Orchestrating applications with
TOSCA and Docker

Candidate
Luca Rinaldi

Supervisors
Antonio Brogi
Jacopo Soldani

Academic Year 2015/2016
Contents

1 Introduction 2

2 Background 5
   2.1 TOSCA ......................................................... 5
   2.2 Docker .......................................................... 6
      2.2.1 Containers and Images ................................. 7
      2.2.2 Volumes .................................................. 8
      2.2.3 Linking Containers .................................... 9
      2.2.4 Docker Orchestration ................................ 9

3 Design of TosKer 11
   3.1 Requirements and functionalities .......................... 11
   3.2 Custom TOSCA types ......................................... 13
      3.2.1 Artifact types .......................................... 14
      3.2.2 Node types ............................................. 14
      3.2.3 Relationship types .................................... 18
   3.3 Overall architecture ........................................ 20
      3.3.1 TOSCA utility module .................................. 21
      3.3.2 Docker interface module ............................... 23
      3.3.3 Manager modules ...................................... 23
      3.3.4 Orchestrator module .................................. 25

4 Implementation of TosKer 28
   4.1 Language and libraries ...................................... 28
   4.2 Code structure ............................................... 29
   4.3 Implementation details ...................................... 30
      4.3.1 TOSCA functions ....................................... 30
      4.3.2 Deployment Graph ...................................... 32
## CONTENTS

4.3.3 Software deployment ........................................... 36
4.3.4 Networking ....................................................... 40
4.4 Packaging of TosKer ............................................... 42
4.5 Testing ................................................................... 43

5 TosKer at work ......................................................... 46
  5.1 How to install TosKer ............................................... 46
  5.2 How to use TosKer .................................................. 47
  5.3 Case study: The ThoughtsSharing application ................. 48
    5.3.1 Modelling the application ................................... 48
    5.3.2 Deploying and managing ThoughtsSharing ......... 54

6 Related work ............................................................. 60
  6.1 Docker Orchestrators ............................................... 60
  6.2 TOSCA Deployers .................................................. 61
  6.3 Cloudify ............................................................... 62
  6.4 Brooklyn, Clocker and Brooklyn-TOSCA ..................... 64
  6.5 Summary ............................................................. 65

7 Conclusions .............................................................. 67

Appendices ................................................................. 73

A TosKer Types ........................................................... 74
Abstract

The objective of this thesis was to contribute to automating the deployment of (complex) applications over heterogeneous infrastructures, by trying to combine the orthogonal approaches proposed by TOSCA and Docker. After analysing the state of the art, we designed and prototyped TosKer — the first engine (to the best of our knowledge) that is capable to input a TOSCA description of a multi-component application, and to automatically deploy and orchestrate it on top of Docker.

The source code of TosKer prototype is available on GitHub under the MIT licence at https://github.com/di-unipi-socc/tosKer
Chapter 1

Introduction

The possibility of automating the deployment of (complex) applications over heterogeneous infrastructures, by taking into account both application requirements and infrastructure constraints, is receiving increasing attention [30]. TOSCA [33] and Docker [16] are two emerging solutions trying to address this problem from different perspectives.

On the one hand, TOSCA (Topology and Orchestration Specification for Cloud Applications [33]) follows a model-driven approach. More precisely, TOSCA [33] is an OASIS standard that provides a YAML-based modelling language for specifying portable cloud applications, and for automating their deployment and management. TOSCA permits describing the structure of a cloud application as a typed, directed topology graph, whose nodes represent application components, and whose arcs represent dependencies among such components. Each node of a topology can also be associated with the corresponding component’s requirements, the operations to manage it, the capabilities it features, and the policies applied to it. Inter-node dependencies associate the requirements of a node with the capabilities featured by other nodes. An application topology can be automatically processed to deploy such application on a TOSCA-compliant cloud platform.

On the other hand, Docker [16] follows a “snapshot-based” approach. More precisely, Docker [16] is an open source platform for building, shipping, and running software components, together with their dependencies, in lightweight virtual environments, called containers. Docker containers run from Docker images, which are the read-only templates used to create them. A Docker image packages any software dependency (e.g., source code, configuration file,
binaries) needed to run a software component. Images hence play a crucial role within Docker, and to ease images distribution Docker permits sharing them through so-called Docker registries (e.g., Docker Hub, which is the official registry for storing and retrieving Docker images). Moreover, due to the file-system structure of Docker, it is not possible to store persistently data inside a container. For this reason Docker introduces Docker volumes, which are independent entities that can be attached to one or more Docker containers to permanently store data.

Of course, both TOSCA and Docker have pros and cons. For instance, while TOSCA is a well-documented standard that permits orchestrating complex applications formed by heterogeneous components, application descriptions tend to become too verbose. In addition to that, there is a lack of orchestration engines that support TOSCA. Docker instead is a production ready tool, with a big repository of images (Docker Hub), but it is not designed to orchestrate complex applications composed by multiple and heterogeneous components, as its main purpose is to permit managing applications composed by Docker containers and Docker volumes.

The objective of this thesis was to identify and develop a solution that trades off the pros and cons of TOSCA and of Docker. For this reason we designed and prototyped TosKer, an orchestration engine capable of deploying, on top of Docker, applications described in TOSCA YAML. More precisely, such orchestration engine inputs a TOSCA description of a multi-component application, where some components are Docker containers and Docker volumes, and automatically deploys and orchestrates it using the Docker engine.

The first phase of our work was devoted to analysing the state of the art, which confirmed the lack of solutions trying to suitably trade-off pros and cons of TOSCA and Docker. Indeed, to the best of our knowledge, current solutions either provide a support for TOSCA (e.g., Alien4Cloud [1], and SeaClouds [37, 12]) or permit orchestrating multi-component Docker applications (e.g., Kubernetes [29], Mesos [5], and Docker Compose [17]). None of them is supporting both TOSCA and Docker (i.e., none of the analysed solutions permits describing multi-components applications in TOSCA, by also allowing some of their components to be Docker containers or Docker volumes).

The second phase of our work was devoted to designing and prototyping TosKer, which orchestrates applications composed by generic software components, which have to be hosted on Docker containers, which in turn may exploit
Docker volumes to persist data. Components, containers, and volumes have a different lifecycles, and they may be interconnected based upon the relationships occurring among them (i.e., network connections, generic dependencies, and storage attachments). The design phase started with the definition of a set of TOSCA custom types, which represent the components orchestrated by TosKer (viz., generic software components, Docker containers, and Docker volumes). We then analysed the features that TosKer must implement (i.e., parsing a TOSCA specification, interacting with the Docker infrastructure, orchestrating components), and we created a set of modules to cover those aspects. Finally, we developed a Python prototype of TosKer, and we released it under open source MIT licence.

To the best of our knowledge, TosKer is the first orchestration engine that permits orchestrating TOSCA applications combining heterogeneous components, including Docker containers and Docker volumes. TosKer hence advances the state-of-the-art for orchestrating application with Docker and TOSCA. Indeed, on the one hand, TosKer extends the orchestration capabilities of Docker by allowing to orchestrate generic software components (in addition to Docker containers and Docker volumes). On the other hand, TosKer extends the current support for TOSCA by providing a Docker-based orchestration engine for TOSCA applications.

The rest of this thesis is organised as follows. Chapter 2 provides the necessary background on TOSCA and Docker. Chapter 3 describes the design of TosKer, while Chapter 4 discusses the implementation of TosKer. Chapter 5 illustrates how to use TosKer, also by referring to a concrete case study. Chapters 6 and 7 discuss related work and draw some concluding remarks, respectively.
Chapter 2

Background

2.1 TOSCA

TOSCA \[33\] is an OASIS standard whose main goals are to enable (i) the specification of portable cloud applications and (ii) the automation of their deployment and management. TOSCA provides a YAML-based and machine-readable modelling language that permits describing cloud applications. Obtained specifications can then be processed to automate the deployment and management of the specified applications. We hereby report only those features of the TOSCA modelling language that are used in this thesis\(^1\).

TOSCA permits specifying a cloud application as a service template, that is in turn composed by a \texttt{topology_template}, and by the types needed to build such a \texttt{topology_template} (Figure 2.1). The \texttt{topology_template} is a typed directed graph that describes the topological structure of the composite cloud application. Its nodes (called \texttt{node_templates}) model the application components, while its edges (called \texttt{relationship_templates}) model the relations occurring among such components (if any). The \texttt{topology_template} may also contain policies, which are typed by means of \texttt{policy_types} and permit specifying non-functional information about the \texttt{node_templates} they target.

\(^1\)A more detailed, self-contained introduction to TOSCA can be found in \[9, 11\].
CHAPTER 2. BACKGROUND

The node templates and relationship templates are typed by means of node types and relationship types, respectively. A node type defines the observable properties of a component, its possible requirements, the capabilities it may offer to satisfy other components’ requirements, and the interfaces through which it offers its management operations. Requirements and capabilities are also typed, to permit specifying the properties characterising them. A relationship type instead describes the observable properties of a relationship occurring between two application components.

Finally, it is worth noting that the TOSCA type system supports inheritance. A node type can be defined by extending another, thus permitting the former to inherit the latter’s properties, requirements, interfaces, and operations. Analogously, a relationship type (as well as a requirement type, a capability type, or a policy type) can extend another to inherit all its properties.

2.2 Docker

Docker [16] is a Linux-based platform for developing, shipping, and running applications through container based virtualisation. Container-based virtualisation exploits the kernel of the host’s operating system to run multiple guest
instances. Each guest instance is called a container, and each container is isolated from others (i.e., each container has its own root file-system, processes, memory, devices, and network ports).

Docker achieves container virtualisation through the Docker engine (Figure 2.2), which acts as a server on the host operating systems and enables containers to be built, shipped, and run. To ensure isolation among different containers, the Docker engine uses Linux kernel containment (LXC [38]) features, such as control groups and namespaces.

![Figure 2.2: Container virtualisation in Docker.](image)

### 2.2.1 Containers and Images

Each container packages the applications to run, along with whatever software they need (e.g., libraries, binaries, etc.). Containers are built by instantiating so-called Docker images.

A Docker image is a read-only template that provides the set of instructions for creating a container. It is built by layering multiple images (Figure 2.3), with the bottom image being the base image, and with each image being the parent of the image right above it. Each image has its own file system, that extends the file system of the parent image (by exploiting UnionFS [34]). Since all file systems are mounted as read-only, Docker permits changes to a container that is launched from an image by adding a writable layer (i.e., a read-write layer).

---

2 Each interaction with the Docker engine is carried out through a so-called Docker client.
file system) on top. The processes launched inside the container run on this read-write layer, and any change that occurs is stored at this layer.

