Institute of Architecture of Application Systems

A TOSCA-based Programming Model for Interacting Components of Automatically Deployed Cloud and IoT Applications

Michael Zimmermann, Uwe Breitenbücher, Frank Leymann

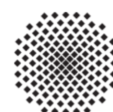
Institute of Architecture of Application Systems, University of Stuttgart, Germany,
lastname@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings {Zimmermann2017,
  author     = {Michael Zimmermann and Uwe Breitenb{\"u}cher and Frank Leymann},
  title      = {{A TOSCA-based Programming Model for Interacting Components of
                 Automatically Deployed Cloud and IoT Applications}},
  booktitle  = {Proceedings of the 19th International Conference on Enterprise
                Information Systems - Volume 2: ICEIS},
  year       = {2017},
  month      = apr,
  pages      = {121--131},
  publisher  = {SciTePress}
}
```

These publication and contributions were presented at ICEIS 2017
ICEIS 2017 Web site: http://iceis.org

© 2017 SciTePress. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the SciTePress.



Universität Stuttgart
Germany

A TOSCA-based Programming Model for Interacting Components of Automatically Deployed Cloud and IoT Applications

Michael Zimmermann, Uwe Breitenbücher, and Frank Leymann
Institute of Architecture of Application Systems, University of Stuttgart, Germany
{lastname}@iaas.uni-stuttgart.de

Keywords: Programming Model, Orchestration, Interaction, Communication, Automated Deployment, TOSCA

Abstract: Cloud applications typically consist of multiple components interacting with each other. Service-orientation, standards such as WSDL, and the workflow technology provide common means to enable the interaction between these components. Nevertheless, during the automated application deployment, endpoints of interacting components, e.g., URLs of deployed services, still need to be exchanged: the components must be wired. However, this exchange mainly depends on the used (i) middleware technologies, (ii) programming languages, and (iii) deployment technologies, which limits the application's portability and increases the complexity of implementing components. In this paper, we present a programming model for easing the implementation of interacting components of automatically deployed applications. The presented programming model is based on the TOSCA standard and enables invoking components by their identifiers and interface descriptions contained in the application's TOSCA model. The approach can be applied to Cloud and IoT applications, i.e., also software hosted on physical devices may use the approach to call other application components. To validate the practical feasibility of the approach, we present a system architecture and prototype based on OpenTOSCA.

1 INTRODUCTION

Cloud computing is an important paradigm for the realization of modern IT systems focussing on automated deployment and management (Leymann, 2009). According to an overview of forecasts published by Forbes (Columbus, 2016), the importance of cloud computing for the market is still growing. Since this paradigm nearly allows using infinite computing resources on demand, it enables developers to easily build highly elastic cloud applications. To benefit from cloud properties, applications are typically composed of multiple interacting components and services. As a result, the orchestration and wiring of application components are major issues. But also in the field of the Internet of Things (IoT) where different sensors and actuators need to be connected in a way that they can be controlled over the internet, orchestration and wiring services and devices play an important role.

However, if different kinds of technologies have to be used to build an application, also different information about each component need to be exchanged between them during the automated deployment to enable their interaction. Consider an IoT scenario in which a cloud platform offering is used for hosting a graphical frontend component showing information about a physical device, e.g., a measured temperature. The participating components, i.e., the software on the device, the device itself, the frontend software, and the platform need to be wired during the deployment of the overall application: the endpoint of the frontend needs to be known by the device software to publish information. As a result, during the deployment of such *composite applications*, typically endpoint information, e.g., the IP-address, credentials, and communication protocols, need to be exchanged between components to enable their interaction. Unfortunately, such mechanisms are typically bound to a certain technology and depend on the used (i) middleware technologies, (ii) programming languages, and (iii) deployment technologies. As a result, custom code needs to be written in components to receive endpoint information. Thus, despite the availability of technologies for describing and abstracting communication, e.g., WSDL (W3C, 2001), service buses, and orchestration capabilities of deployment technologies such as Docker Compose[1], if multiple heterogeneous technologies need to be combined, orchestration and wiring are still technology-specific and open issues—a standards-based programming model is missing.

