TosKer: Orchestrating Applications with TOSCA and Docker

Antonio Brogi, Luca Rinaldi, and Jacopo Soldani(✉)

Department of Computer Science, University of Pisa, Pisa, Italy
soldani@di.unipi.it

Abstract. Docker is emerging as a simple yet effective solution for deploying and managing multi-component applications in virtualised cloud platforms. Application components can be shipped within portable and lightweight Docker containers, which can then be interconnected to allow components to interact each other. At the same time, the need for an enhanced support for orchestrating the management of the application components shipped within Docker containers is emerging.

In this paper we show how TOSCA can be exploited to provide such an enhanced support, by proposing a representation for describing the components forming an application, as well as the Docker containers used to ship such components. We also present TosKer, an engine for orchestrating the management of multi-component applications based on the proposed TOSCA representation and on Docker.

1 Introduction

Cloud computing has revolutionised IT, by allowing to run on-demand distributed applications at a fraction of the cost which was necessary just a few years ago [3]. This is possible as cloud providers exploit virtualisation techniques to achieve elasticity of large-scale shared resources [22]. Container-based virtualisation (where the operating system kernel permits running multiple isolated guest instances, called *containers*) can thus play an important role for cloud platforms, especially because it provides a lightweight virtualisation framework for PaaS/edge clouds [19,30] Applications can be packaged, along with all software dependencies they need to run, into portable and lightweight containers, which can then be managed on cloud platforms [28].

Containers are also an ideal solution for SOA-based architectural patterns (e.g., microservices [24]) that are emerging in the cloud community to decompose monolithic applications into suites of independently deployable, lightweight components. Application components can indeed be packaged in independently deployable, lightweight containers, which can then be interconnected to allow components to interact with each other (forming multi-container applications [31]).

Docker [14] is considered the de-facto standard for container-based virtualisation [29]. Docker permits packaging software components in Docker *images*,

Z. Á. Mann and V. Stolz (Eds.): ESOCC 2017, CCIS 824, pp. 130–144, 2018.
https://doi.org/10.1007/978-3-319-79090-9_9

which are then exploited as read-only templates to create and run Docker *containers*. Docker containers can also mount external *volumes*, which ensure data persistence independently of the lifecycle of containers [23].

Docker permits orchestrating containers, by allowing to define multi-container Docker applications [31]. Given (the images of) the containers forming a multi-container application, the volumes they must mount, and the connections to set up among containers, Docker compose [15] is indeed capable of automatically deploying the corresponding application.

Docker containers are however treated as "black-boxes", and they constitute the minimum orchestration entity considered by currently existing approaches for orchestrating multi-component applications with Docker (e.g., [2,15,16,32]). Application components must be manually packaged, along with all their software dependencies, in (images of) Docker containers. Components are then strictly bound to their hosting containers, as it is not possible to orchestrate the management of the components forming an application independently of the Docker containers hosting them. For instance, it is not possible to run only some of the components hosted on a container, as whenever a container is started, all components it hosts are also started. Also, if we wish to change the container used to host a component, new Docker images must be manually developed (e.g., if a `maven` container is hosting the front-end and back-end of an application, and we wish to move the front-end to a `java` container, we must develop two new Docker images, one for hosting the front-end on a `java` container and one for hosting *only* the back-end on a `maven` container).

To fully exploit the potential of SOA, the current support for orchestrating multi-component applications with Docker should be enhanced. A concrete solution is to still rely on Docker containers as a portable and lightweight mean to deploy application components on cloud platforms, by also allowing to independently manage the components and containers forming a multi-component application [28]. In this paper we propose a solution precisely following this idea, which relies on the OASIS standard TOSCA [27] as the mean for orchestrating multi-component applications on top of Docker containers.

- We propose a TOSCA-based representation for multi-component applications, which permits modularly specifying the components forming an application, the Docker containers and Docker volumes needed to run them, as well as the relationships occurring among them (e.g., a component is hosted on a container, a component connects to another).
- We also present TosKer, an engine for orchestrating the management of multi-component applications based on the proposed TOSCA representation and on Docker.

Our approach enhances the current support for orchestrating the management of multi-component applications in Docker, as it considers application components as orchestration entities, which are independent from the Docker containers and Docker volumes used to build their runtime infrastructure. For instance, TosKer allows to independently manage the application components hosted on a Docker container, hence allowing to run only some of them

(if needed). Our approach also eases the change of Docker containers used to host the components of an application, as this only requires to update the corresponding TOSCA specification[1] (which will then be processed by TosKer to automatically deploy and manage the specified application).