A Docker image can be built by loading a base image, by performing the necessary updates to that image, and by committing the changes. Alternatively, one can write a Dockerfile, which is a configuration file containing the instructions for building an image. The latter provides a more effective way to build images, as it only involves writing some configuration instructions (like installing software or mounting volumes), instead of having to launch a container and to manually perform and commit changes. This permits creating independently deployable containers that can package application components.

It is also possible to look for existing images instead of building them from scratch. Images are stored in registries, like Docker Hub\footnote{https://hub.docker.com}. Inside a registry, images are stored in repositories. Each repository is devoted to a specific software application, and can contain different versions of such software. Each image is simply identified by the name of the repository it comes from and by the tag assigned to the version it represents, which can be used for retrieving it.

### 2.2.2 Volumes

Docker containers are volatile: A container runs until the stop command is issued, or as long as the process from which it has been started is running. By default, the data produced by a container is lost when a container is stopped, and even if the container is restarted, there is no way to access the data previously produced. This is why Docker introduces volumes.
A volume is a directory in a container, which is designed to let data persist, independently of the container’s lifecycle. Docker therefore never automatically deletes volumes when a container is removed, and it never removes volumes that are no longer referenced by any container.

In is also possible to give a name to a volume and manage it as independent entity (e.g., create, attach to containers, and remove). Moreover, a volume can be also mounted to a host directory, thus the container and the host can share data.

2.2.3 Linking Containers

A software component running in a container application may need to interact with another component running in another container. A basic approach to support this is the so-called network port mapping: A container can be configured to expose network ports externally (by mapping the container’s ports to ports of the host operating system). However, the Docker user guide [18] discourages this approach, since it “limits” the amount of potential port mappings, and makes a container’s connections unsafe (as its ports are public).

Other approaches are linking and Docker networks. Linking is a communication method between two Docker containers that permits to securely transfer data from one container to another (without requiring the former to publicly expose its network ports). By linking a recipient container to a source container, the recipient gets access to the data stored in the source.

The above approach is now discouraged in favour of Docker networks [20], which are virtual networks toward containers can dynamically be attached. Docker networks improve container networking, by also give the possibility to create user-defined networks specifying network type (e.g., bridge network, overlay network), and network addresses.

2.2.4 Docker Orchestration

The term orchestration refers to the composition, coordination and management of multiple software components, including middleware and services. In the context of container-based virtualisation, this corresponds to composite applications whose components independently run in their own containers and talk to each other by exploiting lightweight communication mechanisms, such as linking.
Docker supports orchestration with two dedicated applications. Docker *Swarm* \[23\] permits creating a cluster of Docker containers and turning it into a single, virtual Docker container (with a container acting as “master” and scheduling incoming tasks to “worker” containers). Docker *Compose* \[23\] permits creating multi-container applications, by specifying the containers’ images to run and the links occurring among them. Docker Compose and Docker Swarm aim to have seamless integration, meaning that you could describe a multi-container application with Docker Compose, and deploy it with Docker Swarm \[24\].
Chapter 3

Design of TosKer

3.1 Requirements and functionalities

The main goal of this thesis is to develop an orchestrator engine for deploying and managing composite cloud applications, by exploiting TOSCA to specify such applications, and Docker to concretely ship and run their components.

TOSCA permits describing the topology of a composite cloud application as a typed directed topology graph, whose nodes represent application components, and whose arcs represent the relationships occurring among such components. In this thesis, we focus on applications that are composed by three main component types: generic software components, Docker containers, and Docker volumes. Each component type must described and processed differently to be effectively deployed and managed. For example a software component has to specify how it has to be installed and managed throughout its lifecycle (by providing the scripts/installables/executables that allows to install and manage such component), while a container node has to specify the image to use to create such container and where to find it.

Relationships are also of different types, and each of them has to be processed properly. For example, given two application components $C_1$ and $C_2$:

- $C_1$ may be “hosted on” $C_2$, and this would mean that $C_1$ has actually to be installed in $C_2$.

- $C_1$ may be “connected to” $C_2$, and this would mean that a network communication has to be settled to allow to $C_1$ and $C_2$ to communicate.
• If $C_1$ is a container and $C_2$ is a volume to which $C_1$ has to be connected, then $C_1$ may be “attached to” $C_2$.

• $C_1$ may be specified as generically “depending on” $C_2$, and this would mean that $C_1$ cannot be installed until $C_2$ is not available.

We also identify some requirements on how to interconnect components and relationships to structure the topology of a composite cloud application:

• The directed graph describing the topology of an application has to be acyclic.

• Each software component has to be “hosted on” exactly one other application component, which can either be a software component or a Docker container. Docker containers and Docker volumes are not “hosted on” other application components.

• Relationships of type “attaches to” can only connect Docker containers to Docker volumes.

• Relationships of types “depends on” and “connects to” can be used to arbitrarily interconnect software components and Docker containers.

From the above, it also follows that all specifiable application topologies have (as a leaf) at least a Docker container, which has no dependencies on other Docker containers, and which may be attached to Docker volumes.

So far we have talked about containers as compute nodes, something similar to virtual machines where it is only possible to host some software. Instead, containers can also be intended as black boxes that execute a command. Users should be able to distinguish among those two types of containers (as such distinction cannot be automatically determined a priori for all containers\(^1\)).

For example, the official DockerHub image and the official DockerHub image represent a perfect example of those two kinds of behaviour. Indeed, the image is intended to be used as instance of database, thus it has as start-up command that executes the MySQL database. Instead, the image represents a Linux distribution, thus it is only useful as base-image to install software.

\(^1\)Most of existing containers (e.g. nginx, mongo) can be both used as compute nodes and as black-boxes executing programs
3.2 Custom TOSCA types

We hereby illustrate the custom TOSCA definitions that are required to model the composite cloud applications described in the previous section. More precisely, we illustrate the node types and artefact types that permit modelling software components, Docker containers, and Docker volumes\(^2\). We are instead not going to introduce any novel type for modelling relationships, as the normative TOSCA types (viz., `tosca.relationships.DependsOn`, `tosca.relationships.HostedOn`, `tosca.relationships.ConnectsTo`, `tosca.relationships.AtatchesTo`) already permit providing all the necessary information.

Before delving into details, it is worth noting that TOSCA is already providing a normative specification of generic software components, and a non-normative specification of Docker containers. However, such specifications are including information that is not necessary for us, and we would have needed to extend them to model some other information that is missing there (e.g., the capability and requirements specifying the interconnection constraints listed in Section 3.1). For this reason, we created our own alphabet of types (3.1 and 3.2). More precisely, we defined two artefacts types (viz., `tosker.artifacts.Image` and `tosker.artifacts.Dockerfile`), three node types for modelling Docker containers (viz., `tosker.nodes.Container` and `tosker.nodes.Container.Executable`), a node type for modelling Docker volumes (viz., `tosker.nodes.Volume`), and a node type for modelling software components (viz., `tosker.nodes.Software`).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{tosker_artifacts_types.png}
\caption{Hierarchy structure of the artifacts types.}
\end{figure}

\(^2\)In this section we only describe such types. Their actual specification in TOSCA YAML can be found in Appendix A.
3.2.1 Artifact types

The types `tosker.artifacts.Dockerfile` and `tosker.artifacts.Image` respectively permit describing Dockerfiles that will be used to build containers and Docker images (which can be both present locally or on a remote repository).

3.2.2 Node types

3.2.2.1 Container types

The two different types modelling Docker containers permit distinguishing the two different situations described in Section 3.1.

- The type `tosker.nodes.Container` type (Figure 3.3) is the base type and it permits modelling containers whose only purpose is to host software, as witnessed by the offered `host` capability. This type also permits specifying `storage` requirements (if a container needs a connection to a volume), and the specification of the port binding (through the property `ports`).

- The type `tosker.nodes.Container.Executable` (Figure 3.4), extends the type `tosker.nodes.Container` and it permits modelling containers that not only host software, but also running a default command/software when activated. More precisely, the `tosker.nodes.Container.Executable` offers more operations to manage its lifecycle, as it can also be started and stopped. It also exposes two additional, optional requirements, called...
CHAPTER 3. DESIGN OF TOSKER

Endpoint and Dependency. The former permits specifying that the described container requirements a connection to another container or software, while the latter permits specifying that it depends on another component.

![Graphical representation of tosker.nodes.Container](image)

**Figure 3.3:** Graphical representation of tosker.nodes.Container.

![Graphical representation of tosker.nodes.Container.Executable](image)

**Figure 3.4:** Graphical representation of tosker.nodes.Container.Executable.

Listing 3.1 provide an example of the use of the node type tosker.nodes.-Container.Executable. The listing indeed describes an application containing a single Docker container, which is created from an ubuntu Docker image, and which executes the command "echo Hello World!" when activated (hence overriding the default start-up command).

**Listing 3.1:** A simple example of use tosker.nodes.Container.Executable.

```python
node_templates:
  hello_container:
    type: tosker.nodes.Container:Executable
    artifacts:
      hello_image:
```

1
2
3
4
5

15
3.2.2.2 Volume type

The node type `tosker.nodes.Volume` (Figure 3.5) permits modelling named Docker volumes. The node type `tosker.nodes.Volume` offers a capability attachment so that Docker containers can be attached to specified volumes (through relationships of type `tosca.relationships.AttachesTo`). It is possible to have multiple containers attached to the same volume. Additionally, it is possible to specify the maximum size of the volume by using the property `size`.

Listing 3.2 describes a Docker container built from a `mysql` image and attached to a Docker volume named `mysql_volume`.

Listing 3.2: An example of specification of a Docker container attached to a Docker volume.

```yaml
node_templates:
  mysql_container:
    type: tosker.nodes.Container.Executable
    artifacts:
      my_image:
        file: mysql
        type: tosker.artifacts.Image
        repository: docker_hub
    requirements:
      - attach:
        node: mysql_volume
        relationship:
```

Figure 3.5: Graphical representation of `tosker.nodes.Volume`.
CHAPTER 3. DESIGN OF TOSKER

```yaml
properties:
  location: /var/lib/mysql

mysql_volume:
  type: tosker.nodes.Volume
  properties:
    size: 200m
```

### 3.2.2.3 Software type

The type `tosker.nodes.Software` (Figure 3.6) permits modelling generic software components. It uses the Standard lifecycle interface of TOSCA to specify the set of operations (viz., `create`, `configure`, `start`, `stop`, `delete`) that permits managing a component, and to associate to them the scripts/installables/executeables actually implementing them. This type has a mandatory `host` requirement that can be satisfy both by a container and by another software. It also has two optional requirements, to specify that is depends on/connects to another components.