[1] https://docs.docker.com/compose/

In this paper, we tackle these issues. We present a programming model for easing the implementation of interacting components of automatically deployed applications by abstracting endpoint handling. The presented programming model is based on the TOSCA standard (OASIS, 2013b; OASIS, 2013a) and enables invoking components by their identifiers and interface descriptions contained in the application's TOSCA model via a service bus. The approach can be applied to both Cloud and IoT applications, i.e., also software hosted on physical devices may use the approach to call other application components and to abstract configuration issues. To validate the practical feasibility of the approach, we present a system architecture and prototype including a Camel-based service bus, which understands the corresponding TOSCA model to route the invocations between components. To summarize, the main contributions we present in this paper are:

- An extension of the Topology and Orchestration Specification for Cloud Applications (TOSCA) (OASIS, 2013b) to define business operations of components in a technology-agnostic manner
- A TOSCA-based programming model that enables the unified communication between components of an automatically deployed application
- A system architecture of an automated deployment and orchestration system including a service bus that enables communication between components following the presented programming model
- A prototypical implementation of the architecture based on the standards-based deployment and management system OpenTOSCA (Binz et al., 2013)

The remainder of this paper is organized as follows: In Section 2, we discuss different state of the art approaches for automating the orchestration and wiring of components and illustrate the existing problems and limitations we tackle in this work. As our work is mostly based on TOSCA, we explain the standard in Section 3. In Section 4, we present our TOSCA-based programming model, which abstracts the communication between components and any endpoint handling. Our extension to enable the modeling of application interfaces is presented in Section 5, followed by the corresponding communication concepts implemented as service bus, discussed in Section 6. Our approach is validated by a prototypical implementation in Section 7. Finally, related work is discussed in Section 8 and our conclusion is presented in Section 9.

2 PROBLEM STATEMENT

In this section, we discuss different state-of-the-art approaches for automating the orchestration and wiring of components using existing technologies. Furthermore, on the basis of the discussed approaches, we illustrate the problems, e.g., the exchange of endpoint information, that take place when utilizing them.

Of course, using a single composition technology, such as Docker Compose[2] or Kubernetes[3], solves the issue of automatically wiring components of applications in which all components are deployed and operated using only one technology: Such technologies typically provide built-in wiring and orchestration capabilities that must be considered when implementing a component, e.g., by propagating environment variables to containers or by placing and sharing configuration files, which are used by a component to connect to another one (Burns et al., 2016). However, in composite cloud applications that consist of multiple heterogeneous components—especially if physical devices are involved in IoT scenarios—typically multiple technologies have to be combined (Breitenbücher et al., 2013). Unfortunately, this also requires combining multiple invocation mechanisms, protocols, and endpoint exchange mechanisms leading to custom code that binds a component to an invoked component and its implementation if no *service bus* (Chappell, 2004) or—for cyber-physical scenarios—*IoT middleware* (Guth et al., 2016) is used for abstraction.

Accordingly, for the interaction of (micro)services, the endpoint of a service needs to be known by the calling service to enable their communication. While the service bus concept solves this issue from a communication layer perspective, if a concrete target service shall be invoked, at least its unique identifier (*ID* in the following) is needed and must be contained in the message sent to the bus. If an IoT middleware is used, such as a message broker, typically the ID of the topic to which a devices publishes must be known by sender and receiver. However, exchanging such IDs is technically similar to exchanging endpoints of the invoked components, e.g., URLs of the deployed components. Thus, nevertheless which approach is used, an appropriate exchange mechanism is required. Especially, such information is typically required during deployment time of a component to tell it to which other components (or to which service bus) it shall connect[4]. Since there is no standardized approach for

[2]https://docs.docker.com/compose/

[3]http://kubernetes.io/

[4]This is a general requirement if an entire application gets deployed automatically. Of course, this does not apply to hard-wired scenarios, which are not the focus of our work.

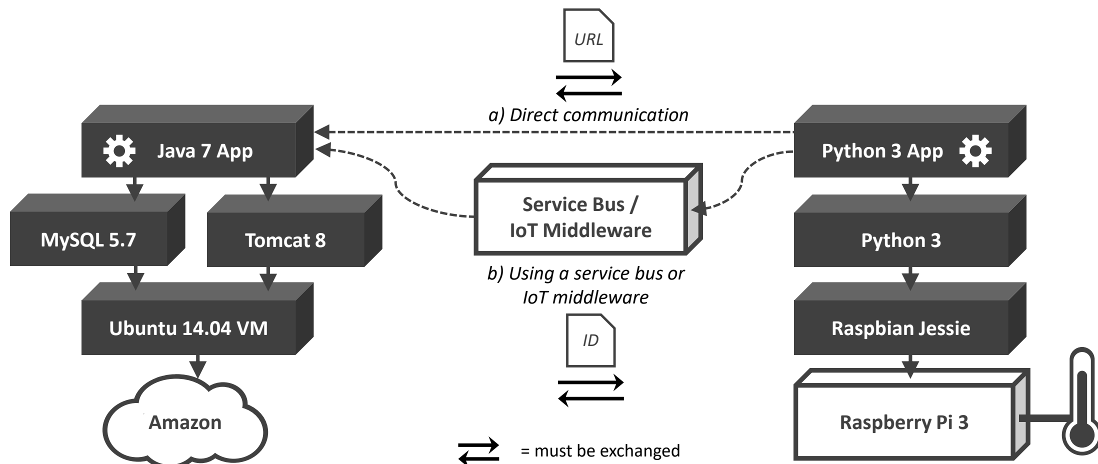


Figure 1: Two exemplary state of the art orchestration variants of an IoT-Cloud scenario.

(i) exchanging arbitrary kinds of endpoint information between components which they require to communicate with each other and (ii) exchanging IDs to enable components to invoke a certain component via a service bus, this kind of information is typically handled in an application-specific manner during the deployment time of the overall application using manually created configuration scripts and similar approaches.