The rest of the paper is organised as follows. Section 2 provides some background on TOSCA. Section 3 illustrates our proposal for specifying multi-container Docker applications in TOSCA, and Sect. 4 presents the TosKer engine for actually orchestrating such applications. Finally, Sects. 5 and 6 discuss related work and draw some concluding remarks, respectively.

2 Background

TOSCA (*Topology and Orchestration Specification for Cloud Applications* [27]) 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 paper[2].

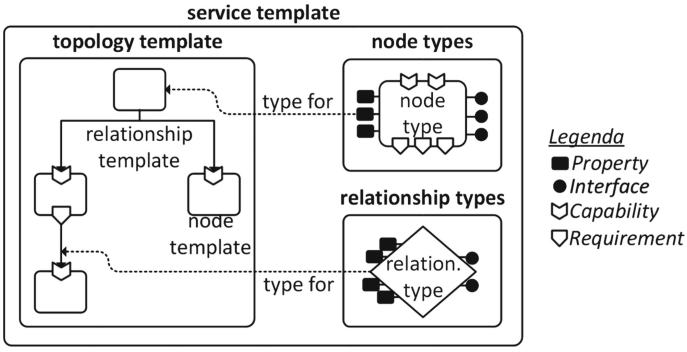

Fig. 1. The TOSCA metamodel [27].

TOSCA permits specifying a cloud application as a service template, that is in turn composed by a topology template, and by the types needed to build such a topology template (Fig. 1). The topology template is a typed directed graph that

[1] This can also be done automatically by exploiting TosKeriser [10]. Given a TOSCA application specification, TosKeriser can indeed automatically (discover and) include the Docker containers offering the software support needed by its components.

[2] A more detailed, self-contained introduction to TOSCA can be found in [5,12].

describes the topological structure of a multi-component application. Its nodes (called node templates) model the application components, while its edges (called relationship templates) model the relations occurring among such components.

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. As the TOSCA type system supports inheritance, a node/relationship type can be defined by extending another, thus permitting the former to inherit the latter's properties, requirements, capabilities, interfaces, and operations (if any).

Node templates and relationship templates also specify the artifacts needed to actually realise their deployment or to implement their management operations. As TOSCA allows artifacts to represent contents of any type (e.g., scripts, executables, images, configuration files, etc.), the metadata needed to properly access and process them is described by means of artifact types.

TOSCA applications are then packaged and distributed in CSARs (*Cloud Service ARchives*). A CSAR is essentially a zip archive containing an application specification along with the concrete artifacts realising the deployment and management operations of its components.

3 Specifying Multi-component Applications

Multi-component applications typically integrate various and heterogeneous components [18]. We hereby define a TOSCA-based representation for such components, as well as for the Docker containers and Docker volumes that will be used to form their runtime infrastructure.

We first define three different TOSCA node types[3] to permit distinguishing the Docker containers, the Docker volumes, and the application components forming a multi-component application (Fig. 2).

– *tosker.nodes.Container* permits representing Docker containers, by indicating whether a container requires a *connection* (to another Docker container or to an application component), whether it has a generic *dependency* on another node in the topology, or whether it needs some persistent *storage* (hence requiring to be attached to a Docker volume). *tosker.nodes.Container* also permits indicating whether a container can *host* an application component, whether it offers an *endpoint* where to connect to, or whether it offers a generic *feature* (to satisfy a generic *dependency* requirement of another

[3] The actual definition of all TOSCA types discussed in this section is publicly available on GitHub at https://github.com/di-unipi-socc/tosker-types.