![Figure 3.6: Graphical representation of tosker.nodes.Software.](image)

Listing 3.3 provides an example of application having a software component (called `nginx`) to be hosted on Docker container (called `server`). As `server` is a `tosker.nodes.Container`, we only need to specify the `ubuntu` image from which it has to be run. Instead, `nginx` is a software component, for which we have to provide the set of scripts that implement the operations of its lifecycle interface.
CHAPTER 3. DESIGN OF TOSKER

Listing 3.3: A simple example of a Software component hosted on a container.

```plaintext
node_templates:
  nginx:
    type: tosker.nodes.Software
    requirements:
      - host: server
    interfaces:
      Standard:
        create:
          implementation: create.sh
        configure:
          implementation: configure.sh
      inputs:
        port: 80
      start:
        implementation: start.sh
      stop:
        implementation: stop.sh
      delete:
        implementation: delete.sh

  server:
    type: tosker.nodes.Container
    properties:
      ports:
        80: 8080
    artifacts:
      my_image:
        file: ubuntu:16.04
        type: tosker.artifacts.Image
        repository: docker_hub
```

3.2.3 Relationship types

TosKer, together with the aforementioned node types, uses the normative TOSCA relationships. Those relationships are enough to model all possible types of connections occurring among the components of a composite cloud application. Listing 3.4 provides a concrete example of this, with two server applications on two different containers, and the server micro1 requires a connection to the server micro2. It is worth to highlight that this simple config-
uration is hard to model by using the normative TOSCA types, because the software does not have a connect requirements and an endpoint capability. Moreover, this example shows also the possibility to host one software node on top of another. Indeed, server micro1 is hosted on the nodejs1 software. This other node represent the installation of the nodejs interpreter, and so it also has only the create and delete interface.

Listing 3.4: A simple example of connection between two Software components.

```yaml
node_templates:
  micro1:
    type: tosker.nodes.Software
    artifacts:
      code: code1.js
    requirements:
      - host: nodejs1
        - connect: micro2
    interfaces:
      Standard:
      ...

  nodejs1:
    type: tosker.nodes.Software
    requirements:
      - host: server1
    interface:
      Standard:
        create:
          implementation: create_nodejs.sh
        delete:
          implementation: delete_nodejs.sh

  server1:
    type: tosker.nodes.Container
    ...

  micro2:
    type: tosker.nodes.Software
    ...
```
CHAPTER 3. DESIGN OF TOSKER

3.3 Overall architecture

In this section we describe the structure of TosKer and the design choices made\footnote{Details about the actual implementation of TosKer can be found in Chapter 4.}. The design was driven by the goal of creating a modular architecture, which partitions the aspects of the orchestration performed by TosKer in a loosely coupled fashion, and which can be easily extended by plugging-in new modules implementing new functionalities. For these reasons the TosKer architecture was divided in six main modules (Figure 3.7):

- the \textit{TOSCA utility}, that parses and validates the TOSCA yaml file,
- the \textit{Docker interface}, which communicates with the Docker engine,
- the \textit{Managers}, that implements the lifecycle for each one of the components (\textit{software, container and volume}),
- the \textit{Orchestrator}, which orchestrates the deployment.

The \textit{Orchestrator} is the module exposed to the user and it uses the \textit{Managers} to implement the deployment of the components and the \textit{TOSCA utility} to parse the TOSCA YAML application specification. The managers also use the \textit{Docker interface} as a proxy to communicate with the Docker engine.

![Figure 3.7: Design of TosKer.](image)

20
Each of those modules concerns a specific aspect of the orchestrator and interacts with the other modules through a public API. In this way it is simple to add a new functionality, for example it is possible to deploy the applications orchestrated by TosKer on clusters of Docker hosts (instead of deploying them on a single Docker host) by changing the actual implementation of the Docker interface. To achieve the goal of extendibility, we also decide to split the managers in three different modules (one module for one component), making simpler updates the lifecycle operations or even add new component types.

TosKer also exploits two external components, the TOSCA parser \[35\] developed with the OpenStack community, and the Docker engine\[4\].

In the next sections we analyse each one of the TosKer modules to understand it main characteristics.

### 3.3.1 TOSCA utility module

The TOSCA utility is in charge of parsing the TOSCA files given in input, validating it and generating a more convenient structure that the other modules can use to deploy the application. The validation phase checks that the TOSCA file is well-formatted and that it does not have any syntax error.

If the TOSCA file is valid, it is parsed by the TOSCA utility, and the result is a directed acyclic graph with all the information needed to deploy and manage the application. An example of graph is in Figure 3.8, which shows the graph resulting from the application specified in Listing 3.4.

\[4\]More details on the use of those modules can be found in Chapter [21]
After this construction phase, a topological sorting algorithm is used to find a possible deployment order of the components. This order is exploited by the Orchestrator to deploy the application in such a way that all dependencies are satisfied. Figure 3.9 shows a possible result of this algorithm with the numbering of the components. In this example the deployment order is the following: (1) server1, (2) nodejs1, (3) server2, (4) micro2, (5) micro1.

The last phase is the extension of the graph with extra edges, which are automatically inferred by transitively closing host relationships, and by observing that if two components are connected, then their containers have to be connected (to allow such components to communicate).
inference process to the graph in Figure 3.9, we obtain the graph showed in Figure 3.10.

It is important to point out that the obtained graph representation is not intended to replace the original TOSCA model. It is rather a compact way to keep information that will be used during the deployment operations, which preserves all information that was present in the original TOSCA model.

### 3.3.2 Docker interface module

The **Docker interface** is the module in charge of talking with the Docker engine. It implements functions to manage containers, volumes and virtual networks.

This is a common interface to manage all the Docker components and do operations with them. This is manly used by the managers to create, start and stop containers or to execute commands inside running containers. The most interesting operation implemented by this module is the creation of a Docker container. This operation mimics the create operation of the Docker client, that is not available by using the Docker API. This operation goes through a set of steps:

1. It builds the image if there is a Dockerfile,
2. if not, it checks if the image exists locally,
3. if not, it downloads through the Dockerhub or another repository,
4. after that the image is used to create the container.

The **Docker interface** module also deals with all errors generated by the Docker engine during the execution of the operations, and it filters them by forwarding only those causing the failure of the deployment of an application. For example, if during the creation of an image an error says that another image with the same name already exists then the procedure automatically deletes this image and tries one more. In this way the deployment can continue.

### 3.3.3 Manager modules

The **Container manager**, **Software manager** and **Volume manager** implement the lifecycle for Docker containers, software components, and volumes, respectively.
The *Container manager* is in charge of implementing the lifecycle of the Docker containers through the *create*, *start*, *stop* and *delete* operations, by also taking into account that the components of type *tosker.nodes.Container* have to be processed differently from those of type *tosker.nodes.Container.Executable*. The main differences reside in the implementation of the operations *create* and *start*, while the other two operations are implemented similarly. The differences come from the fact that *create* and *start* are executed only if the type of the container is *tosker.nodes.Container.Executable*, since the latter is the type of container actually running a command/software. The type *tosker.nodes.Container* instead models whose only purpose is to host software, which means that their creation only requires to retrieve their image (locally, remotely, or by building it). Such image will be run only later, when the container will actually be used to host the software components deployed on it.

The *Software manager* implements the routine that performs the execution of the TOSCA standard lifecycle of the *software* components. It uses the module *Docker interface* to run commands on the container and to manage the execution of the implementation script of the software node. These operations are the more sophisticated because they have to interact with the container to install and run software inside. The *create* one for example goes thought the following steps:

1. The artefacts and the interface scripts are copied inside the container\(^5\)

2. the container starting command is modified with the creation script of the software\(^6\)

3. the container is started and the process waits for it,

4. the container is committed as new image,

5. the container is created again with the original starting command.

At the end of this procedure we have the container ready with all the artefacts inside and the software installed. This procedure is similar to the one used when a Docker image is built from a Dockerfile. Indeed, every line of the Dockerfile coincides to the execution of the command inside the container and

\(^5\)The actual implementation of this phase can be found in Chapter 4.

\(^6\)In Chapter 4 there are more details on the command generation and in the injection of the TOSCA inputs.
commits those changes on the image. With this mechanism also in TosKer it is created a set of levels inside the container one for every software installed on it.

The configure operation is implemented in the exact same way of the create one. Instead, the start, stop and delete operations are implemented by executing the interface script inside the container. The implementation of those operations is different if the container is running or not. Indeed, if the container is already running for example because it is a tosker.nodes.Container.Executable or because it is a tosker.nodes.Container but there is already software running on it, a Docker exec operation is used. Instead, if the container is not running (because it is of type tosker.nodes.Container) the default command of the container is overridden and the container is created again by putting as starting command the interface script of the software.

The Volume manager implements the routine that performs the execution of the lifecycle of Docker Volumes. These components, as specified by the Docker documentation, can only be created or deleted. Therefore, the Volume manager implement only the create and delete operations by invoking those operations on the Docker Engine, through the Docker interface.

3.3.4 Orchestrator module

The Orchestrator is the main component of the architecture, as it is in charge of orchestrating the lifecycle of the application described in TOSCA. It takes as input a TOSCA file and parses it with the TOSCA utility module, and it implements the procedure to create, start, stop and delete the application by using the Software manager, the Volume manager and the Docker interface module.

The deployment phase starts after the creation of the graph structure of the components and the identification of the deployment order, done by the TOSCA utility module. Each deployment operation is implemented by executing the same operation on each component by following the deployment order. For example, the implementation of the create operation is shown in Algorithm 1. The orchestrator invokes the create operation of each component, by exploiting the Container manager if the components is a Docker container, the Volume manager if the components is a Docker volume, and the Software manager if the components is a software component. The implementation of
the other operations on the application are similar to the create one, but for the order in which the components are scanned.

```
Algorithm 1: Create operation of the Orchestrator.

for i=0 to deploy_order.length do
    node = deploy_order[i];
    if node is Container then
        ContainerManger.create(node);
    end
    if node is Volume then
        VolumeManager.create(node);
    end
    if node is Software then
        SoftwareManager.create(node);
        SoftwareManager.configure(node);
    end
end
```

Indeed, for the stop and delete operation the reverse order of deployment is used, because in those phases the application must be teared down and not created. The orchestrator is also in charge of merging the lifecycle of each component, is because they do not have all the same operations. The software also features the configure operation, that has to be executed after the creation. The 

\texttt{tosker.nodes.Volume} has only the create and delete operations. To complete the description of the Orchestrator we show in Algorithm 2, 3, 4 the pseudocode of the other operations.