For example, if a component is implemented as script, typically environment variables are used to pass information. This kind of exchange is used, for example, in an approach of Wettinger et al. enabling the unified invocation of scripts implementing management operations (Wettinger et al., 2014). Also, often configuration files need to be updated, e.g., as used by da Silva et al. in an IoT deployment scenario (da Silva et al., 2016). All these issues are reflected in the implementations of components, which limits the application's portability since the used technologies and their exchange mechanisms need to be considered.

To summarize, despite service-orientation, standards such as WSDL, service buses and the workflow technology, which provide common means to enable the *interaction* between components, their *automated deployment and wiring* is still a technology-dependent issue. Moreover, this issue itself is highly depending on the used (i) middleware technologies, (ii) programming languages, and (iii) deployment technologies. As a result, this significantly increases the complexity of implementing components as well as orchestrating them. Thus, leading to custom written code.

In the next section, we illustrate the problems that occur when using state of the art wiring approaches, for example, establishing a direct communication between two components or applying a service bus instead by means of an exemplary IoT-Cloud scenario.

2.1 Motivating Scenario

Figure 1 depicts a typical IoT-Cloud scenario describing the wiring of components. In this scenario, the *Python 3 App* running on a Raspberry Pi measures temperature data that shall be sent to the *Java 7 App*, which is responsible for storing and displaying this data. In order to enable the *Python 3 App* sending the measured temperature data to the *Java 7 App* after the automated provisioning of all shown components, the *Python 3 App* needs additional endpoint information.

Two possibilities to connect the components are shown: (i) a direct communication and (ii) a communication via a central service bus. However, both variants require exchanging endpoint information: The *Python 3 App* either needs (i) an endpoint (e.g. an URL) of the *Java 7 App* in case of a direct communication, or in case of using the service bus, (ii) some kind of ID specifying the *Java 7 App* needs to be known. Moreover, in case of the service bus, the *Java 7 App* must be registered at the bus to make itself known. Even when using a standard such as WS-Addressing[5], some information needs to be exchanged before an initial connection can be established. This results in custom code written for each component to accomplish the initial exchange of the required endpoint information. However, this decreases the portability of components since it binds them to the used orchestration technology, in particular, to its endpoint exchange mechanism. Moreover, additional effort and expertise is required for implementing components and debugging due to multiple error sources. To address these issues, we present a standards-based programming model to abstract the communication between heterogeneous components and proprietary endpoint exchange mechanisms.

[5]https://www.w3.org/TR/ws-addr-core/

3 THE TOSCA STANDARD

To provide a comprehensive background, we first explain the TOSCA standard in this section because all the following concepts are based on this language.

The Topology and Orchestration Specification for Cloud Applications (TOSCA) (OASIS, 2013b; OASIS, 2013a; Binz et al., 2014) is an official OASIS standard enabling to describe the needed infrastructure resources, the components, as well as the structure of a cloud application in an interoperable and portable manner. Furthermore, TOSCA supports the definition of operations required for the management of an application. Thus, TOSCA enables the automated provisioning and management of cloud applications. The structure of a cloud application is defined in a *topology template*. Figure 2 shows such a template that models the cloud part of the motivating scenario. A topology template is a graph consisting of nodes and directed edges. The nodes are called *node templates* and represent components of the application, e.g., an Apache Tomcat, a MySQL-Database, a virtual machine, or a cloud provider. The edges connecting the nodes are called *relationship templates*, allowing to model the relationships between the nodes. For example, a relation could be *"hosted on"* specifying that a component is hosted on another component, *"depends on"* specifying that a component has dependencies to another component, or *"connects to"* specifying that a component needs to connect to a database, for example.

For reusability purposes, TOSCA allows the specification of *node types* and *relationship types* defining the semantics of the node and relationship templates. For example, *properties*, e.g., credentials or the port of a web server, as well as available *management operations* of a modeled component are defined within the types. In Figure 2, types are put into brackets following the visual notation VINO4TOSCA (Breitenbücher et al., 2012). Management operations are bundled in *interfaces* and enable the management of the respective component. For example, a component node usually provides an "install" operation for installing the component, while a hypervisor or cloud provider node typically provides a "createVM" operation for creating a new virtual machine. The artifacts implementing the management operations are called *implementation artifacts*, which are, for example, implemented as web service packaged as WAR file or just a simple SH script. Additionally, besides implementation artifacts, TOSCA defines *deployment artifacts*, which represent the artifacts implementing the business logic of the nodes. A deployment artifact, for example, could also be a WAR file, but implementing the Java application that should be provisioned on the VM.