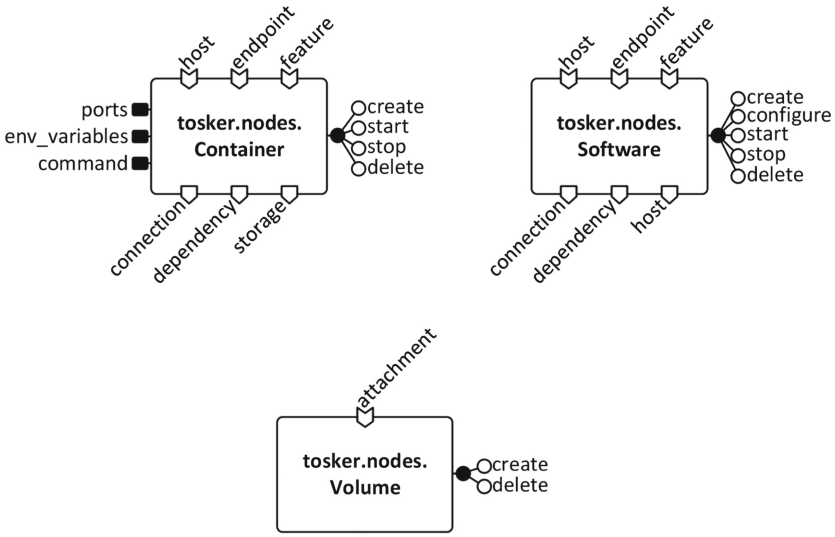

Fig. 2. TOSCA node types for multi-component, Docker-based applications, viz., *tosker.nodes.Container*, *tosker.nodes.Software*, and *tosker.nodes.Volume*.

container/application component). To complete the description, *tosker.nodes.Container* provides placeholders (through the properties *ports*, *env_variables* and *command*, respectively) for specifying the port mappings, the environment variables, and the command to be executed when running the corresponding Docker container, and it lists the operations to manage a container (which correspond to the basic operations offered by the Docker platform [23]).

- *tosker.nodes.Volume* permits specifying Docker volumes, and it defines a capability *attachment* to indicate that a Docker volume can satisfy the *storage* requirements of Docker containers. It also lists the operations to manage a Docker volume (which corresponds to the basic operations offered by the Docker platform [23]).
- *tosker.nodes.Software* permits indicating the software components forming a multi-component application. It permits specifying whether an application component requires a *connection* (to a Docker container or to another application component), whether it has a generic *dependency* on another node in the topology, and that it has to be *host*ed on a Docker container or on another component[4]. *tosker.nodes.Software* also permits indicating whether an application component can *host* another application component, whether it provides an *endpoint* where to connect to, or whether it offers a generic *feature* (to satisfy a generic *dependency* requirement of a container/application

[4] The *host* requirement is mandatory for nodes of type *tosker.nodes.Software*, as we assume that each application component must be installed in another component or in a Docker container.

component). Finally, *tosker.nodes.Software* indicates the operations to manage an application component (viz., *create*, *configure*, *start*, *stop*, *delete*).

The interconnections and interdependencies among the nodes forming a multi-component application can be indicated by exploiting the TOSCA normative relationship types [27].

- *tosca.relationships.AttachesTo* can indeed be used to attach a Docker volume to a Docker container.
- *tosca.relationships.ConnectsTo* can indicate the network connections to establish between Docker containers and/or application components.
- *tosca.relationships.HostedOn* can be used to indicate that an application component is hosted on another component or on a Docker container (e.g., to indicate that a web service is hosted on a web server, which is in turn hosted on a Docker container).
- *tosca.relationships.DependsOn* can be used to indicate generic dependencies between the nodes of a multi-component application (e.g., to indicate that a component must be deployed before another, as the latter depends on the availability of the former to properly work).

Example 1. Consider *Thinking*, an open-source[5] web application that allows users to share their thoughts, so that all other users can read them. *Thinking* is composed by three main components, namely (i) a Mongo database storing the collection of thoughts shared by end-users, (ii) a Java-based REST API to remotely access the database of shared thoughts, and (iii) a web-based GUI visualising all shared thoughts and allowing to insert new thoughts into the database. Figure 3 illustrates a representation of the *Thinking* application in TOSCA.

(i) The database is obtained by directly instantiating a *MongoDB* container, which needs to be attached to a volume where the shared thoughts will be persistently stored.
(ii) The *API* is hosted on a *Maven* Docker container, and it requires to be connected to the *MongoDB* container (for remotely accessing the database of shared thoughts).
(iii) The *GUI* is hosted on a *NodeJS* Docker container, and it depends on the availability of the *API* to properly work (as it sends GET/POST requests to the *API* to retrieve/add shared thoughts). □

Finally, also artifacts must be typed [27], as they are used to implement deployment and management operations of the nodes forming a multi-component application and they must specify the metadata needed to properly access and process them. We hence define *tosker.artifacts.Image* and *tosker.artifacts.Dockerfile* to permit indicating that an artifact is an actual image or a Dockerfile, which will then be used to create a Docker container. We also extend such artifact types by defining *tosker.artifacts.Image.Service* and *tosker.artifacts.Dockerfile.Service*,

[5] The source code of *Thinking* is publicly available on GitHub at https://github.com/di-unipi-socc/thinking.

