLA Policy Keeper) in accordance to both the description of the application and the policies that norm its lifecycle. The **Policy Keeper** will enforce the policies by matching its values to those provided by the components that monitor the infrastructure and will instruct Occopus and Swarm accordingly

The interactions between Occopus, Swarm, Docker and other architecture components are further detailed in Deliverable 6.1.

10. TOSCA Policies in COLA

10.1 TOSCA Description Design Assumptions

The design proposed in this Deliverable is based on the following assumptions and restrictions

- **Declarative Model**. The policies description follows the Declarative Model. E.g. they describe a policy but they do not imperatively describe how to implement the policy, which is left to the proper components of the COLA architecture
- **Selectable and Composable Policies**. Users can select policies from a basket of existing policies but cannot create new policies
- **Modular Policies**. Users can select and specify the parameters of atomic (that cannot be decomposed further) policies of different kinds that cover the various facets of one service. These policies are combined for each service to define overall policies of that service and the overall policies of each service are then combined in the overall policies that govern the complete service topology (the application). In this first draft, users will only define policies at service level.
- **User-defined Policy Parameters**. Users can define parameters of the selected policies but cannot change the structure of the selected policies.
- **No Consistency is ensured**. Policies may be contradictive, at this stage of the policies description, only a priority level will be offered to the user to define which policy has the highest priority but this priority may not always be respected.

The concepts of modularity and consistency are further explained in Figure 15 below. Policies that define certain aspects of the entire application topology (and hence each of the services that compose the application) may raise conflicts with policies that define aspects of a specific service. Furthermore, policies that define the various aspects of the policies of individual services may raise conflicts.

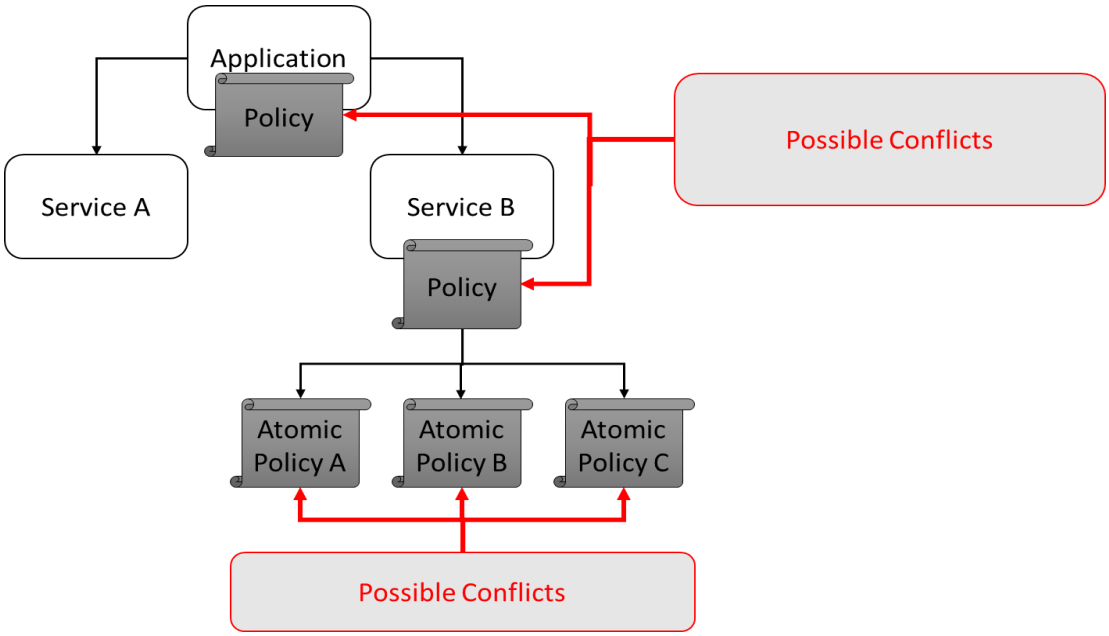

Figure 15, Policies Structure and Potential Conflicts

COLA will define policies at three different levels: at abstract policy hierarchy level, application and service policy level, and policy description level.