Finally, it is important to highlight that each operation on the components of the application assumes a given state of the components. As show in Figure 3.11 there are three main states (Initial, Created, Started) and in each state only some operations can be performed. Indeed, starting from the Initial state is possible to use the create operation to go in the Created state. From this state is possible with the delete operation to go back to the Initial state instead with the start operation is possible to go in the Started state. From this last state is only possible to go back to the Created state with the stop operation. If this order is not respected an error has to be generated and the deployment cannot be performed.
CHAPTER 3. DESIGN OF TOSKER

![State Graph](image)

**Figure 3.11:** State graph.

**Algorithm 2:** Start operation of the Orchestrator.

```plaintext
1 for i=0 to deploy_order.length do
2     node = deploy_order[i];
3     if node is Container then
4         ContainerManger.start(node);
5     end
6     if node is Software then
7         SoftwareManager.start(node);
8     end
9 end
```

**Algorithm 3:** Stop operation of the Orchestrator.

```plaintext
1 for i=deploy_order.length to 0 do
2     node = deploy_order[i];
3     if node is Container then
4         ContainerManger.stop(node);
5     end
6     if node is Software then
7         SoftwareManager.stop(node);
8     end
9 end
```

**Algorithm 4:** Delete operation of the Orchestrator.

```plaintext
1 for i=deploy_order.length to 0 do
2     node = deploy_order[i];
3     if node is Container then
4         ContainerManager.delete(node);
5     end
6     if node is Volume then
7         VolumeManager.delete(node);
8     end
9     if node is Software then
10        SoftwareManager.delete(node);
11 end
12 end
```
Chapter 4

Implementation of TosKer

We now describe a prototype implementation of TosKer, by also detailing the main choices that we made during its development. The source code of the described TosKer prototype is publicly available on GitHub under the MIT licence at https://github.com/di-unipi-socc/tosKer.

4.1 Language and libraries

TosKer is mainly written in Python, as shown in Table 4.1. Python has been chosen because is simply and complete, a perfect programming language for rapid prototyping. Furthermore, the TosKer project uses two open source Python libraries: docker-py [22] and tosca-parser [35]. The first one implements a Python interface for the Docker engine API, and it has recently become an official Docker library, maintained by Docker developers. The tosca-parser [35] library instead implements a parser for TOSCA YAML files and CSAR archives, and it is developed and maintained by OpenStack community.

<table>
<thead>
<tr>
<th>Language</th>
<th>files</th>
<th>blank line</th>
<th>commented line</th>
<th>code line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>25</td>
<td>316</td>
<td>93</td>
<td>1305</td>
</tr>
<tr>
<td>YAML</td>
<td>15</td>
<td>121</td>
<td>16</td>
<td>873</td>
</tr>
<tr>
<td>Bourne Shell</td>
<td>33</td>
<td>26</td>
<td>3</td>
<td>165</td>
</tr>
<tr>
<td>JavaScript</td>
<td>5</td>
<td>21</td>
<td>27</td>
<td>143</td>
</tr>
<tr>
<td>JSON</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>SUM:</td>
<td>80</td>
<td>484</td>
<td>139</td>
<td>2511</td>
</tr>
</tbody>
</table>

Table 4.1: Code statistics of TosKer.

Unfortunately, tosca-parser is not yet a stable library, and we incurred in
some unexpected errors due to features not yet implemented. However, using this library simplifies the implementation of TosKer, mainly for TOSCA validation features. The most relevant errors that we found are in the validation of the \texttt{get\_attribute} function, in the validation of artefacts, and in the validation of the relationship definitions. We reported all of those errors to the repository’s maintainers\textsuperscript{1}. In the wait of a new official release, we decided to patched the source code of TosKer with some workarounds, to let it work with the current release.

Furthermore, \texttt{tosca-parser} partially implements the execution of TOSCA functions. Indeed, \texttt{tosca-parser} executes just some type of functions and only if such functions are placed in specific fields of the TOSCA definition file. We fixed this behaviour in \textit{TOSCA utility} module by searching and running not executed functions (see Section \ref{sec:tosca-utility}).

4.2 Code structure

The project, as shown by Figure 4.1, is composed by the following components: A main Python module, called \texttt{tosker}, a set of Python files, and three others sub-modules. Each component of the TosKer architecture (see Chapter 3) is implemented by a dedicated Python class. Indeed, we have:

- \texttt{orchestrator.py}, which implements the \textit{Orchestrator} module,
- \texttt{docker\_interface.py}, which implements the \textit{Docker interface} module,
- \texttt{tosca\_utility.py}, which implements \textit{TOSCA utility} module,
- \texttt{container\_manager.py}, \texttt{volume\_manager.py}, and \texttt{software\_manager.py}, which implement the three \textit{Managers} modules.

Others important components of TosKer implementation are the \texttt{graph} and \texttt{tests} Python modules. The first one contains the implementation of the deployment graph described in Chapter 3. The \texttt{tests} Python module instead contains the Python unit tests and the resources used for validation TosKer\textsuperscript{2}.

We also have \texttt{helper.py} that contains utility functions and a logging system used by TosKer components, and \texttt{shell.py} that implements the command

\textsuperscript{1}The submitted bugs can be found at https://bugs.launchpad.net/tosca-parser/+bug/1622573 and at https://bugs.launchpad.net/tosca-parser/+bug/1643611.

\textsuperscript{2}the complete list of files and tests can be found in Section 4.5.
4.3 Implementation details

In this section, we detail the implementation of the most interesting parts of the TosKer engine. In particular, we discuss the implementation of the TOSCA functions, of the deployment graph, of the Software deployment, and of the networking of an application’s components.

4.3.1 TOSCA functions

TOSCA provides four functions, called get_input, get_attribute, get_property, and get_artifact, which can be used as arguments for the fields in the definition of a node template. The functions are usually used in the interface’s inputs declaration or in the property definition, to avoid value repetition.

The execution of these functions, as we highlighted in Section 4.1, is not
CHAPTER 4. IMPLEMENTATION OF TOSKER

completely covered by the tosca-parser. Indeed, there are cases where functions are executed and evaluated, and other cases in which the functions are ignored. For example the get_input function is evaluated when it is in the interface inputs of a node, but it is ignored when it is in the property definition. To fix the aforementioned behaviour, after the tosca-parser validation, a procedure is run to execute the ignored functions. The Listing 4.1 shows the procedure that is executed on each TOSCA node to evaluate the get_input, get_property, and get_artifact functions. Instead, get_attribute function is evaluated as soon as its value is needed, because the value of an attribute can change at runtime.

Listing 4.1: Recursive search and execution of TOSCA functions inside the definition of a node.

```python
1 def rec_parse_functions(name, node):
2     
3     # This function returns the result of the TOSCA function
4     def execute_function(value, args):
5         if 'SELF' == args[0]:
6             args[0] = name
7             return helper.get_attributes(args[1:], tpl[args[0]][value
8     
9     for k, v in node.items():
10         # If the function is already parsed by toscaparser, then use the result
11         if isinstance(v, toscaparser.functions.Function):
12             node[k] = v.result()
13         elif type(v) is dict:
14             
15         # Found a get_property function
16         if 'get_property' in v:
17             node[k] = execute_function('properties', v['
18         get_property'])
19
20         # Found a get_artifact function
21         elif 'get_artifact' in v:
22```

31
CHAPTER 4. IMPLEMENTATION OF TOSKER

```python
art = execute_function('artifacts', v['get_artifact'])
node[k] = _parse_path(base_path, art)

# Found a get_input function
elif 'get_input' in v:
    if v['get_input'] in inputs:
        node[k] = inputs[v['get_input']]  
    else:
        node[k] = tosca.inputs[v['get_input']]['default']
else:
    rec_parse_functions(name, v)
```

4.3.2 Deployment Graph

The deployment graph is implemented through the files template.py and nodes.py of the python module graph. In template.py there is the Template class, which represents the deployment graph. As shown in Listing 4.2, the Template class contains two main properties: nodes, and deploy_order. The first one is a dictionary with all the components of an application, while the second one defines the deployment order of such components. There are other properties, which generate a filter in the deploy_order list to retrieving only certain types of components (e.g., container_order permits retrieving the deployment order of the components of type Container).

Listing 4.2: The Template class.

```python
from .nodes import Container, Software, Volume

class Template:
    def __init__(self, name):
        self.name = name
        self._nodes = {}
        self.deploy_order = []
```
self.outputs = []

@property
def container_order(self):
    return (i for i in self.deploy_order if isinstance(i, Container))

@property
def volume_order(self):
    return (i for i in self.deploy_order if isinstance(i, Volume))

@property
def software_order(self):
    return (i for i in self.deploy_order if isinstance(i, Software))

def push(self, node):
    self._nodes[node.name] = node
    self.deploy_order.append(node)

def __getitem__(self, name):
    return self._nodes.get(name, None)

def __str__(self):
    return ', '.join((i.name for i in self.deploy_order))


nodes.py instead defines the four possible types of nodes: Root, Container, Software, and Volume. Each class contains the attributes and requirements of the corresponding TOSCA types, along with some additional information useful for the deployment. To distinguish the attributes in the corresponding TOSCA definitions, each class has the names of such attributes inside a private ATTRIBUTE dictionary. The latter can be retrieved using square bracket dictionary operator (e.g., to get the id attribute container["id"]).

The Root class (Listing 4.3) is the parent of the other classes and it defines the public property name. The Container class (Listing 4.4) stores the
relationship information in the following properties depend, connection, and volume. The latter correspond respectively to the TOSCA requirements dependency, connect and storage. In this class there is the property persistent, that is True if the container type is tosker.nodes.Container.Executable, and False if it is tosker.nodes.Container.

Listing 4.3: Constructor and properties of the Root node.

```python
class Root(object):
    def __init__(self, name):
        self.name = name
        self._ATTRIBUTE = {}

    def __getitem__(self, item):
        return self._ATTRIBUTE.get(item, lambda: None)()
```

Listing 4.4: Constructor and properties of the Container.