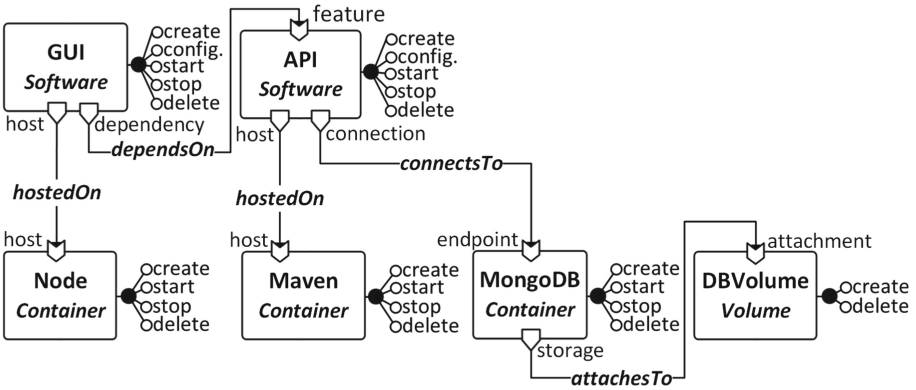

Fig. 3. An example of multi-component application specified in TOSCA (where nodes are typed with *tosker.nodes.Container*, *tosker.nodes.Volume*, or *tosker.nodes.Software*, while relationships are typed with TOSCA normative types [27]).

to permit distinguishing images that execute a service when started from those that "simply package" a runtime environment. We can instead rely on TOSCA normative artifact types [27] for all other kinds of artifacts linked by the nodes in a multi-container Docker application.

Example 1 (cont.). Consider again the application in Fig. 3. The image artifact associated to the *MongoDB* container is of type *tosker.artifacts.Image.Service*, as it links to an image offering a MongoDB server when executed. The image artifacts associated to the containers *Node* and *Maven* are instead of type *tosker.artifacts.Image*, as they link to images just offering runtime environments (for NodeJS-based and Maven-based applications, respectively). The management operations of *GUI* and *API* are instead implemented by ".sh" scripts[6]. □

4 TosKer

We hereby present TosKer, an orchestrator capable of automatically deploying and managing multi-component applications specified with the proposed TOSCA representation. We first illustrate the architecture of TosKer, and we then discuss its current prototype implementation.

[6] The resulting TOSCA application specification is publicly available at https://github.com/di-unipi-socc/TosKer/blob/master/data/examples/thoughts-app/thoughts/thoughts.yaml. A CSAR packaging such specification (together with all artifacts needed to deploy and manage the *Thinking* application) is available at https://github.com/di-unipi-socc/TosKer/blob/master/data/examples/thoughts-app/thoughts.csar.

4.1 The Architecture of TosKer

Figure 4 shows the architecture of TosKer, which is designed to be modular and easily extensible. The architecture of TosKer indeed partitions the functionalities of TosKer into lightweight modules that interact with each other, and new functionalities can be easily added to TosKer by developing and plugging-in new modules.

User interface. The UI allows to feed TosKer with the necessary input. The latter includes a CSAR (packaging the TOSCA specification of a multi-component application together with all artifacts needed to realise its management), a sequence of management operations to be executed, and (optionally) the subset of the application components on which to perform such a sequence of management operations.

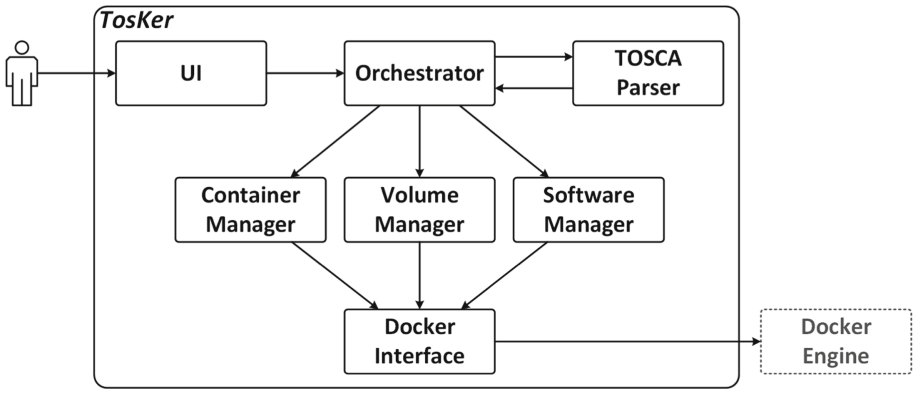

Fig. 4. The architecture of TosKer.