10.2 Abstract Policy Hierarchy in COLA

TOSCA allows to define type hierarchies of arbitrary complexity, COLA will define a multi-layered hierarchy of policies that will all derive from a Root Policy. Each Policy Type, will then be further detailed in each sub-type. The first level the abstract hierarchy of different Policy Types is depicted in Figure 16. Each of the sub-policies hierarchies are defined in Annexes 4 to 6

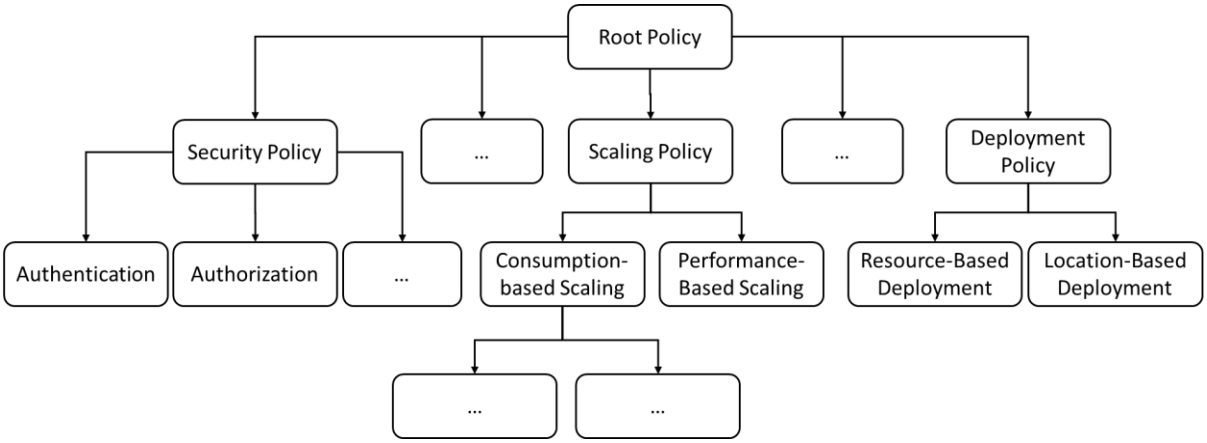

Figure 16, Abstract Policy Hierarchy

10.3 Application and Service Policy Structure in COLA

The second level of policy description is the placement and relationship between the policies and at the various level of the TOSCA Application Template. As introduced in Section 0.2, policies are composed by sub-policies and can be placed either at application (entire topology) or service (single or multiple topology node levels). This structure level of the policies is illustrated in Figure 17 below.

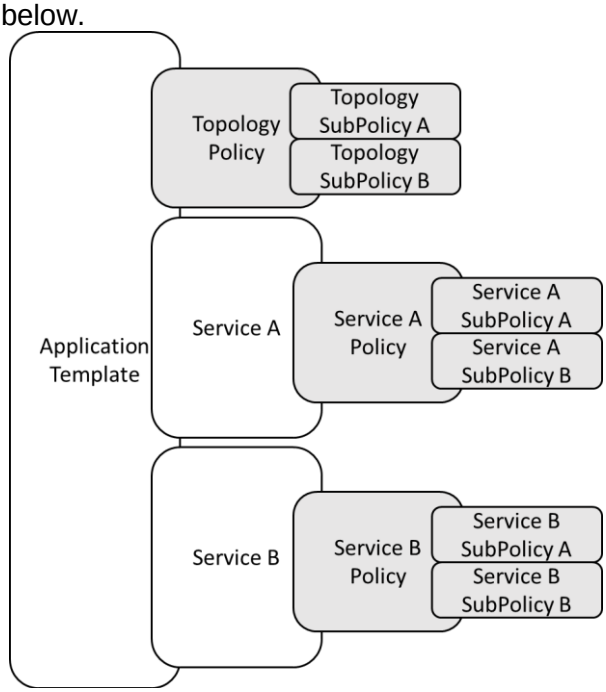

Figure 17, Policies at Application and Service Level

10.4 Policy Description Structure in COLA

Finally, each policy adheres to the generic structure drafted in Figure 18.