```python
class Container(Root):
    def __init__(self, name):
        super(self.__class__, self).__init__(name)
        # attributes
        self.id = None
        self.env = None
        self.cmd = None
        self.ports = None
        self._ATTRIBUTE = {
            'id': lambda: self.id,
            'ports': lambda: self.ports,
            'env_variable': lambda: self.env,
            'command': lambda: self.cmd
        }

        # artefacts
        self.image_name = None
```

34
self.tag_name = None
self.dockerfile = None

# flags
self.persistent = False

# requirements
self.depend = None
self.connection = None
self.volume = None

@property
def to_build(self):
    return self.dockerfile is not None

@property
def image(self):
    return '{}:{}'.format(self.image_name, self.tag_name)

@image.setter
def image(self, attr):
    self.image_name, self.tag_name = attr.split(':') if ':' in attr
    else (attr, 'latest')

The Software class (Listing 4.5) does not have attributes, but instead, a set of requirements stored as fields, such as depend, connection, host, and host_container. The first two properties have the same semantics as the Container class. The host property instead stores the pointer to the component which will host the software. The host_container stores the pointer to the Container that will host the software. This last property is added by the TOSCA utility when the deployment graph is enrichment with additional information (as shown in Section 3.3.1).

The Volume class (Listing 4.6) only contains the ATTRIBUTE map with id and size attributes.

Listing 4.5: Constructor and properties of the Software.
class Software(Root):

    def __init__(self, name):
        super(self.__class__, self).__init__(name)
        self.artifacts = None
        self.interfaces = {}

        # requirements
        self.host = None
        self.host.container = None
        self.depend = None
        self.connection = None

Listing 4.6: Constructor and properties of the Volume.

class Volume(Root):

    def __init__(self, name):
        super(self.__class__, self).__init__(name)
        self.id = None
        self.size = None

        self.__ATTRIBUTE = {
            'id': lambda: self.id,
            'size': lambda: self.size
        }

4.3.3 Software deployment

In this section we discuss some implementation details concerning the life-
cycle management of the software components. More precisely, we explain how
we have implemented the management of the artefacts and interface scripts,
of interface inputs, and of software installation.

An important step, to deploy a software component inside a Docker con-
tainer, is to copy inside the container, all the artefacts implementing the
management operations of the component’s management interface. This is
CHAPTER 4. IMPLEMENTATION OF TOSKER

done by exploiting a Docker Volume mounted on a temporarily folder in the host computer. By default, TosKer attaches to each container that it creates a volume inside the folder /tmp/dt (mounted on the host’s folder /tmp/tosker/<container name>). At this point, the software manager makes a folder, with the name of the software, inside this volume, and copy all the files inside. These operations are executed by the procedure in Listing 4.7, called by Software manager.

Listing 4.7: Procedure that copies files inside a Docker container.

```python
def _copy_files(self, node):
    # generate path for the tmp folder
    tmp = path.join(self._docker.tmp_dir,
                    node.host_container.name,
                    node.name)

    # create the folder for the software
    try:
        os.makedirs(tmp)
    except os.error:
        pass

    # copy all the interfaces scripts
    for key, value in node.interfaces.items():
        copy(value['cmd']['file_path'], tmp)

    # if present copy all the artefacts
    if node.artifacts:
        for key, value in node.artifacts.items():
            copy(value['file_path'], tmp)
```

Another important step, to deploy a software component inside a Docker container, is to generate the commands to manage the lifecycle of the software inside the container. Indeed, we recall that in a TOSCA definition, it is possible to have together with the definition of the implementation script, a set of optional inputs (i.e., TOSCA functions). To install correctly the software it is important to provide those inputs to the implementation script. In order
to simplify the writing of those scripts by the user, we chose to pass those parameters in two ways. Indeed, the inputs are passed either as environment variable with the name INPUT.<input name>, or as command argument (using the standard unix notation — e.g., --<input name> <input value>). To create a string with the command to be execute inside the container, and all the inputs, the Software manager uses the procedure in Listing 4.8. Finally, in Listing 4.9 it is possible to see the command to install the Software component of the Listing 4.9 generated by the previous function.

Listing 4.8: procedure that generates the implementation command to be executed inside a container.

```python
def _get_cmd_args(self, node, interface):
    def _get_inside_path(p):
        return path.join('/tmp/dt/', node.name, p['file'])

    if interface not in node.interfaces:
        return None

    args = []
    args_env = []
    res = None
    if 'inputs' in node.interfaces[interface]:
        for key, value in node.interfaces[interface]['inputs'].items():
            if type(value) is dict:
                value = _get_inside_path(value)
                args.append('−−{}{}'.format(key, value))
                args_env.append(
                    'export INPUT.{}={}'.format(key.upper(), value))
            else:
                res = "sh −c \"{}\sh {} \".format(
                    ;'.join(args_env),
                    _get_inside_path(node.interfaces[interface]['cmd']),
                    ;'.join(args)
                )
```

38
CHAPTER 4. IMPLEMENTATION OF TOSKER

res = 'sh {}.format(' + 
    _get_inside_path(node.interfaces[interface][\'cmd\'])),
    )

return res

Listing 4.9: A snippets of a TOSCA interface definition

 interfaces:
  Standard:
    create:
      implementation: scripts/app/install.sh
      inputs:
        code: { get_artifact: [ SELF, app_code ] }
        user: { get_input: user_name }

Listing 4.10: The command string that execute the interface of the Listing 4.9

sh -c 'export INPUT_CODE=/tmp/dt/code.zip; export INPUT_USER=master; sh /tmp/dt/install.sh --code /tmp/dt/code.zip --user test'

The software installation on top of a Docker container, as explained in Chapter 3, is done by using the commit operation provided in the Docker engine. The procedure that starts the container, installs the software component, and commits the changes (by also restoring the entrypoint of the container) is shown in Listing 4.11.

Listing 4.11: The procedure to add a new layer on top of Docker container.

def update_container(self, node, cmd):
    assert isinstance(node, Container)

    # copy the command and entrypoint of the image
    stat = self.inspect_image(node.image)
    old_cmd = stat[\'Config\'][\'Cmd\'] or None
    old_entry = stat[\'Config\'][\'Entrypoint\'] or None
CHAPTER 4. IMPLEMENTATION OF TOSKER

```
# delete the container if it exists
if self.inspect_container(node):
    self.stop_container(node)
    self.delete_container(node)

# create a new container with the new command
self.create_container(node,
    cmd=cmd,
    entrypoint=''
)

# start and wait the container
self.start_container(node.id, wait=True)
self.stop_container(node.id)

# commit the changes
self._cli.commit(node.id, self.get_saved_image(node))

# create again the container with the old command and entrypoint
self.delete_container(node)
self.create_container(node,
    cmd=node.cmd or old_cmd,
    entrypoint=old_entry,
    from_saved=True)

# commit again the changes
self._cli.commit(node.id, self.get_saved_image(node))
```

### 4.3.4 Networking

The networking of the components of an application deployed with TosKer is implemented by exploiting Docker container networking [19]. This feature permits creating container networking with the Docker API, and connecting containers by specifying the network name during the container creation. This
CHAPTER 4. IMPLEMENTATION OF TOSKER

Figure 4.2: Schema of all possible network connections starting from a software component.

operation connects the container together, and it is also possible to specify links to other containers on the network, by providing their names. The latter are used by the Docker engine to update the internal DNS servers of the container networking. Hence, the container can communicate with other containers by indicating only their name. Moreover, it is possible to add an alias for each one of these links, in this way a new entry with such name is added to the DNS.

The above mentioned aliasing feature is exploited by TosKer to set up connections among container and software components. More precisely, TosKer sets aliases to permit accessing software components by directly referring their names, even if links are actually connecting the containers hosting such components. To do so, TosKer must consider all possible cases, which are summarised in Figure 4.2 (where the only “real” link is between the containers C and D):

1. when a container C requires a link to a software components B, TosKer creates a link from C to D with tan alias that permits referring the container D as B.

2. when a software component A requires a link to a container D, TosKer creates a link from C to D.

3. when a software component A requires a link to another software component B, TosKer creates a link from C to D with tan alias that permits referring the container D as B.

The above operations are performed during the enrichment phase done by the TOSCA utility module on the deployment graph. In this phase, as already
CHAPTER 4. IMPLEMENTATION OF TOSKER

mentioned in Chapter 3, the deployment graph is updated with additional information. In Listing 4.12 we show the procedure that scans all components and adds new links and aliases as explained before. After the execution of this procedure each container stores, in addition to the links required by the container itself, all links required by the software that it hosts.

Listing 4.12: Procedure that adds links (and aliases) among containers.

```
def post_computation(tpl):
    # Manage the case where a Software is connected
    # to a Container or a Software
    for node in tpl.software_order:
        if node.link is not None:
            for link, _ in node.link:
                if isinstance(tpl[link], Container):
                    container_name = tpl[link].name
                if isinstance(tpl[link], Software):
                    container_name = tpl[link].host_container.name
                node.host_container.add_link((container_name, link))