TOSCA utilities. The TOSCA Parser is an utility module for parsing a CSAR and generating an internal representation of the application it packages. Such representation will then be exploited by the other modules in TosKer to deploy and manage the corresponding application.

Orchestration core. The Orchestrator is the core component of TosKer, as it is in charge of planning and orchestrating the management of multi-component applications. It first receives the input from the UI, and it exploits the TOSCA Parser to generate an internal representation of the multi-component application contained in the input CSAR.

The Orchestrator automatically determines which management operations have to be executed on which components, and in which order[7]. (to permit

[7] The Orchestrator assumes that components are managed according to the TOSCA standard management lifecycle [27]. If such lifecycle is not respected (e.g., by requiring to *delete* a component that has not yet been created), then the Orchestrator will raise an error and stop orchestrating the application management.

executing the input sequence of operations on the indicated subset of application components). The result is a (possibly expanded) sequence of management operations, each to be executed on a certain application component.

The Orchestrator then orchestrates the actual execution the above mentioned sequence of management operations by coordinating the Container Manager, Volume Manager and Software Manager. It indeed iterates over the sequence, and it dispatches the actual execution of an operation on a component to the corresponding manager (e.g., to *create* a component of type *tosker.nodes.Container*, the Orchestrator dispatches the actual execution of *create* on such component to the Container Manager). dispatched to the

Managers. The Container Manager, Volume Manager, and Software Manager implement the actual lifecycle for components of type *tosker.nodes.Container*, *tosker.nodes.Volume*, and *tosker.nodes.Software*, respectively.

- The Container Manager is in charge of implementing the operations to *create*, *start*, *stop* and *delete* Docker containers, by also taking into account the different types of artifacts from which they are generated (viz., Docker images or Dockerfiles—see Sect. 3).
- The Volume Manager has to implement the operations to *create* and *delete* Docker volumes (as volumes can only be created or deleted [23]).
- The Software Manager is in charge of implementing the operations to *create*, *configure*, *start*, *stop* and *delete* a component of type *tosker.nodes.Software*. Notice that, as such a kind of components will be hosted on Docker containers, the actual execution of a management operation on a component requires to issue commands to its container. For instance, to *create* a component, the Software Manager has to (i) copy all artifacts of the component inside a dedicated folder of its container, (ii) start the container by executing the script implementing the *create* operation of the component, (iii) commit the changes applied to the container as a new image, and (iv) re-create the container by exploiting the newly created image.

Notice that each manager implements management operations by instructing the Docker Interface on which Docker commands to execute.

Docker interface. The Docker Interface is in charge of interacting with the Docker engine installed on the host where TosKer is running. It is used by the managers to manage Docker containers and Docker volumes, and to execute operations inside running containers.

Notice that the Docker Interface decouples TosKer from the actual Docker engine used, meaning that it can issue commands to a classic Docker engine (as in the current implementation of TosKer—see Sect. 4.2), but it could also be used to issue commands to an engine capable of distributing containers in a cluster (e.g., Docker swarm [16] or Kubernetes [32]).

4.2 Prototype Implementation

We have implemented a prototype of TosKer, which is open-source and publicly available on GitHub[8]. The prototype is written in Python[9], and it is composed by a main package (`tosker`) containing the set of Python modules implementing the various components forming the architecture of TosKer (viz., `ui.py`, `tosca_parser.py`, `orchestrator.py`, `container_manager.py`, `volume_manager.py`, `software_manager.py`, and `docker_interface.py`).

The current prototype of TosKer is also published on PyPI[10] (*Python Package index*), which permits installing it on a host by simply executing the command `pip install tosker`. It can then be used as a standard Python library, or as a command line software by executing:

```
$ tosker FILE [COMPONENTS] COMMANDS [INPUTS]
```

where `FILE` is a CSAR archive or a TOSCA YAML file (containing the specification of a multi-component application), `COMPONENTS` is optional and permits specifying the subset of application components to be managed, `COMMANDS` is the sequence of management operations to be executed, and `INPUTS` is an optional sequence of input parameters to be passed to the TOSCA application[11].