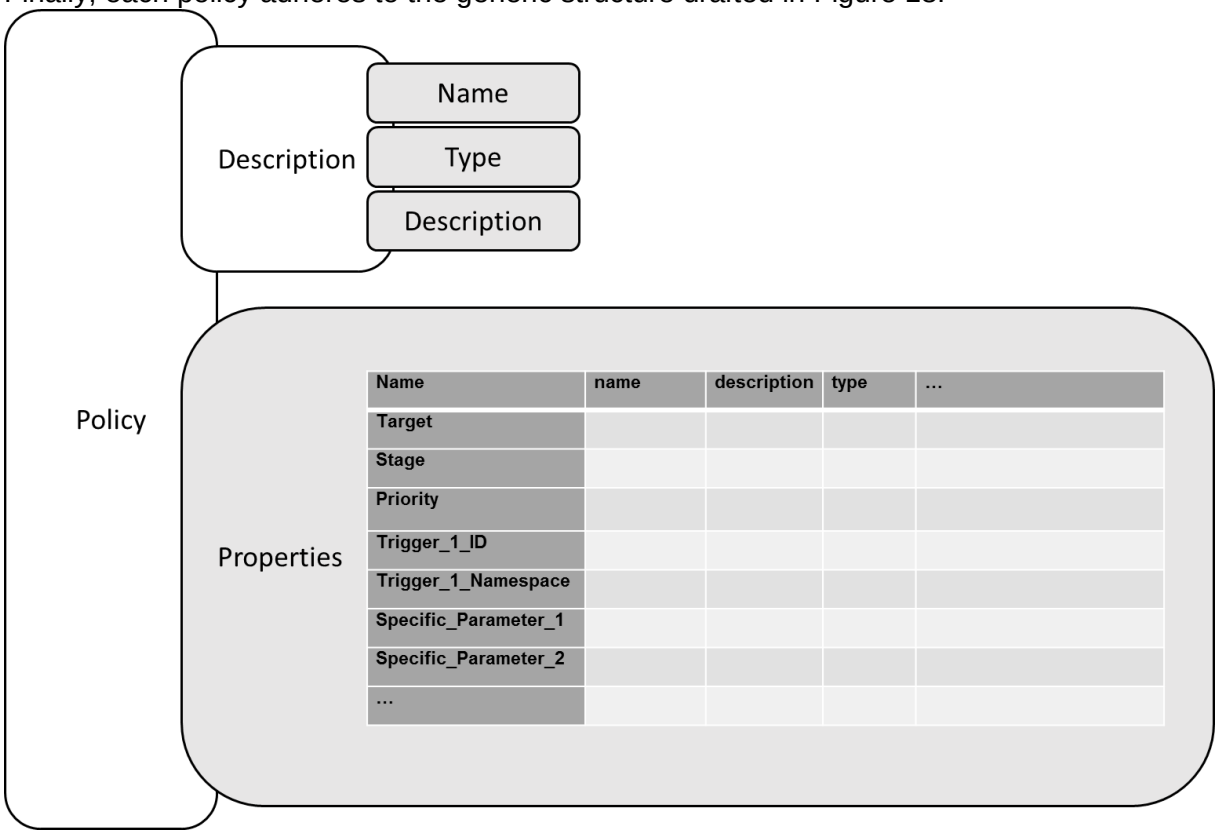

Figure 18, Generic Policy Structure

WP5 elaborated a Policy Template to support definition of COLA policies. This template is detailed in Annex 1 and examples are given in Annexes 4 to 6. Each policy is divided into two main section:

- **Description:** This section of the Policy is composed of different fields that give an overall description of on which service and during which part of their lifecycle the policy are applied. This section is further composed of the following fields:
  - Name: A String that represents the name of the Policy
  - Type: The type defined in the policy hierarchy. This field defines the link with the first level of policy structure (Abstract Policy Hierarchy Type)
  - Description: A textual description of the policy

- **Properties.** This section contains two kinds of parameters: those that are common to all COLA policies, and those that are specific to each policy.

- **Common Properties** that are present in all COLA policies (it is expected that all these parameters are defines in a COLA policy).
  - **Target**: define the technological element to which the policy has to be applied. As an example, a policy could be applied to a single service, to a set of services, or to an entire application topology. This fields define the link with the second level of the policy structure (Application and Service Policy Structure)

- **Stage**: define at which stage the policy applies. As an example, a policy may apply at deployment stage, or execution stage]. It may cover more than one stage, if no value is defined, the policy will not be implemented.
- **Priority:** This is an arbitrary integer of 0 to 100 used to define the priority with which the policy will be implemented. It is used to resolve possible conflicts with other policies, if no value is given, a default value of 50 is given to the policy. A Priority of 0 signifies that the policy will not be enforced, a Priority of 100 signifies that the policy MUST always be enforced (the case of two conflicting policies with priority 100 has not been solved yet). Any value between 1 and 99 signifies that the policy will be enforced unless it conflicts with a policy with a higher priority.

- **Specific Properties** are specific to each Policy. Most Policies will define Triggers in this sections along with their namespace. Triggers are the values that will be used to determine whether to execute or not a policy (e.g. the cpu consumption of an application). To allow for support to multiple sources of information of the infrastructure (possibly not known at the time of the specification of the policy structure), triggers are associated with a namespace. Properties are defined as a list of TOSCA parameter definition as specified in section 3.5.12 of the TOSCA Simple Profile [1] and contains various parameter definition including type, constraints and other useful fields such as required and constraints (among other). A complete description of the parameter definition is in Annex 3.

11. Conclusions

This deliverable continues the investigation on the best approach for the description of applications in the COLA project. Deliverable D5.1 detailed the reasons that lead to the choice of TOSCA and, based on this decision, the investigation focused on how to propose a TOSCA-based structure to define the multi-layer application proposed in COLA's DoW and how to best represent policies that can be applied to one or more services or to the entire application in different stages to its lifecycle.
At the time of writing of this Deliverable, there were some open issues both conceptual and technical that will be object of further investigation in conjunction with other COLA work packages.

Priority Conflict. The case in which two policies with priority set to 100 are in conflict has not solution to the moment being.