    # Manage the case where a Container is connected to a Software
    for node in tpl.container_order:
        if node.link is not None:
            for i, (link, _) in enumerate(node.link):
                if isinstance(tpl[link], Software):
                    container_name = tpl[link].host_container.name
                node.link[i] = (container_name, link)
```

4.4 Packaging of TosKer

TosKer can be used both as a Python library or as command line software. TosKer has been published on the Python Package Index (PyPI) the official

---

3In the Docker API to specify a link with an alias, it is possible to provide a tuple (container_name, alias), instead of just the container name.

4https://pypi.python.org/pypi/tosKer
software repository of Python, which allows the installation and the distribution of Python libraries. This choice has been guided by the fact that TosKer is all written in Python, which makes PyPI the natural way to distribute it.

For what concerns the distribution of the TOSCA types we defined, we chose to create a repository in GitHub\footnote{https://di-unipi-socc.github.io/tosker-types}. In this way it is possible to import inside the definition of TOSCA application a specific type version. Naturally, this version must match to the TosKer engine version used.

\section{Testing}

To validate TosKer we developed four examples and twelve units tests, as it is shown in Figure 4.3. The examples are the following:

- \textit{node-mongo}, a web application written in nodejs connected to a database mongoddb. The application exposes a REST API to retrieve the information mongoddb stored inside.

- \textit{software-link}, an application composed of three servers written in nodejs. The application exposes a GET operation that returns the combination of the results of the same request forwarded to the other two servers.

- \textit{thoughts}, complete case study (which will be discussed in Chapter 5).

- \textit{wordpress}, a Wordpress application composed of a php webserver that runs Wordpress and a mySQL database.

Based on those example we created the following units tests:

- \texttt{test\_tosca\_csar.py}, which tests the feature to get a CSAR as input and orchestrate the application.

- \texttt{test\_tosca\_dockerfile.py}, which tests the feature of create a Docker container from a Dockerfile.

- \texttt{test\_tosca\_node\_mongo\_mix1.py}, which tests a network connection from a software component to a Docker container.

- \texttt{test\_tosca\_node\_mongo\_mix2.py}, which tests a network connection from a Docker container to a software component.
CHAPTER 4. IMPLEMENTATION OF TOSKER

tests
    __test_tosca_base.py
    __test_tosca_csar.py
    __test_tosca_dockerfile.py
    __test_tosca_node_mongo_mix1.py
    __test_tosca_node_mongo_mix2.py
    __test_tosca_node_mongo.py
    __test_tosca_node_mongo_single_server.py
    __test_tosca_software_lifecycle.py
    __test_tosca_software_link.py
    __test_tosca_thoughts.py
    __test_tosca_wordpress_light.py
    __test_tosca_wordpress.py
    __test_tosca_wordpress_volume.py
    __test_tosca_hello.py

TOSCA
    __csar
        __...
    __dockerfile
        __...
    __node-mongo
        __...
    __software-lifecycle
        __...
    __software-link
        __...
    __thoughts-app
        __...
    __wordpress-light.yaml
    __wordpress-volume.yaml
    __wordpress.yaml
    __hello.yaml

Figure 4.3: The directory structure of the TosKer tests.

- **test_tosca_node_mongo.py**, which tests a network connection from a software component to another software component.

- **test_tosca_node_mongo_single_server.py**, which tests a network connection between two software components that are hosted on the same container.

- **test_tosca_software_lifecycle.py**, which tests the lifecycle of a software component.
CHAPTER 4. IMPLEMENTATION OF TOSKER

Table 4.2: Coverage of the test executed on TosKer

<table>
<thead>
<tr>
<th>Module</th>
<th>statements</th>
<th>missing</th>
<th>coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>tosker/graph/nodes.py</td>
<td>94</td>
<td>9</td>
<td>90%</td>
</tr>
<tr>
<td>tosker/graph/template.py</td>
<td>20</td>
<td>1</td>
<td>95%</td>
</tr>
<tr>
<td>tosker/managers/container_manager.py</td>
<td>31</td>
<td>2</td>
<td>94%</td>
</tr>
<tr>
<td>tosker/managers/software_manager.py</td>
<td>68</td>
<td>4</td>
<td>94%</td>
</tr>
<tr>
<td>tosker/managers/volume_manager.py</td>
<td>12</td>
<td>2</td>
<td>83%</td>
</tr>
<tr>
<td>tosker/tosca_utility.py</td>
<td>211</td>
<td>16</td>
<td>92%</td>
</tr>
<tr>
<td>tosker/docker_interface.py</td>
<td>169</td>
<td>19</td>
<td>89%</td>
</tr>
<tr>
<td>tosker/orchestrator.py</td>
<td>94</td>
<td>17</td>
<td>82%</td>
</tr>
<tr>
<td>tosker/helper.py</td>
<td>69</td>
<td>38</td>
<td>45%</td>
</tr>
<tr>
<td>tosker/shell.py</td>
<td>72</td>
<td>72</td>
<td>0%</td>
</tr>
<tr>
<td>Total</td>
<td>841</td>
<td>180</td>
<td>79%</td>
</tr>
</tbody>
</table>

- test_tosca_software_link.py, which tests a network connection between three software components.
- test_tosca_thoghts.py, which tests the orchestration of the application Thoughts.
- test_tosca_wordpress_light.py, which tests the orchestration of a Wordpress container connected to a mySQL container, using a compact TOSCA file.
- test_tosca_wordpress.py, which tests the orchestration of a Wordpress container connected to a mySQL container,
- test_tosca_wordpress_volume.py, which tests the orchestration of a Wordpress application composed of two container wordpress and mySQL, and a Docker volume attach to the mySQL container.
- test_tosca_hello.py, which tests the orchestration of Docker container.

Table 4.2 shows the coverage of the previous unit tests on the TosKer Python files. In particular each row contains a file name, the total number of statements (without blank line), the number of statement missed by the tests and the percentage of code tested. Moreover, it is possible to see that all files have a coverage of more than 80%, but for helper.py and shell.py, which has less than 50%.
Chapter 5

TosKer at work

In this chapter we report a brief user guide of TosKer, where we show how to install and use the command line interface of TosKer. We then describe a case study of a multi-component application and show how to orchestrate it using TosKer.

5.1 How to install TosKer

Before starting the installation make sure that you match the following software requirements:

- python >= 2.7
- docker >= 1.8

TosKer is available on the PyPI index and can be installed on Linux using pip package manager with the following command:

```
# pip install tosker
```

To run the unit tests, TosKer must be installed from source, by doing the following:

1. download TosKer from the GitHub repository (e.g., git clone https://github.com/di-unipi-socc/tosKer),

2. run the following command inside the created folder to install TosKer from source:

---

1The “#” symbol means that the command must be executed as super-user.
# python setup.py install

3. start the tests with the command:

```bash
$ python setup.py test
```

## 5.2 How to use TosKer

The command line interface of TosKer has the following usage:

```
tosker FILE COMMANDS... [OPTIONS] [INPUTS]
tosker -h|--help
tosker -v|--version
```

**FILE:** TOSCA YAML file or CSAR file

**COMMANDS:**
- create   Create application components
- start    Start applications components
- stop     Stop application components
- delete   Delete application components (except volumes)

**OPTIONS:**
- `-h --help`  Print usage
- `-v --version`  Print version
- `-q --quiet`  Enable quiet mode
- `--debug`  Enable debugging mode (override quiet mode)

**INPUTS:** provide TOSCA inputs (syntax: `--NAME VALUE`)

The command line interface of TosKer has two mandatory parameters: a TOSCA file or a CSAR archive, and the operation to be performed on the application (viz., create, start, stop, delete). Flag `--debug` show debugging logs during the orchestration, while `-q` or `--quiet` prevents TosKer from writing ²

²The “$” symbol means that the command can be executed as user.
CHAPTER 5. TOSKER AT WORK

on the standard output. Finally, if the TOSCA file has defined inputs, it is possible to supply those inputs by the standard Unix way (i.e., --<input> <value>). For instance:

```bash
$ tosker app.csar create --password w5TtwvpC \
   --port 8000
```

creates application `app.csar`, setting the input variable `password` to `w5TtwvpC` and the input variable `port` to `8000`.

It is also possible to stack more than one operation together, for instance the command:

```bash
$ tosker app.csar create start
```

creates the application and then starts it.

5.3 Case study: The ThoughtsSharing application

In this section we describe how we orchestrate the ThoughtsSharing application [8] with TosKer. ThoughtsSharing is a simple web application where users can share thoughts and see thoughts that other users published. ThoughtsSharing is composed of three main components:

- `gui`, the web interface of ThoughtsSharing, written in Node.js,
- `api`, the software that implements the REST API used by the web application, implemented in Java, and
- `database`, a mongoDB database used by the `api` to store thoughts.

5.3.1 Modelling the application

To orchestrate the ThoughtsSharing application with TosKer, we must determine the Docker containers that are needed to run the application and the relationships between the components. We decide to use the `maven:3.3-jdk-8` and `node:6` Docker images to host respectively the `api` and the `gui` software, the `mongo:3.4` Docker image to implement the MongoDB database, and a Docker volume to store database data persistently. Figure [5.1] shows all of
those components and the relationships between them. In particular, it is possible to see that we use a \textit{DependsOn} relationship between gui and api, and a \textit{ConnectsTo} relationship between api and db\_container. In the first case because the api software must start before app, and in the second one because the api must communicate to the db\_container.

![Graphical representation of the ThoughtsSharing application.](image)

**Figure 5.1:** Graphical representation of the ThoughtsSharing application.

Finally, we create a CSAR archive, shown in Figure 5.2 with the TOSCA file that describes the application and all the scripts for the installation of the software components.
Listing 5.1 reports the complete TOSCA file that describes the application. The file is composed of four main parts:

- Repository definition (lines 5-6), where we define the Docker Hub repository that will be used later in the container node.

- Import of TosKer types (lines 8-9), which are the types used in the node_templates.

- Input definition (lines 12-28), those are the inputs used to configure at runtime the orchestration (e.g., the port where api and gui listen).

- Components description (lines 31-120):
  - api (lines 31-55), a node of type tosker.nodes.Software that describes the REST API application written in Java. The component requires to be hosted on maven_container and to be connected to db_container. Furthermore, the node includes the interfaces scripts to implement the lifecycle operations.
  - gui (lines 57-80), a node of type tosker.nodes.Software that describes the web-application implemented in JavaScript. The node
also requires to be hosted on node_container and to be dependent on api. It also includes the interfaces scripts to implement the lifecycle operations.

- maven_container (lines 82-91), a node of type tosker.nodes.Container that describes a container based on the Docker image maven:3.3-jdk-8. The node binds the internal port 8080 of the container to the host port in the api_port input variable.

- node_container (lines 93-102), a node of type tosker.nodes.Container that describes a container based on the Docker image node:6. The node binds the internal port 3000 to the host port in the gui_port input variable.

- db_container (lines 104-117), a node of type tosker.nodes.Container.Executable that describes a container based on the Docker image mongo:3.4. The node requires to mount volume db_volume on the folder /data/db inside the container.

- db_volume (lines 119-120), a node of type tosker.nodes.Volume, which describes a Docker volume.

LISTING 5.1: Thoughts TOSCA file descriptor

```yaml
tosca_definitions_version: tosca_simple_yaml_1_0
description: TOSCA description of the Thoughts application.

repositories:
  docker_hub: https://registry.hub.docker.com/

imports:
  - tosker: https://di-unipi-socc.github.io/tosker-types/0.0.6/tosker.yaml

topology_template:
  inputs:
    gui_port:
      type: integer
      default: 8080
      description: GUI port
```

51
CHAPTER 5. TOSKER AT WORK

```
    type: string
default: master
description: Branch of the GUI repository
api_port:
type: integer
default: 8000
description: API port
api_branch:
type: string
default: master
description: Branch of the API repository

node_templates:
  api:
    type: tosker.nodes.Software
    requirements:
      - host: maven_container
      - connection: db_container
    interfaces:
      Standard:
        create:
          implementation: scripts/api/install.sh
          inputs:
            repo: https://github.com/jacopogiallo/thoughts
        -api
          branch: { get_input: api_branch }
        configure:
          implementation: scripts/api/configure.sh
          inputs:
            dbURL: db_container
dbPort: 27017
dbName: thoughtsSharing
collectionName: thoughts
        start:
          implementation: scripts/api/start.sh
        stop:
          implementation: scripts/api/stop.sh
        delete:
          implementation: scripts/api/uninstall.sh

  gui:
    type: tosker.nodes.Software
    requirements:
```
- host: node_container
- dependency: api

interfaces:
  Standard:
  create:
    implementation: scripts/gui/install.sh
    inputs:
      repo: https://github.com/jacopogiallo/thoughts

    - gui
      branch: { get_input: gui_branch }
  configure:
    implementation: scripts/gui/configure.sh
    inputs:
      apiUrl: localhost
      apiPort: { get_input: api_port }
      apiResource: thoughts
  start:
    implementation: scripts/gui/start.sh
  stop:
    implementation: scripts/gui/stop.sh
  delete:
    implementation: scripts/gui/uninstall.sh

maven_container:
  type: tosker.nodes.Container
  properties:
    ports:
      8080: { get_input: api_port }
  artifacts:
    my_image:
      file: maven:3.3-jdk-8
      type: tosker.artifacts.Image
      repository: docker_hub

node_container:
  type: tosker.nodes.Container
  properties:
    ports:
      3000: { get_input: gui_port }
  artifacts:
    my_image:
      file: node:6
      type: tosker.artifacts.Image
CHAPTER 5. TOSKER AT WORK