Example 2. Consider again the *Thinking* application in Example 1. Suppose, for instance, that we wish to *create* and *start* its *API* and *MongoDB*. We can instruct TosKer to do so, by executing:

```
$ tosker /usr/share/tosker/examples/thougthts.csar \
  API MongoDB create start
```

Notice that this will not only result in creating and starting *API* and *MongoDB*, but also the *Maven* container and *DBVolume* they require to properly work. *GUI* and *Node* will instead be ignored by TosKer, as they are not contained in set of components input to TosKer, nor they are needed by *API* or *MongoDB*. □

To test the current prototype of TosKer, we specified the open-source application *Thinking* in TOSCA, as well as three other existing applications, viz., (i) a Wordpress instance running on a PHP web server and connecting to a MySQL back-end, (ii) a NodeJS-based REST API connecting to a MongoDB back-end, and (iii) an application with three interacting servers written in NodeJS. All applications were effectively deployed by the current prototype of TosKer, and they constituted the basis for developing a battery of unit tests[12], which covered 96% of the source code of the Python modules we implemented (see Table 1).

[8] https://github.com/di-unipi-socc/TosKer.

[9] The choice of Python was mainly motivated by the availability of two open-source Python libraries: *docker-py* (https://github.com/docker/docker-py) and *tosca-parser* (https://github.com/openstack/tosca-parser/). *docker-py* implements a

Table 1. Unit test coverage in the current prototype of TosKer (obtained by running the *coverage-py* tool—https://coverage.readthedocs.io).

Module	Total statements	Missed statements	Coverage
`ui.py`	75	18	76%
`docker_interface.py`	168	4	98%
`tosca_parser.py`	219	2	99%
`orchestrator.py`	105	2	98%
`container_manager.py`	26	0	100%
`volume_manager.py`	9	0	100%
`software_manager.py`	67	2	97%
Total	669	28	96%

5 Related Work

We hereby position TosKer with respect to other currently available solutions for orchestrating the management of multi-component applications with Docker and/or TOSCA.

Docker-based orchestration. Docker natively supports multi-container Docker applications with Docker compose [15]. Docker compose permits specifying the (images of) containers forming an application, the links/connections to be set between such containers, and the volumes to be mounted. Based on that, Docker compose is capable of deploying the specified application. However, Docker compose treats containers as black-boxes, meaning that there is no information on which components are hosted by a container, and that it is not possible to orchestrate the management of application components separately from that of their containers (as it is instead possible with TosKer).

Other approaches worth mentioning are Docker swarm [16], Kubernetes [32], and Mesos [2]. Docker swarm permits creating a cluster of replicas of a Docker container, and seamlessly managing it on a cluster of hosts. Kubernetes and Mesos instead permit automating the deployment, scaling, and management of containerised applications over clusters of hosts. Docker swarm, Kubernetes and Mesos differ from TosKer as they focus on how to schedule and manage containers on clusters of hosts, rather than on how to orchestrate the management of the components and containers forming multi-component applications.

Python interface for the Docker engine API. *tosca-parser* is instead a parser for TOSCA application specifications (developed by the OpenStack community).

[10] https://pypi.python.org/pypi/tosKer.

[11] Details on how to process inputs for TOSCA applications can be found in [27].

[12] The TOSCA application specifications and the battery of unit tests that we implemented are publicly available on GitHub at https://github.com/di-unipi-socc/TosKer/tree/master/data/examples and https://github.com/di-unipi-socc/TosKer/tree/master/tests, respectively.

TOSCA-based orchestration. OpenTOSCA [4] is an open-source engine for deploying and managing TOSCA applications. It is designed to work with a former, XML-based version of TOSCA [25], and to process applications "imperatively" (viz., by executing management plans defined by the application developer in the form of BPEL or BPMN workflows). TosKer instead works with the newer, YAML-based version of TOSCA [27], and it is designed to process applications "declaratively" (viz., by automatically determining the management plans to be executed from the topology of an application).

Other approaches worth mentioning are SeaClouds [8], Brooklyn [1], Alien4Cloud [17], and Cloudify [20]. SeaClouds [8] is a middleware solution for deploying and managing multi-component applications on heterogeneous IaaS/PaaS clouds. SeaClouds fully supports TOSCA, but it lacks a support for Docker containers. The latter makes SeaClouds not suitable to orchestrate the management of multi-component applications including Docker containers.