Hidden Policies. TOSCA supports the description of hidden policies (particularly regarding deployments) that are implemented with node filtering in the overall topology. There is no consensus as of now on how to address possible conflicts between these implicit policies and those that are explicitly declared as policy types.

Incomplete Hierarchies: As of now, the policy hierarchies are in draft status and they will be further detailed and modified. More specifically, Cost-Based Policies and Quality of Service Policies must be introduced, Deployment and Scaling Policies must be reviewed and detailed further and Security Policies must be reviewed when the Security Architecture will be released in its final version

12. References

[1] Various, “TOSCA Simple Profile in YAML Version 1.0,” 2014. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.html. [Accessed: 20-Jul-2017].

[2] A. Brogi, J. Soldani, and P. Wang, “TOSCA in a Nutshell: Promises and Perspectives,” *Lect. Notes Comput. Sci.*, pp. 8745171–186, 2014.

[3] Various, “Pure-Play Cloud Orchestration & Automation Based on TOSCA | Cloudify.” [Online]. Available: http://cloudify.co/. [Accessed: 26-Jul-2017].

[4] Various, “Apache Mesos.” [Online]. Available: http://mesos.apache.org/. [Accessed: 26-Jul-2017].

[5] T. Binz *et al.*, “OpenTOSCA – A Runtime for TOSCA-Based Cloud Applications,” Springer, Berlin, Heidelberg, 2013, pp. 692–695.

[6] “OpenTOSCA Eco System.” [Online]. Available: http://install.opentosca.org/. [Accessed: 26-Jul-2017].

[7] “IAAS | OpenTOSCA.” [Online]. Available: http://www.iaas.uni-stuttgart.de/OpenTOSCA/. [Accessed: 26-Jul-2017].

[8] Various, “Docker - Build, Ship, and Run Any App, Anywhere.” [Online]. Available: https://www.docker.com/. [Accessed: 26-Jul-2017].

[9] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann, and U. Breitenb, “Winery – A Modeling Tool for TOSCA-based Cloud Applications Title = {{Winery} --Modeling Tool for {TOSCA}-based Cloud Applications} Institute of Architecture of Application Systems Winery – A Modeling Tool for TOSCA-based Cloud Applications.”

[10] Various, “Policy in Tosca - Domino - OPNFV Wiki.” [Online]. Available: https://wiki.opnfv.org/display/domino/Policy+in+Tosca. [Accessed: 26-Jul-2017].

[11] T. Waizenegger *et al.*, “Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing,” Springer, Berlin, Heidelberg, 2013, pp. 360–376.

[12] Various, “The Official YAML Web Site.” [Online]. Available: http://yaml.org/. [Accessed: 24-Jul-2017].

Annex 1. COLA Policy Template

The following defines the structure of a COLA Policy Template, for the sake of conciseness and clarity only the relevant fields are added

#
Policy Description
#
Description:

Name of the Policy
type: string
required: yes
Name:

Name of the Type of the Policy, this defines which policy type is used to derive the policy
template from the policy hierarchy
type: TOSCA type
required: yes
Type:

A textual description of the policy
type: string
required: yes
Description:

Properties:

#
Common Properties
#
List of target nodes that Defines the technological layer where the policy has to be applied:
required: yes
Type: List of Tosca Node(s)
Targets:

List of stages of the nodes to which the policy applies (Deployment, Execution, etc…)
required: yes
Type: Defined as per normative TOSCA node states
Stage:

Defines importance of the policy in a range from 0 (lowest priority) to 100 (highest priority)
to resolve possible conflicts with other policies
required: yes
Type: Integer
Expected values: 0 -100
Priority:

#
The trigger(s) that are used to activate/deactivate the policies
#

The trigger identifier
required: no
Type: Tosca Parameter
Trigger_Id:

The trigger namespace
required: no
Type: Tosca Parameter
Trigger_Namespace:

#
Specific Properties
#
Each Property that is specific to the policy will be defined as a Tosca Parameter Type
required: yes/no
Type: Tosca Parameter (detailed in Annex 3)
Property_X:

Annex 2: Minimum TOSCA Service Template

```
tosca_definitions_version:
    # Required TOSCA Definitions version string

    // Optional metadata keyname: value pairs
    metadata:
        template_name:
            # Optional name of this service template
        template_author:
            # Optional author of this service template
        template_version:
            # Optional version of this service template

    description: <template_type_description>

    dsl_definitions:
        # list of YAML alias anchors (or macros)repositories:
    repositories:
        # list of external repository definitions which host TOSCA artifactsimports:
    imports
        # ordered list of import definitions

    artifact_types:
        # list of artifact type definitions
        derived_from:       <parent_artifact_type_name>
        version:            <version_number>
        description:        <artifact_description>
        mime_type:          <mime_type_string>
        file_ext:           [ <file_extensions> ]
        properties:
            <property_definitions>

    data_types:
        # list of datatype definitions
    capability_types:
        # list of capability type definitions
    interface_types
        # list of interface type definitions

    relationship_types:
        # list of relationship type definitions
        tosca.relationships.Root:
        description: The TOSCA root Relationship Type all other TOSCA base Relationship
        Types derive from
        attributes:
            tosca_id:
                type: string
```

```
        tosca_name:
            type: string
        interfaces:
            Configure:
                type: tosca.interfaces.relationship.Configure

node_types:
    # list of node type definitions
    derived_from:       <parent_node_type_name>
    version:            <version_number>
    description:        <node_type_description>
    properties:
        <property_definitions>
    attributes:
        <attribute_definitions>
    requirements:
        - <requirement_definitions>

group_types:
    # list of group type definitions
    derived_from: <parent_group_type_name>
    version:            <version_number>
    description:        <group_description>
    properties:
        <property_definitions>
    targets: [ <list_of_valid_target_types> ]
    interfaces:
        <interface_definitions>

policy_types:
    # list of policy type definitions

topology_template:
    # topology template definition of the cloud application or service
```

Annex 3: Complete TOSCA Parameter Definition

- parameter_name: represents the required symbolic name of the parameter as a string.
- parameter_description: represents the optional description of the parameter.
- parameter_type: represents the optional data type of the parameter. Note, this keyname is required for a TOSCA Property definition, but is not for a TOSCA Parameter definition.
- parameter_value, parameter_value_expresssion: represent the type-compatible value to assign to the named parameter. Parameter values may be provided as the result from the evaluation of an expression or a function.
- parameter_required: represents an optional boolean value (true or false) indicating whether or not the parameter is required. If this keyname is not present on a parameter definition, then the property SHALL be considered required (i.e., true) by default.
- default_value: contains a type-compatible value that may be used as a default if not provided by another means.
- status_value: a string that contains a keyword that indicates the status of the parameter relative to the specification or implementation.
- parameter_constraints: represents the optional sequenced list of one or more constraint clauses on the parameter definition.
- entry_description: represents the optional description of the entry schema.
- entry_type: represents the required type name for entries in a list or map parameter type.
- entry_constraints: represents the optional sequenced list of one or more constraint clauses on entries in a list or map parameter type.

Annex 4: Scaling Policies

Scaling Policies define how services should be scaled up or down. We have isolated two main classes of scaling policies: consumption based policies that scale up or down one or more services based on the consumption of some metrics obtained by the infrastructure monitoring and performance-based. The Scaling Policies Hierarchy as defined at the moment of this Deliverable (the Hierarchy is likely to be extended in the future) is represented in Figure 19

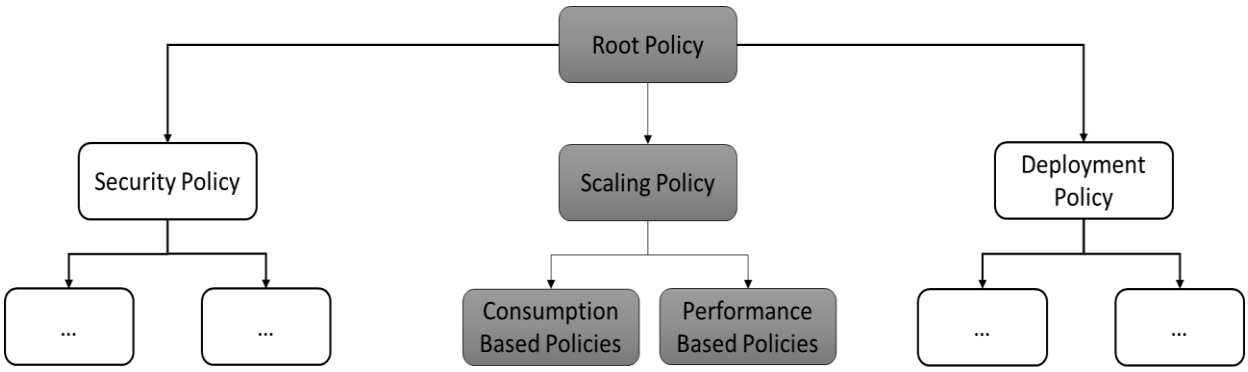

Figure 19, Scaling Policies Hierarchical Structure

Simple Consumption-Based Scaling Policy Example

One example of a Consumption-Based Policy expressed with the COLA Policy Template is described below:

#
Simple CPU Consumption Scaling Policy
#
Description:

Name: SimpleCPUConsumption
Type: Tosca.Policy.Scaling.Consumption.SimpleCPU
Description: This policy dictates that a new instance of the listed services is deployed when the CPU consumption exceeds the given threshold for more than the defined trigger time. The additional instances of the service are undeployed when the CPU consumption falls below the lower threshold for more than the defined trigger time**.**