```yaml
repository: docker_hub

db_container:
  type: tosker.nodes.Container.Executable
  artifacts:
    my_image:
      file: mongo:3.4
      type: tosker.artifacts.Image
      repository: docker_hub
  requirements:
    - storage:
      node: db_volume
        relationship:
          type: tosca.relationships.AttachesTo
          properties:
            location: /data/db

db_volume:
  type: tosker.nodes.Volume
```

5.3.2 Deploying and managing ThoughtsSharing

After creating the CSAR, ThoughtsSharing can be created, using the following command of the TosKer command line interface:

```bash
$ tosker thoughts.csar create --gui_port 80
```

whose output is reported in Figure 5.3. The command shows the deployment order determined by the internal algorithm (see Chapter 4) and then executes the requested operation on the components. If the operation is executed without errors a green tick is displayed near the component name and TosKer proceeds to deploy another component. After the execution of this command, TosKer will have created three containers using the Docker engine, which can be seen using the command `docker ps -a`, as shown in Figure 5.4. Those containers are: `maven_container`, which contains the installation and configuration of `api` component, `node_container`, which contains the installation and configuration of `gui` component, and finally `db_container`, which is a container based on the mongo image.

After creating the application it is possible to start it by the command

```bash
$ tosker thoughts.csar start --gui_port 80
```
CHAPTER 5. TOSKER AT WORK

Figure 5.3: Output of the create command.

Figure 5.4: Docker container status after TosKer created ThoughtsSharing application. The Docker container status after TosKer stopped ThoughtsSharing is similar.

which output is shown in Figure 5.5. This command has the same output of the create command, i.e. it shows first the deployment order and then the list of the component on which the operation is applied. After the execution of the start command TosKer starts the Docker container, which can be seen by using the command `docker ps -a` (Figure 5.6). At this point the container `node_container`, the one that contains the `gui` web-application, is listening on port 80 and it is accessible by a web-browser at http://127.0.0.1/thoughts.html,
as shown in Figure 5.5.

![Output of the start command.](image)

**Figure 5.5:** Output of the start command.

Finally, the application can be stopped with the command

```
$ tosker thoughts.csar stop --gui_port 80
```

and deleted with the command

```
$ tosker thoughts.csar delete --gui_port 80
```
Those commands have the same output of the previous ones, as it can be seen in Figure 5.8 and Figure 5.9, but for the ordering of the operations. Indeed, both commands stop/delete gui and api software at first. After the execution of the delete command all the containers are eliminated as shown in Figure 5.10.

It is important to highlight that it is possible to omit the gui\_port input parameter or specify other input parameters from the one available in the ThoughtsSharing TOSCA file (viz., gui\_port, gui\_branch, api\_port, api\_branch).
CHAPTER 5. TOSKER AT WORK

Figure 5.8: Output of the stop command.

Figure 5.9: Output of the delete command.
Figure 5.10: Docker container status after TosKer deleted the ThoughtsSharing application.
Chapter 6

Related work

In this chapter we discuss currently available solutions for orchestrating composite cloud application by exploiting Docker and/or TOSCA. We also illustrate how TosKer positions with respect to such solutions.

6.1 Docker Orchestrators

In this section we discuss Kubernetes [29], Mesos [5] and Docker compose [17], which are tools extending the orchestration capabilities of Docker by permitting to create and manage applications composed by multiple containers and volumes. Kubernetes [29] and Mesos [5] permit doing so on clusters of nodes, while Docker Compose [17] is instead intended to work only with single hosts.

Kubernetes [29] is an open source system for automating deployment, scaling and managing containerized applications. It is a high level cluster orchestrator, which is not only compatible with Docker, but also with rkt [36]. The main concepts behind Kubernetes [29] are cluster, pod and service. A cluster is a group of compute nodes, which constitute the runtime for applications. One of the nodes work as master, and it is in charge of orchestrating the deployment of the applications. A pod is the minimal unit that can be deployed on a node, and it can be composed of multiple containers and volumes. A service is an application that the user want to expose, and it is composed of one or more pods connected to the same network.

Mesos [5] is a Linux-based distributed system kernel, which runs on clusters of machines and provides an API for managing and scheduling the resources
in a cluster. Mesos \cite{Mesos} is based on partitioning the nodes of a cluster in two
different sets of agents, called master and worker. An application is then rep-
resented as a framework, composed by schedulers and executors. A scheduler
runs on master nodes and manages the resources of a framework. An executor
is instead a process launched on agents, whose purpose is to execute a set
of tasks. Mesos \cite{Mesos} natively supports Dockers, as it is possible to run Docker
containers as executors or tasks. In the first case, only one container is instan-
tiated, and the tasks are executed within it. In the second case, it is created a
new container for each task.

Docker Compose \cite{DockerCompose} is a tool for defining and running multi-container
Docker applications. It permits describing a multi-container application in a
(non-standard) YAML file. In the description file it is possible also to specify
the links to be set between the containers forming the application, and the
volumes to mounted by each container. Based on that, Docker Compose \cite{DockerCompose}
is capable of deploying the specified application on a single host. It requires to
be used in conjunction with Docker Machine \cite{DockerMachine} and Docker Swarm \cite{DockerSwarm} to
permit deploying applications over clusters of multiple Docker hosts.

In summary, all aforementioned tools focus on cluster orchestration rather
than on application orchestration. Indeed, all of them manage cluster of nodes
and deploy application (composed of Docker components) on such a cluster.
Instead, our main objective, that is orthogonal to TOSCA, is to improve appli-
cation description abstracting as much as possible from the infrastructure. For
this reason we decide to use Docker engine and focus on application description
using TOSCA rather than exploits solution with different objectives.

6.2 TOSCA Deployers

The reference implementation for TOSCA XML \cite{TOSCAXML} is provided by the
OpenTOSCA \cite{OpenTOSCA} open source ecosystem. It is composed of three parts: the
OpenTOSCA runtime environment, the Winery graphical editor for
TOSCA applications, and Vinothek self-service portal for retrieving avail-
able TOSCA applications. OpenTOSCA \cite{OpenTOSCA} differs from TosKer as it is de-
signed to work with a former, XML-based version of TOSCA \cite{TOSCAXML}, while TosKer
is an implementation of a newer, YAML-based version of TOSCA \cite{TOSCAYAML}. Also,
OpenTOSCA \cite{OpenTOSCA} process TOSCA applications imperatively, i.e., based on the
deployment/management plans (e.g., BPEL or BPMN workflows) defined by
the application developer. It however currently lacks the implementation of the declarative processing of TOSCA applications (i.e., it cannot automatically determine deployment/management plans based on the topology of the application). TosKer is instead capable of declaratively processing TOSCA applications.

Other projects worth mentioning are Alien4Cloud \cite{1} and SeaClouds \cite{12,37}, which provide engines for deploying TOSCA YAML applications. Alien4Cloud \cite{1} is a service manager for cloud based applications that aims at simplifying the specification and deployment of a cloud application in TOSCA with a modern web interface. The project is ongoing, but it already supports the description and processing of TOSCA-based applications, provided that such applications are described by exploiting the alphabet of TOSCA types defined by Alien4Cloud itself (as TosKer does with its own alphabet of types). To do so, it exploits Cloudify \cite{25}, and it hence suffers of its same limitations (which will be discussed in Section 6.3).

SeaClouds \cite{12,37} aims at working as middleware for deploying TOSCA-based application on multiple IaaS and PaaS cloud providers. To do so, it exploits Brooklyn, and it hence suffers of its same limitations (which will be discussed in Section 6.4).

In summary, all aforementioned tools share with TosKer the objective of deploying and orchestrating cloud-based application exploiting TOSCA standard. Although all of those solutions lack an integration with the Docker platform, which TosKer instead supports.

6.3 Cloudify

Cloudify \cite{25} is an open source cloud orchestrator, which is capable of deploying and monitoring applications. Cloudify \cite{25} exploits event-driven system, to trigger custom correction scripts and auto-scaling capabilities. It supports different cloud providers and infrastructures by a set of plugins, which exploits a common API to implement the operations to deploy and manage applications. One of those plugins is Docker \cite{3}, which permits employing Docker containers as nodes of an application.

The deployment of an application is described with a blueprint file that uses a syntax inspired by, but not fully-compliant with TOSCA \cite{33}. The two main differences between the blueprint language of Cloudify \cite{25} and TOSCA
are concerning the specification of relationship between nodes and of policies.

- In TOSCA, a relationship between two components is defined by interconnecting a requirement of one component to a capability of the other. Nodes can define interconnection constraints, e.g., by specifying the type of a capabilities that can satisfy their requirements. Application topologies have then to be built by satisfying such constraints, hence permitting to validate them \[10\]. Instead, in a Cloudify \[25\] blueprint, the requirements and capabilities of a node are substituted by a field called “relationships”, which permits indicating its dependencies on other components.

- With regard to policies, Cloudify \[25\] offers an imperative policy system, which is in contrast with the declarative policy system of TOSCA \[33\]. In TOSCA \[33\], each policy is described declaratively by the application developer, and it is a task of the TOSCA engine to determine and actuate the operations to guarantee the satisfaction of the policy. Instead, in the Cloudify \[25\] policy system, application developers have to imperatively specify the policy, by indicating the trigger and scripts indicating then and how to actuate the policy. Policy scripts must be written in Clojure \[26\], and they have to refer to as specific API of Cloudify.

**Listing 6.1: An example of a Cloudify blueprint**

```yaml
1 tosca_definitions_version: cloudify_dsl_1_3
2
3 imports:
4  - http://www.getcloudify.org/spec/cloudify/3.4/types.yaml
5
6 node_templates:
7  host:
8    type: cloudify.nodes.Compute
9    properties:
10       ip: localhost
11       install_agent: false
12
13  http_web_server:
14    type: cloudify.nodes.WebServer
15    properties:
16       port: 8080
```

63
CHAPTER 6. RELATED WORK

Listing 6.1 illustrate a simple Cloudify application, which is composed by a compute node and a web server. Within the description of the web server, we can observe the property relationships, which specifies the node where to install the web server (in a different way with respect to TOSCA).

In summary, Cloudify differs from TosKer since, despite it permits orchestrating multi-container Docker applications, it does so by exploiting a language inspired by, but not fully compliant with TOSCA.

6.4 Brooklyn, Clocker and Brooklyn-TOSCA

Brooklyn is a framework for modelling, monitoring, and managing applications through autonomic blueprints. Such blueprint permits specifying multiple aspects of cloud applications, from their structure to their desired QoS and auto-scaling policies. Brooklyn also includes a web-based GUI and a set of API to manage and orchestrate the applications.

The blueprint language employed by Brooklyn to specify applications based on the OASIS standard CAMP. The specification language is equipped with a set of predefined types that can be used as base building blocks to create the applications. The basic structure of a blueprint (an example of which is shown in Listing 6.2), is composed by the name of the specification file, a location that specifies on which cloud provides or infrastructures the application should be deployed, and a list of typed services to be deployed.

Listing 6.2: A Brooklyn blueprint example