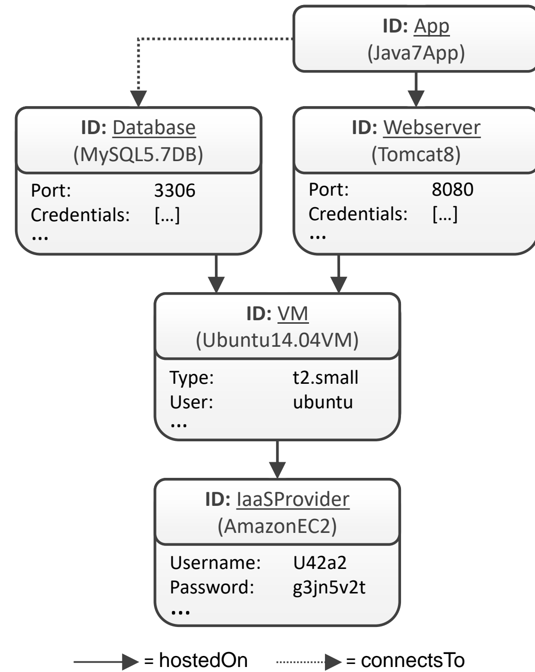


Figure 2: Exemplary TOSCA Topology Template.

In order to create or terminate an instance of a topology template and to allow the automated management of the application, so-called *management plans* can be specified in TOSCA models. Management plans are executable workflow models that implement a certain management functionality. They define which management operations need to be executed in which order to achieve a higher level management goal, e.g., to provision a new instance of the entire application or to scale out a component. TOSCA does not specify a particular process modeling language for the definition of plans, however, recommends to use a workflow language such as the *Business Process Execution Language (BPEL)* (OASIS, 2007) or the *Business Process Model and Notation (BPMN)* (OMG, 2011).

Moreover, the standard specifies a portable packaging format: All artifacts, type definitions, the topology template, management plans, and all additional files required for automating the provisioning and management can be packaged into a self-contained *Cloud Service ARchive (CSAR)*. This archive can be executed by all standard-compliant *TOSCA Runtime Environments*, e.g., OpenTOSCA (Binz et al., 2013), and, thus, ensures the application's portability and interoperability.

In the next sections, we present our novel programming model as well as our extension of TOSCA enabling the definition of operations implementing the business logic of cloud applications in the almost exact same manner as the management operations are defined when using the standard TOSCA elements.

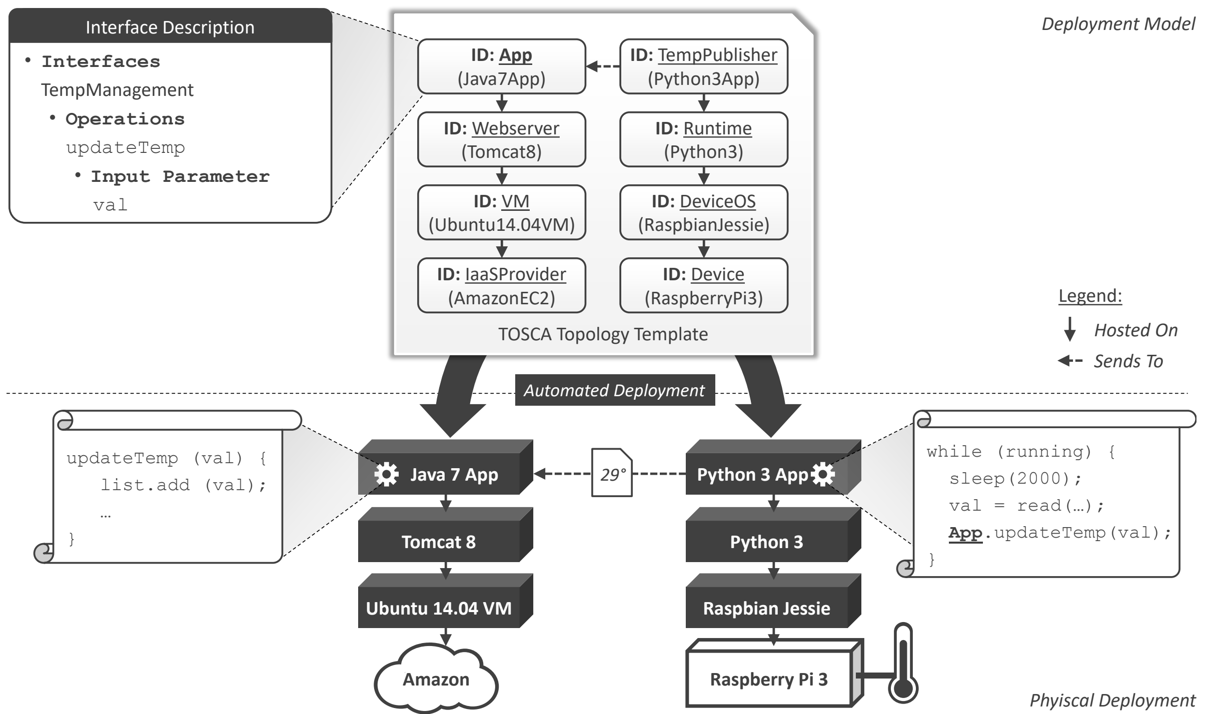


Figure 3: TOSCA-based Programming Model based on the simplified motivating scenario: All components of the application can be invoked within the implementation of another component using an unique identifier specified in the TOSCA model.