Properties:

#
Common Properties
#
List of target nodes that Defines the technological layer where the policy has to be applied:
required: yes
Type: List of Tosca Node(s)
Targets: [TopologyTemplate.NoteTemplates.ServiceX]

List of stages of the nodes to which the policy applies (Deployment, Execution, etc…)
required: yes
Type: Defined as per normative TOSCA node states
Stage: Started

Defines importance of the policy in a range from 0 (lowest priority) to 100 (highest priority)
to resolve possible conflicts with other policies
required: yes
Type: Integer
Expected values: 0 -100
Priority: 100

#
The trigger(s) that are used to activate/deactivate the policies
#
The trigger identifier
required: no
Type: Tosca Parameter
Trigger_Id: CPU_TIME

The trigger namespace
required: no
Type: Tosca Parameter
Trigger_Namespace: Prometheus

#
Specific Properties
#
Each Property that is specific to the policy will be defined as a Tosca Parameter Type
required: yes/no
Type: Tosca Parameter (detailed in Annex 3)
Property1:
Parameter_name: max_cpu_threshold
Parameter_description: the max cpu above which the service(s) will be scaled up by deploying new instance.
Parameter_type: integer
Parameter_value: 80
Parameter_required: true
Default_Value: 80
Parameter_contraints: [0, 100]
Property2:
Parameter_name: min_cpu_threshold
Parameter_description: the min cpu below which the service(s) will be scaled down by undeploying one instance.
Parameter_type: integer
Parameter_value: 20
Parameter_required: true
Default_Value: 20
Parameter_contraints: [0, 100]
Parameter_trigger: CPU_CONSUMPTION
Parameter_trigger_namespace: PROMETHEUS
Property3:

Parameter_name: cpu_reaction_time
Parameter_description: the time (in seconds) in which the trigger must be above or below the threshold before the policy is enforced.
Parameter_type: integer
Parameter_value: 600
Parameter_required: true
Default_Value: 600
Parameter_contraints: [0, 3600]

Annex 5: Placement Policies

Deployment (or placement) Policies define where services should be deployed. We have isolated two main classes of deployment policies: Resource based policies that decide where to deploy a service based on the resources available (Computational and Storage Resources or even available services) or Location Based Deployment that decide where to deploy a service based on its geographical location (as an example for data protection/privacy law compliancy). The Deployment Policies Hierarchy as defined at the moment of this Deliverable (the Hierarchy is likely to be extended in the future) is represented in Figure 19. TOSCA does provide the possibility to define implicit deployment policies (by node matching and filtering) without having to explicitly declare a policy but in COLA, we suggest to try to avoid this solution as it will not result visible as a policy to application developers.

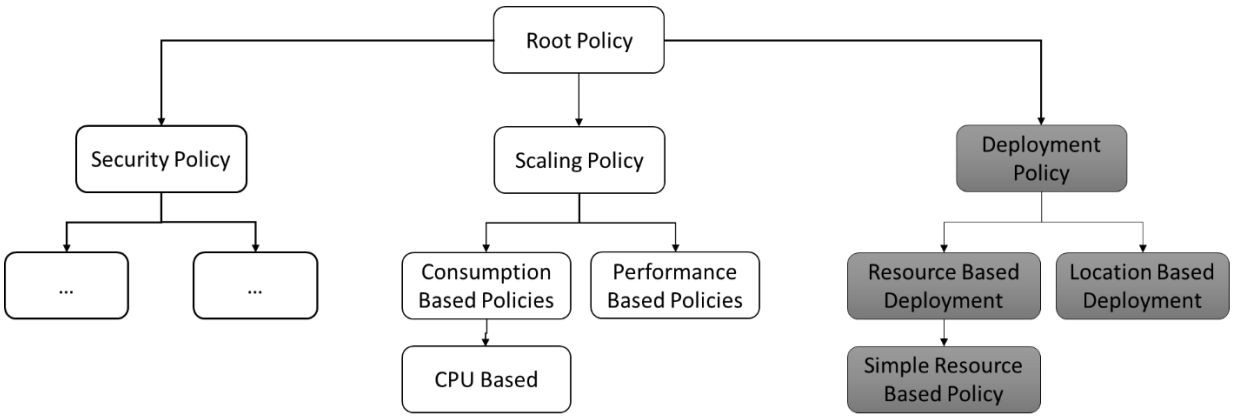

Figure 20, Deployment Policies Hierarchical Structure

Simple Resource Based Deployment Policy Example

One example of a Simple Resource Deployment Policy expressed with the COLA Policy Template is described below:

#
Simple CPU-Based Resource Deployment Policy
#

Description:

Name: SimpleCPUDeployment

Type: Tosca.Policy.Deployment.ResourceBased.SimpleCPU

Description: This policy dictates that the services specified must be deployed on resources with a number of CPU greater to the specified value**.**

Properties:

#
Common Properties
#
List of target nodes that Defines the technological layer where the policy has to be applied:
required: yes
Type: List of Tosca Node(s)

Targets: [TopologyTemplate.NoteTemplates.ServiceX]

List of stages of the nodes to which the policy applies (Deployment, Execution, etc…)
required: yes
Type: Defined as per normative TOSCA node states
Stage: Initial

Defines importance of the policy in a range from 0 (lowest priority) to 100 (highest priority)
to resolve possible conflicts with other policies
required: yes
Type: Integer
Expected values: 0 -100
Priority: 100

#
The trigger(s) that are used to activate/deactivate the policies.
This policy has not triggers
#

#
Specific Properties
#
Each Property that is specific to the policy will be defined as a Tosca Parameter Type
required: yes/no
Type: Tosca Parameter (detailed in Annex 3)
Property1:

Parameter_name: min_available_cpu
Parameter_description: the minimum number of cpu that must be available for the service to be deployed.
Parameter_type: integer
Parameter_value: 4
Parameter_required: true
Default_Value: 4
Parameter_contraints: [0, 64]

Annex 6: Security Policies

Security Policies have been the topic of active cooperation between WP5 and WP7 and are at the moment, more detailed than the other basic types, their hierarchical structure is shown in Figure 21

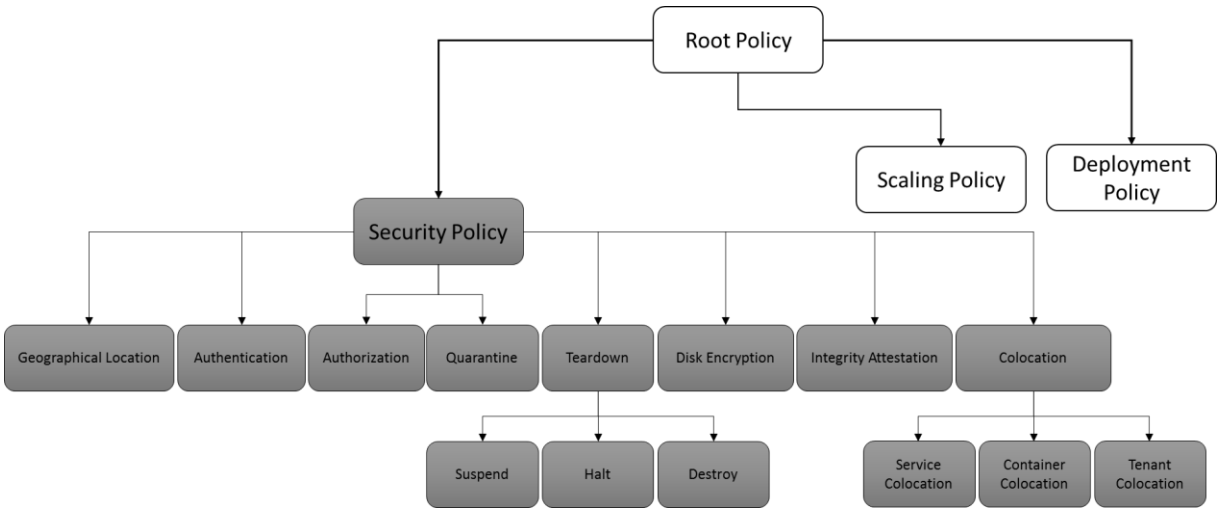

Figure 21, Security Policies Hiearchical Structure

Security Policies Types cover the following main aspects:

The **Authentication** policy defines the communication channel that will be used between the server and the entities that wish to authenticate themselves as well as the possible ways that an entity can be authenticated (e.g. providing a valid certificate). In addition to that, authentication policy is responsible for defining parameters such as the expiration of a session, how the log files will be stored and how many failure authentication requests are allowed by a user.

The **Authorization** policy controls the authorization of components, in the sense of granting a component access to a resource. Note that the component may need to be authenticated, as a pre-condition of being authorized to access a certain resource. This is a targeted policy, which is applicable to the top layer of the COLA Architecture, namely the service layer.

The **Resource Colocation** policy controls the colocation of components, in the sense of sharing the underlying physical of virtual platform with peer components. This is a recursive policy, which is applicable – with respective modifications – to several layers of the COLA Architecture.

The **Disk Encryption** policy describes the confidentiality mechanisms that will be responsible for protecting sensitive data from external attacks. The policy allows to define the encryption algorithm that will be used for the encryption of a storage resource as well as the length of the underlying encryption key.