Brooklyn [1], Alien4Cloud [17] and Cloudify [20] instead natively support Docker containers, and they permit orchestrating the management of the software components and Docker containers forming cloud applications. They however all differ from TosKer because they treat Docker containers as black-boxes (hence not permitting to orchestrate the management of application components separately from that of the containers hosting them).

Brooklyn [1] and Cloudify [20] also differ from TosKer as they require to specify applications in non-standard blueprint languages (inspired to, but not fully compliant with, the OASIS standards CAMP [26] and TOSCA [26], respectively). For instance, a relationship is specified in TOSCA by connecting a requirement of one component to a capability of another, and requirements/capabilities can be used to express interconnection constraints (which then permit validating TOSCA application topologies [9]). Cloudify blueprints instead do not include any notion of requirements or capabilities, as relationships just connect a source node to a target node.

Summary. To the best of our knowledge, ours is the first solution that permits specifying and orchestrating multi-component, Docker-based applications in TOSCA, and managing software components independently of the containers hosting them.

6 Conclusions

Container-based virtualisation is emerging as a simple yet effective solution for deploying and managing multi-component applications in cloud platforms [28]. Application components can be shipped within portable and lightweight Docker containers, which can then be interconnected to allow components to interact with each other. At the same time, the current support for orchestrating the management of the application components shipped within Docker containers is limited [29]. For instance, components must be manually packaged in Docker containers, and it is not possible to manage components independently of the

containers hosting them (e.g., whenever a container is started/stopped, all components hosted on such container are also started/stopped).

In this paper we illustrated how TOSCA [27] can enhance the support for orchestrating multi-component applications with Docker. We indeed (i) proposed a TOSCA-based representation for multi-component applications, which permits distinguishing the Docker containers and software components in a multi-component application, as well as the relationships occurring among them. We also (ii) presented TosKer, an orchestration engine for automatically deploying and managing multi-component applications based on TOSCA and Docker.

Our approach enhances the current support for orchestrating the management of multi-component applications in Docker. TosKer can indeed automatically install application components within the containers hosting them (instead of requiring to manually package components in images of Docker containers), and it permits independently orchestrating the management of components and containers (instead of binding the management lifecycle of components to that of the containers hosting them).

We believe that our approach can also facilitate the widespread adoption of the TOSCA standard. TosKer indeed provides a lightweight, easy-to-use engine for deploying and managing TOSCA-based applications (exploiting Docker to host their components).

We tested the current prototype of TosKer by developing a battery of unit tests based on four existing applications. A more thorough evaluation of TosKer, based on concrete case studies and/or on datasets of multi-component applications (e.g., μset [6]), is in the scope of our immediate future work.

Additionally, the current prototype of TosKer permits orchestrating applications on single hosts and it does not yet support horizontal scaling of containers. TosKer can be adapted to include such features, for instance, by simply including a new version of the Docker Interface which interacts with Docker Swarm [16] or Kubernetes [32] (instead of with the Docker engine installed on a host). This is also in the scope of our future work.

It is finally worth noting that TosKer permits orchestrating the management of multi-component applications, by already offering some basic planning capabilities. For instance, when required to *start* a component of an application, TosKer automatically determines which other components have to be started, and it plans the sequence of operations that permits starting all such components. Such planning is however based on a fixed set of operations, whose behaviour is fixed by the TOSCA standard management lifecycle [27]. This is because our approach does not yet include a way to customise the management behaviour of application components. A solution can be to integrate our approach with models designed precisely to permit compositionally describing the management behaviour of the components forming an application (e.g., Aeolus [13] or fault-aware management protocols [7]), which would also permit improving the planning capabilities of TosKer (e.g, by exploiting the Aeolus-based planning algorithm in [21]). The integration of our approach with an existing solution for modelling, analysing and planning the management of multi-component applications is also in the scope of our future work.

Acknowledgments. The authors would like to thank Claus Pahl for all helpful and stimulating discussions on how to enhance the current support for orchestrating multi-component applications with Docker, which were reported in [11] and laid the foundations for the work presented in this paper.