4 TOSCA-BASED PROGRAMMING MODEL

This section presents our TOSCA-based programming model, whose goal is to completely abstract (i) the communication between components and (ii) any endpoint handling. It allows to program the invocation of operations offered by other components in almost the same manner as they would be available locally.

Figure 3 shows the concept of the programming model. On the upper half, a simplified deployment model of the motivating scenario is modeled as TOSCA topology template. On the left side of the template, the component *Java 7 App* with ID *App* and its underlying stack hosted on the Amazon cloud is shown. Furthermore, a description of a business interface *TempManagement* and its operation *updateTemp* to update a temperature value with the input parameter *val* is illustrated. On the right side of the template, the stack of the component *Python 3 App* with ID *TempPublisher*, which shall be hosted on a physical *Raspberry Pi 3*, is depicted. The main function of the *TempPublisher* is to send measured temperature data to the *Java 7 App* by invoking its operation *updateTemp*. On the lower half of the figure, a physical deployment of this template is shown. Also, the temperature sensor connected to the *Raspberry Pi 3* is depicted in this physical deployment view. On the left side, an exemplary implementation of the *updateTemp* operation is shown in pseudo code while the right side shows the simplified implementation of the *TempPublisher*.

The main idea of our programming-model is to enable the invocation of operations provided by other components only based on information contained in the TOSCA topology template: to implement an invocation, the TOSCA ID of the component to be invoked is used as object in the code while the desired operation is called as usual in object-oriented programming. For example, the code of the *TempPublisher* contains an invocation of the *updateTemp*-operation of the TOSCA node template having the ID *App* (`App.updateTemp(val)`). Thus, although the *App* component is hosted on the Amazon cloud and the *TempPublisher* component is hosted on a physical device, the operation *updateTemp* can be used within the *TempPublisher* component as it would be a locally available method. Therefore, no programming for endpoint handling or against a service bus is required, which abstracts all relevant wiring aspects. Furthermore, since all IDs of the components are specified in the topology template and, therefore, are well-known, no exchange of IDs is required at all in order to discover components and to establish a connection enabling the communication between components.

5 TOSCA EXTENSION

For the realization of our programming-model, the operations implementing business logic of an application must be defined in the corresponding TOSCA model. Since this is not supported by TOSCA out of the box, in this section, we present a TOSCA-extension to define *business operations* of applications modeled using TOSCA. Thus, we extend TOSCA by an *Interface Definition Language (IDL)* for business operations. In order to ease understanding business operations, we first dissociate them from management operations, which can be already modeled in TOSCA. We also discuss how our new abstraction layer nicely complements accepted standards such as WSDL (W3C, 2001).

5.1 Application Interfaces

In Section 3 we presented the fundamentals of the TOSCA standard. We outlined that implementation artifacts implement the management operations provided by node templates. Implementation artifacts can be realized using any arbitrary technology such as a simple shell script, a WAR file exposing a web service, or more sophisticated technologies such as Chef recipes (Taylor and Vargo, 2014) or Ansible playbooks (Mohaan and Raithatha, 2014). Orchestrated by management plans, these management operations enable automating arbitrary management tasks of cloud applications. But besides management operations, nodes of course can also have operations implementing the business logic of the corresponding node. For example, in our motivating scenario, the Java application provides an operation to update the temperature data to be displayed (cf. Section 2.1). However, currently this business operation cannot be modeled in TOSCA, which is required to realize our new programming model. Therefore, we extend TOSCA node types by a modeling schema for business operations.

Our TOSCA extension enables the communication between components contained (i) in one topology template or (ii) in different templates and, thus, also allows other applications to utilize the offered operations. We extended the metamodel of TOSCA node types by an *ApplicationInterfaces* element, which follows the schema of the TOSCA *ManagementInterfaces* (OASIS, 2013b). Thus, within the ApplicationInterfaces element, the elements defined for ManagementInterfaces are reused: *Operation*, *InputParameter*, and *OutputParameter* are reused. However, contained in ApplicationInterfaces, an *Operation* specifies a business operation and not a management operation. The following Listing 1 shows how application interfaces and business operations can be defined using the extension:

```
1 <NodeType name="Java7App">
2  <ot:ApplicationInterfaces
3    xmlns:ot="http://opentosca.org">
4   <Interface name="TempManagement">
5    <Operation name="updateTemp">
6     <documentation>
7      Updates the temperature
8     </documentation>
9     <InputParameters>
10     <InputParameter name="val"
11       type="xs:int"/>
12    </InputParameters>
13   </Operation>
14  </Interface>
15 </ot:ApplicationInterfaces>
16 </NodeType>
```

Listing 1: Example of the TOSCA extension for specifying application interfaces containing business operations.