The **Data Location** policy controls the placement of data across geographically distributed sites. Such sites may be located in different availability zones or different jurisdictions. The policy views the sites as passive endpoints capable of reading and writing data. The sites are not able to perform actions such as modification or analysis of the stored data.

The **Integrity Attestation** policy ensures that a cloud host is running in a trusted state. A trusted state is defined based on a set of predefined security profiles.

The **Service Quarantine** policy defines the behaviour in the situation when a service site must be quarantined. The policy states the conditions of quarantine and remediation actions to maintain availability in such situations.

The **Resource Teardown** policy controls the decommissioning of components, in the sense of withdrawing them from the deployment – either temporarily or permanently. The goal of the policy is to ensure that components are decommissioned without the risk of exposing user code, data or configuration. This is a recursive policy, which is applicable – with respective modifications – to several layers of the COLA Architecture.

Authentication Policy Example

One example of an Authentication Policy expressed with the COLA Policy Template is described below:

#
Simple CPU-Based Resource Placement Policy
#

Description:

Name: Authentication

Type: Tosca.Policy.Security.Authentication

Description: The authentication policy defines the communication channel that will be used between the server and the entities that wish to authenticate themselves as well as the possible ways that an entity can be authenticated (e.g. providing a valid certificate). In addition to that, authentication policy is responsible for defining parameters such as the expiration of a session, how the log files will be stored and how many failure authentication requests are allowed by a user.

Properties:

#
Common Properties
#
List of target nodes that Defines the technological layer where the policy has to be applied:
required: yes
Type: List of Tosca Node(s)
Targets: [TopologyTemplate.NoteTemplates.ServiceX]

List of stages of the nodes to which the policy applies (Deployment, Execution, etc…)
required: yes
Type: Defined as per normative TOSCA node states
Stage: Initial, Created, Configured, Started

Defines importance of the policy in a range from 0 (lowest priority) to 100 (highest priority)
to resolve possible conflicts with other policies
required: yes
Type: Integer
Expected values: 0 -100
Priority: 100

#
The trigger(s) that are used to activate/deactivate the policies.
This policy has not triggers
#

#
Specific Properties
#
Each Property that is specific to the policy will be defined as a Tosca Parameter Type
required: yes/no
Type: Tosca Parameter (detailed in Annex 3)

Property1:
Parameter_name: communication_channel
Parameter_description: Boolean parameter to enforce the use of SSL/TLS.
Parameter_type: boolean
Parameter_value: true
Parameter_required: true
Default_Value: true

Property2:
Parameter_name: authentication_service_type
Parameter_description: defines which authentication service to be used
Parameter_type: String
Parameter_value: LDAP
Parameter_required: true

Property3:
Parameter_name: authentication_service_url
Parameter_description: defines which url of the authentication service to be used
Parameter_type: url
Parameter_value: ldap://ldap.example.com
Parameter_required: true

Property4:
Parameter_name: authentication_service_port
Parameter_description: defines which port of the authentication service to be used
Parameter_type: integer
Parameter_value: 389
Parameter_required: true

Property4:
Parameter_name: authentication_service_protocol
Parameter_description: defines which protocol to access the authentication service
Parameter_type: string
Parameter_value: TCP
Parameter_required: true

Property5:
Parameter_name: authentication_failure_limit
Parameter_description: A limit for available login failures
Parameter_type: integer
Parameter_value: 3

Parameter_required: true
Default_Value: 3
Parameter_constraints: [0, 100]

Property7:

Parameter_name: unlock_modality
Parameter_description: Describes which modality can be used to unlock the account after the maximum number of failed attempts has been reached
Parameter_type: string
Parameter_value: time
Parameter_required: true
Default_Value: time
Parameter_constraints: {time, admin_reset}

Property8:

Parameter_name: unlock_time
Parameter_description: Describes how long (in seconds) the account must be inaccessible after the maximum number of logins has been reached
Parameter_type: integer
Parameter_value: 600
Parameter_required: true
Default_Value: 600
Parameter_constraints: [0, 3600]

Property9:

Parameter_name: cookies_enabled
Parameter_description: Specifies if cookies will be enabled
Parameter_type: boolean
Parameter_value: true
Parameter_required: true
Default_Value: true

Property10:

Parameter_name: revalidate_cookies
Parameter_description: Specifies if a user can login by revalidating a valid cookie
Parameter_type: boolean
Parameter_value: false
Parameter_required: true
Default_Value: false

Property11:

Parameter_name: log_history
Parameter_description: Boolean parameter specifying if a log history should be kept
Parameter_type: boolean
Parameter_value: true
Parameter_required: true
Default_Value: true

Property11:

Parameter_name: log_history_encryption
Parameter_description: Boolean parameter specifying if a log history should be encrypted
Parameter_type: boolean
Parameter_value: true

Parameter_required: true
Default_Value: true