References

1. Apache Software Foundation: Brooklyn. http://brooklyn.apache.org
2. Apache Software Foundation: Mesos. http://mesos.apache.org/
3. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Commun. ACM **53**(4), 50–58 (2010)
4. Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., Wagner, S.: OpenTOSCA – a runtime for TOSCA-based cloud applications. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 692–695. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45005-1_62
5. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: TOSCA: portable automated deployment and management of cloud applications. In: Bouguettaya, A., Sheng, Q., Daniel, F. (eds.) Advanced Web Services, pp. 527–549. Springer, New York (2014). https://doi.org/10.1007/978-1-4614-7535-4_22
6. Brogi, A., Canciani, A., Neri, D., Rinaldi, L., Soldani, J.: Towards a reference dataset of microservice-based applications. In: Cerone, A., Roveri, M. (eds.) SEFM 2017. LNCS, vol. 10729, pp. 219–229. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74781-1_16
7. Brogi, A., Canciani, A., Soldani, J.: Fault-aware application management protocols. In: Aiello, M., Johnsen, E.B., Dustdar, S., Georgievski, I. (eds.) ESOCC 2016. LNCS, vol. 9846, pp. 219–234. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44482-6_14
8. Brogi, A., Carrasco, J., Cubo, J., D'Andria, F., Ibrahim, A., Pimentel, E., Soldani, J.: EU Project SeaClouds - adaptive management of service-based applications across multiple clouds. In: Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER 2014), pp. 758–763 (2014)
9. Brogi, A., Di Tommaso, A., Soldani, J.: Validating TOSCA application topologies. In: Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD, vol. 1, pp. 667–678. SciTePress (2017)
10. Brogi, A., Neri, D., Rinaldi, L., Soldani, J.: From (incomplete) TOSCA specifications to running applications, with Docker. In: Cerone, A., Roveri, M. (eds.) SEFM 2017. LNCS, vol. 10729, pp. 491–506. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74781-1_33
11. Brogi, A., Pahl, C., Soldani, J.: Enhancing the orchestration of multi-container Docker applications (2016). Submitted for Publication
12. Brogi, A., Soldani, J., Wang, P.W.: TOSCA in a Nutshell: promises and perspectives. In: Villari, M., Zimmermann, W., Lau, K.-K. (eds.) ESOCC 2014. LNCS, vol. 8745, pp. 171–186. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44879-3_13
13. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: a component model for the cloud. Inf. Comput. **239**, 100–121 (2014)

14. Docker Inc.: Docker. https://www.docker.com/
15. Docker Inc.: Docker compose. https://github.com/docker/compose
16. Docker Inc.: Docker swarm. https://github.com/docker/swarm
17. FastConnect, Bull, Atos: Alien4cloud. https://alien4cloud.github.io/
18. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Springer, Vienna (2014). https://doi.org/10.1007/978-3-7091-1568-8
19. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and Linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 171–172. IEEE Computer Society (2015)
20. GigaSpaces Technologies: Cloudify. http://cloudify.co/
21. Lascu, T.A., Mauro, J., Zavattaro, G.: A planning tool supporting the deployment of cloud applications. In: Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, ICTAI 2013, pp. 213–220. IEEE Computer Society (2013)
22. Leymann, F.: Cloud computing. it – Information Technology, Methoden und innovative Anwendungen der Informatik und Informationstechnik **53**(4), 163–164 (2011)
23. Matthias, K., Kane, S.P.: Docker: Up and Running. O'Reilly Media, Sebastopol (2015)
24. Newman, S.: Building Microservices. O'Reilly Media, Inc., Sebastopol (2015)
25. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA), Version 1.0 (2013). http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf
26. OASIS: Cloud Application Management for Platforms (CAMP), Version 1.1 (2016). http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.pdf
27. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Simple Profile in YAML, Version 1.0 (2016). http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.pdf
28. Pahl, C.: Containerization and the paas cloud. IEEE Cloud Comput. **2**(3), 24–31 (2015)
29. Pahl, C., Brogi, A., Soldani, J., Jamshidi, P.: Cloud container technologies: a state-of-the-art review. IEEE Trans. Cloud Comput. (in press). https://doi.org/10.1109/TCC.2017.2702586. Early access: http://ieeexplore.ieee.org/document/7922500/
30. Pahl, C., Lee, B.: Containers and clusters for edge cloud architectures - a technology review. In: Proceedings of the 2015 3rd International Conference on Future Internet of Things and Cloud, FICLOUD 2015, pp. 379–386. IEEE Computer Society (2015)
31. Smith, R.: Docker Orchestration. Packt Publishing, Birmingham (2017)
32. The Kubernetes Authors: Kubernetes. http://kubernetes.io/