In the shown example, the component *Java7App* offers the operation *updateTemp* in order to store and display the received temperature data. The temperature value is specified via the input parameter *val*[6]. Based on the shown XML listing defining the provided operation, the input and output parameters, as well as documentations, a code-skeleton (Listing 2) for the Java application can be generated (cf. Sect. 6.2):

```
1 class TempManagement {
2
3    /**
4    * Updates the temperature
5    */
6    static void updateTemp(int val) {
7     // TODO generated method stub
8    }
9 }
```

Listing 2: Generated code skeleton in Java.

5.2 Bindings

However, in order to technically enable callers, such as a service bus, invoking the specified business operation, certain binding information regarding the invocation style and typically application-specific properties are required. These information need to be specified by the application developer in the TOSCA model of the corresponding operation, so that they are available during runtime. The following XML listing (Listing 3) shows an example of such binding information.

[6] Output parameters can be specified the same way

```
1 <ot:ApplicationInterfacesBinding>
2  <ot:Endpoint>/TempApp</ot:Endpoint>
3  <ot:InvocationType>JSON/REST
4  </ot:InvocationType>
5  <ot:ApplicationInterfaceInformations>
6   <ot:ApplicationInterfaceInformation
7    name="TempManagement"
8    class="org.temp.TempManagement"/>
9  </ot:ApplicationInterfaceInformations>
10 </ot:ApplicationInterfacesBinding>
```

Listing 3: Binding information.

These binding information shown in Listing 3 need to be defined in the artifact template, which is referenced by the deployment artifact implementing the business operations defined in an application interface of the corresponding node template. To recap, node templates represent components within a TOSCA topology template whereas deployment artifacts represent the artifacts implementing the business logic of such a component, e.g., a WAR file implementing the Java application that should be installed (cf. Section 3). Thus, these binding information together with our new TOSCA extension described in Section 5 allow to specify the provided business operations of a component as well as how they have to be invoked in detail.

In contrast to using such a custom artifact template for binding business operations defined in application interfaces to their implementation, accepted standards can be used, too. For example, the W3C defined the Web Services Description Language (WSDL) (W3C, 2001) in order to describe the provided functionality of web services. A WSDL file allows to bind the signature of an operation, i.e., the name and the input and output parameters, to information about how this operation can be invoked, such as the endpoint and the supported communication protocol.

However, Wettinger et al. (Wettinger et al., 2014) presented a similar TOSCA-based approach to define such binding information within an artifact template for management operations. Therefore, in order to allow a consistent definition for management operations as well as business operations, we decided to additionally support this custom definitions of binding information within an artifact template, too.

Thus, all together, our approach supports (i) a binding definition as already used within another TOSCA-based approach as well as (ii) standards such as WSDL. In case of using WSDL, the interface and operations specified within the TOSCA model should correspond to the information specified within the WSDL file. Then, our presented approach enables to use all the proven and established tooling possibilities for WSDL, such as automated top-down code generation.

6 SYSTEM ARCHITECTURE

As their was no possibility to define business operations using TOSCA without our extension, no tool support exists enabling the communication (i) between components within one TOSCA topology template and (ii) between components of different TOSCA topology templates. Therefore, in this section, we present a system architecture for TOSCA runtimes that utilizes a service bus supporting our TOSCA extension.

6.1 Overview

The system architecture we introduce is shown in Figure 4 in a simplified manner: We only depict components of TOSCA runtimes that are required for realizing our new programming model. Of course, multiple other components are also required, for example, model interpreter, etc. A comprehensive overview on different TOSCA runtime architectures can be found in the TOSCA Primer (OASIS, 2013a).

The central component of the concept is a service bus, which is integrated in the TOSCA runtime[7]. This service bus provides a generic, unified interface for *incoming* invocations of business operations provided by components. This interface can be realized, for example, as HTTP-based REST interface, which supports synchronous operation invocations within a single HTTP request or asynchronous invocations via resource polling. Also other communication protocols, for example, a SOAP interface supporting WS-Addressing (W3C, 2004) or a plugin-based implementation are possible. Depending on the implementation of the interface, also a proxy may be used in the component's implementation to ease communicating with the bus, e.g., to handle asynchronous callbacks.

For executing invocations, the service bus also contains a plugin system for *outgoing* invocations of different types, e.g., a SOAP/HTTP plugin. In order to enable the invocation of the business operations of components, the service bus needs to be able to determine invocation-relevant properties, such as the IP-address of the deployed component providing the corresponding operation. Therefore, the service bus is integrated with other components of the TOSCA runtime to access such stored information about application instances, e.g., gathered during the provisioning.

In order that the incoming message from the service bus can be processed, the code implementing the communication part of the component of the invoked

[7]Of course, other kinds of middleware may also be used similarly for realizing our programming model, e.g., a messaging middleware. This is part of our future work.