```python
name: sample-single-jboss
description: Single JBoss using Brooklyn
origin: http://brooklyn.io
location: localhost
services:
  - serviceType: brooklyn.entity.webapp.jboss.JBoss7Server
```
CHAPTER 6. RELATED WORK

name: jboss1
brooklyn.config:
  wars.root: http://example.com/webapp-0.5.0.war
  http.port: 9280+

The Brooklyn [2], per se, is not directly related to our case, because it does not natively support TOSCA [33] or Docker [16]. The support for TOSCA and Docker is added in two independent extensions of Brooklyn, viz., Clocker [15] and Brooklyn TOSCA [13, 14].

Clocker [15] is a service running on top of Brooklyn that permits deploying a Docker-based application on a cluster. The application has to be described using an extension of the Brooklyn blueprint language, which permits indicating that a node is a Docker image. Essentially, Clocker integrates Brooklyn with Docker Swarm [23], by exploiting Brooklyn to create and manage Docker containers within the created clusters. Clocker is a pure extension of Brooklyn, which also gives the possibility to mix containers with virtual machines. However, Clocker does not permit indicating, which software container to install, which is something that is instead possible in TosKer (thanks to the fact that TosKer supports TOSCA).

Brooklyn-TOSCA [13] is an extension of Brooklyn that brings support for TOSCA Brooklyn. Brooklyn-TOSCA is not compatible with Clocker, as the latter extension the blueprint language of Brooklyn to support Docker. Hence, also with Brooklyn-TOSCA currently it is not possible to multi-container Docker applications specified in TOSCA. The latter is instead possible with TosKer.

6.5 Summary

In the previous sections we analysed the currently available solutions for orchestrating composite cloud application by exploiting Docker and/or TOSCA. We first focused on solutions working only with Docker, such as Kubernetes [29], Mesos [5] and Docker Compose [17] and then we analysed solutions working only with TOSCA, such as OpenTOSCA [27], Alien4Cloud [1] and SeaClouds [12, 37]. We highlighted how all these projects lack the possibility to combine TOSCA with Docker.

 Afterwards, we analysed Cloudify [25] and Clocker [15]. Both orchestrators are close to TosKer, as they use a declarative description language to describe
applications, and since they permit employing Docker containers as compute nodes in such applications. However, both Cloudify and Clocker do not support TOSCA, and they do not permit indicating the software components to be installed within containers.

At the best of our knowledge, TosKer is hence the first solution that permits specifying multi-container Docker applications in TOSCA, by also indicating, which software components have to be installed in a container, and how containers and software components are interconnected each other — satisfying all the requirements and offering all the functionalities discussed in Section 3.1.
Chapter 7

Conclusions

The problem of automatically deploying and managing complex multi-component applications over heterogeneous infrastructures is gaining more and more momentum in today’s IT \cite{30}. TOSCA \cite{33} and Docker \cite{16} are two emerging solutions trying to address this problem from different perspectives.

The objective of this thesis was to identify and develop a solution capable of trading off the pros and cons of TOSCA and Docker. For this reason we designed and prototyped TosKer, an orchestration engine capable of deploying, on top of Docker, multi-component applications described in TOSCA YAML. TosKer inputs the TOSCA description of a multi-component application, whose software components are hosted in Docker containers which may in turn mount Docker volumes to persist data. TosKer processes the given application description, and it interacts with the Docker engine to automatically deploy and orchestrate an instance of the application.

TosKer can be used both as a Python library or as a command line software. It is open source (under the MIT licence), and it is distributed through PyPI. We have already tested the capabilities of TosKer with unit tests (see Chapter \[4\]), and we have also run a concrete case study on it (see Chapter \[5\]). Additional case studies are however needed to further test and improve the orchestration capabilities of TosKer. This, together with the implementation of a graphical user interface for TosKer, is part of our future work.

To the best of our knowledge, TosKer is the first orchestration engine that permits orchestrating TOSCA applications combining heterogeneous components, including Docker containers and Docker volumes. TosKer hence advances the state-of-the-art for orchestrating applications with Docker and TOSCA,
as it extends the orchestrations capabilities of Docker by allowing it to orches-
trate generic software components, and as it extends the current support for
TOSCA by providing a Docker-based orchestration engine for TOSCA appli-
cations.

Although TosKer advances the current support for orchestrating multi-component
applications, it has also some known limitations. In particular:

- TosKer is designed to permit orchestrating applications on single host
  machines, and it currently does not include a support for deploying and
  managing applications on clusters of workstations or on external cloud
  services. We plan to include the possibility to deploy applications on
  clusters by exploiting Docker Swarm [23] or other cluster managers.

- TosKer does not permit specifying policies to apply to the multi-component
  application it orchestrates (e.g., auto-scaling policies, or failure recovery
  policies). As part of our future work, we also plan to add in TosKer a
  support for the policy system of TOSCA YAML.

Other interesting extensions of TosKer are:

- To improve TosKer in such a way it can automatically determine the
  Docker containers needed by the components forming an application.
  Intuitively, one may specify only the software components she needs in
  her application, along with the requirements they need to be effectively
  run. TosKer could then automatically look for the Docker containers (and
  volumes) offering capabilities that satisfy the components’ requirements,
  include such containers (and volumes) in the TOSCA description of the
  application, and then start orchestrating the automatically completed
  TOSCA application.

- To improve the orchestration capabilities of TosKer by supporting fault-
  aware management protocols [8]. Fault-aware management protocols per-
  mit modelling the management behaviour of application components,
  and automating the management of a multi-component application by
  combining the behaviour of its components according to the application
  topology. The extension of TosKer to support fault-aware management
  protocols is part of our immediate future work.


Appendices
Listing A.1: The TosKer Container type

tosca_definitions_version: tosca_simple_yaml_1_0

description: Definition of the custom types of TosKer.

node_types:
  tosker.nodes.Root:
    derived_from: tosca.nodes.Root

  tosker.nodes.Container:
    derived_from: tosker.nodes.Root

attributes:
  id:
    type: string
  private_address:
    type: string
  public_address:
    type: string

# BEGIN work around for the get_attribute bug in toscaparser
  ports:
    type: map
# END

properties:
  ports:
    type: map
APPENDIX A. TOSKER TYPES

required: false

requirements:
  - storage:
    capability: tosca.capabilities.Attachment
    occurrences: [0, UNBOUNDED]
    node: tosker.nodes.Volume
    relationship: tosca.relationships.AttachesTo

capabilities:
  host:
    type: tosca.capabilities.Container
    valid_source_types: [tosker.nodes.Software]
    occurrences: [0, UNBOUNDED]

Listing A.2: The TosKer Executable Container type

tosker.nodes.Container.Executable:
  derived_from: tosker.nodes.Container

attributes:
  # BEGIN work around for the get_attribute bug in toscaparser
  env_variable:
    type: map
  command:
    type: string
  # END

properties:
  env_variable:
    type: map
    required: false
  command:
    type: string
    required: false

requirements:
  - connection:
    capability: tosca.capabilities.Endpoint
    occurrences: [0, UNBOUNDED]
    node: tosker.nodes.Root
    relationship: tosca.relationships.ConnectsTo
APPENDIX A. TOSKER TYPES

- dependency:
  capability: tosca.capabilities.Node
  occurrences: [0, UNBOUNDED]
  node: tosker.nodes.Root
  relationship: tosca.relationships.Dependson

  capabilities:
  endpoint:
    type: tosca.capabilities.Endpoint
    valid_source_types: [tosker.nodes.Software, tosker.nodes.Container.Executable]
    occurrences: [0, UNBOUNDED]
  feature:
    type: tosca.capabilities.Node
    valid_source_types: [tosker.nodes.Software, tosker.nodes.Container.Executable]
    occurrences: [0, UNBOUNDED]

Listing A.3: The TosKer Volume type

tosker.nodes.Volume:
  derived_from: tosker.nodes.Root

  attributes:
  id:
    type: string
    # BEGIN workaround for the get_attribute bug in toscaparser
  size:
    type: string
    # END

  properties:
  # set the size. For example: 100m
  size:
    type: string
    required: false

  capabilities:
  attachment:
    type: tosca.capabilities.Attachment
    valid_source_types: [tosker.nodes.Container, tosker.nodes.Container.Executable]
APPENDIX A. TOSKER TYPES

occurrences: [0, UNBOUNDED]

Listing A.4: The TosKer Software type

tosker.nodes.Software:
derived_from: tosker.nodes.Root

requirements:
  - connection:
    capability: tosca.capabilities.Endpoint
    occurrences: [0, UNBOUNDED]
    node: tosker.nodes.Root
    relationship: tosca.relationships.ConnectsTo
  - dependency:
    capability: tosca.capabilities.Node
    occurrences: [0, UNBOUNDED]
    node: tosker.nodes.Root
    relationship: tosca.relationships.DependsOn
  - host:
    capability: tosca.capabilities.Container
    occurrences: 1
    node: tosker.nodes.Container
    relationship: tosca.relationships.HostedOn

capabilities:
  endpoint:
    type: tosca.capabilities.Endpoint
    valid_source_types: [tosker.nodes.Software, tosker.nodes.Container.Executable]
    occurrences: [0, UNBOUNDED]
  feature:
    type: tosca.capabilities.Node
    valid_source_types: [tosker.nodes.Software, tosker.nodes.Container.Executable]
    occurrences: [0, UNBOUNDED]
  host:
    type: tosca.capabilities.Container
    valid_source_types: [tosker.nodes.Software]
    occurrences: [0, UNBOUNDED]

Listing A.5: The TosKer Artifacts types
APPENDIX A. TOSKER TYPES

```
tosker.artifacts.Root:
  derived_from: tosca.artifacts.Root

tosker.artifacts.Image:
  derived_from: tosker.artifacts.Root
  description: a Docker image

tosker.artifacts.Dockerfile:
  derived_from: tosker.artifacts.Root
  description: a Dockerfile
```