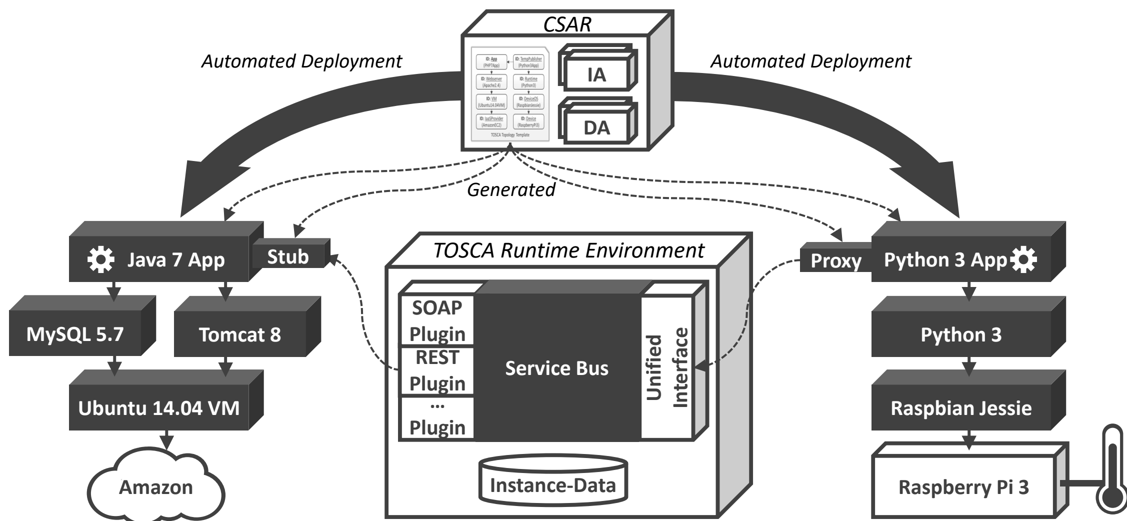


Figure 4: Simplified system Architecture of a TOSCA runtime supporting the presented programming model by a service bus.

operation of course needs to be compatible with the incoming message. There are three possibilities to reach that: (i) manually programming against a provided communication protocol of the service bus, (ii) using a generic stub suitable for an existing plugin, or (iii) using a *TOSCA Interface Compiler* to generate stubs and proxies compatible with the service bus communication protocol out of a TOSCA file. Thus, to ease receiving requests from the bus, a stub may be useful in the receiver similarly to proxies on the side of invoking components. Therefore, we introduce TOSCA Interface Compilers in the next subsection.

6.2 TOSCA Interface Compiler

To ease the implementation of the communication part of a component, our approach enables to generate a client-proxy and server-stub able to communicate with the service bus. Since all information about the business operations, their parameters, and binding are contained in the TOSCA model, similarly to generating code out of WSDL files, our approach supports this, too, in order to ease implementing interacting components. The TOSCA Interface Compiler gets TOSCA files defining the operations implementing the business logic of an application, i.e., the application interface, parses these files, generates the code, compiles it, and finally builds it. Thus, the TOSCA Interface Compiler helps (i) during the implementation phase of an application with the generation of code-skeletons of the specified operations and (ii) to generate a stub and a proxy that enable the communication with the service bus, as it is shown in Figure 4. Before the generation, the compiler can be customized, e.g., to choose the programming languages of the components for including required libraries, etc. If a separate WSDL file is referenced instead of using our binding definition (cf. Section 5.2), also the top-down approach for code generation using any WSDL tool can be used. Thus, our approach complements existing code generation tools and enables their efficient usage during development.

7 VALIDATION

In this section, we validate the practical feasibility of the presented concepts by implementing a prototype, which is integrated with an open-source toolchain.

To implement our prototype, we used and extended the *OpenTOSCA Ecosystem*[8], which consists of: (i) the graphical TOSCA modeling tool *Winery*[9] (Kopp et al., 2013), (ii) the *OpenTOSCA container*[10] (Binz et al., 2013), and (iii) the self-service portal *Vinothek* (Breitenbücher et al., 2014b). Winery is used to model the topology template of the application and to package all files into a CSAR. The resulting CSAR can be used as input for the OpenTOSCA container, which interprets the contained files and deploys the modeled application[11]. The provisioning of the application can be triggered using the self-service portal Vinothek, which provides a graphical, web-based end user interface. All tools mentioned in this section are available as open-source implementations. Thus, our developed

[8]For testing, instructions to automatically deploy the ecosystem can be found at http://install.opentosca.org

[9]https://projects.eclipse.org/projects/soa.winery

[10]https://www.github.com/OpenTOSCA

[11]Details about this deployment can be found in Breitenbücher et al. (Breitenbücher et al., 2014a)

and in OpenTOSCA integrated prototype provides an open-source end-to-end toolchain supporting the modeling, provisioning, management, orchestration and communication of TOSCA-based cloud applications.

The service bus is implemented in the programming language Java 1.7 and can be obtained from GitHub[12]. We implemented the service bus on top of the OSGI framework Equinox[13], a dynamic and modular component model for Java-based applications allowing us to implement the service bus in a plugin-based manner. For example, this allows to add and start new plugins, even during the runtime of the service bus. Furthermore, we used the routing and mediation engine Apache Camel[14] because of the provided support of various different communication protocols and messaging formats. Technically, as unified interface the service bus provides a RESTful web service enabling components to communicate with the service bus via XML and JSON. Long running tasks are supported through the implementation of a polling mechanism. Since our unified interface supports typical HTTP messages, a proxy is not necessarily required. However, by means of the plugin-based implementation of the service bus, further services supporting other communication protocols, such as a SOAP-based web service supporting asynchronous callbacks, can be added anytime. In this case, the generation of a proxy can help because the whole communication can be abstracted with it. Since the provisioning IDs of the components are contained in the TOSCA topology template, this approach works without any assumptions regarding the IDs of components. The service bus also provides different plugins for sending messages to a component providing business operations: one plugin supports components implemented using our generic RESTful web service stub and one plugin supports SOAP messages. Again, further plugins can be added and started if needed. For the receiving side of the component, we provide our generic stub implemented against the RESTful plugin of the service bus, which also uses a polling mechanism in case of long running tasks. The stub is implemented for Java as well as Python and is available as JAR file respectively as Python file and can be obtained from GitHub[15]. Thus, it can be easily used within a corresponding application. In IoT scenarios, Python is a widely-used programming language, e.g., used together with a Raspberry Pi in (da Silva et al., 2016). Moreover, we implemented the TOSCA Interface Compiler, which is available as JAR file[15], using Java 1.7. For the communication with SOAP messages, a stub can be generated supporting asynchronous callbacks through the use of WS-Addressing.

We also evaluated the performance of the communication using a notebook equipped with 16 GB RAM and 4 CPU cores @2.50 GHz running a Windows 10 64bit operating system. On the system we deployed two communicating components (*A* and *B*) using our prototype. In order to get a feeling on how the performance of the communication suffers from the additional central component (the service bus), we measured two timestamps: (i) when the message of the calling component *A* reached the service bus and (ii) when the message of the service bus reached the component *B*. Thus, the additional needed time caused by the service bus can be derived. The average measured time of 10 runs were 33 ms. Of course, depending on the general network performance, the additional required time caused by the bus can differ.

8 RELATED WORK

In this section, we complete our discussion about related work, which we already discussed partially in Section 2. Regarding the dynamic and flexible invocation of web services, there is different work available (De Antonellis et al., 2006; Leitner et al., 2009; Nagano et al., 2004). However, their approaches do not consider topology modeling aspects using standards such as TOSCA. Regarding TOSCA-related work, a concept as well as a prototype (Wettinger et al., 2014) enabling the invocation of operations through a unified interface was proposed. However, in their approach they only consider the invocation of management operations, which is already supported by the TOSCA standard, and do not consider operations implementing the business logic of an application.

In (Happ and Wolisz, 2016) limitations of the publish-subscribe pattern, for example implemented in the widely accepted IoT protocol MQTT, for the area of IoT are presented. For instance, they argue that a potential publisher of sensor data respectively the used topic can not be easily discovered. Furthermore, they mention that the standard is missing details regarding the messaging reliability. Thus, leading to custom solutions and implementations resulting in incompatible applications. Therefore, they provide a concept improving the discovery and reliability. However, standards describing the structure of an application such as TOSCA are not considered in their work.

The problem of integrating different custom components and technologies was already discussed in related work. (Breitenbücher et al., 2013) says, that because most of the web services and APIs of vendors

[12]https://www.github.com/OpenTOSCA

[13]http://www.eclipse.org/equinox/

[14]http://camel.apache.org/

[15]https://github.com/zimmerml/OTServiceBus

and cloud providers available are not standardized, existing solutions cannot integrate them. Thus, in their work they present an approach to integrate provisioning and configuration technologies. However, in their approach they do not consider the invocation of business operations through a unified interface. Instead, they focused only on management technologies.

In the field of container-based orchestration, there is available related work (Pahl, 2015; Tosatto et al., 2015; Bernstein, 2014) discussing orchestration approaches using containers and advantages using container technologies such as Docker Compose[16], Docker Swarm[17] and Kubernetes[18] in the cloud in general. These technologies allow, for instance, to transfer and reuse the containers between different cloud providers. However, they do not consider the orchestration of non-containerized components.

The general approach of generating a stub from an interface definition in order to enable the invocation of a remote method as a local invocation is similar to other approaches such as Java-RMI (Oracle, 2010) and CORBA (OMG, 2012). However, since we are using web service technologies such as HTTP and XML our approach is agnostic regarding the underlying technology. Also, since we use HTTP in our prototype we have no issues with firewalls blocking the traffic.

9 CONCLUSION

In this paper, we presented a programming model to ease the implementation of interacting components of automatically deployed cloud applications. To enable the modeling of operations implementing the business logic of TOSCA-based cloud applications, we introduced application interfaces extending the TOSCA standard. In order to enable the communication between components contained in TOSCA models and, thus, allowing the invocation of the defined application operations through a unified interface, we showed a prototypical implementation of a service bus and presented a system architecture. We showed how the implementation of interacting components can be simplified by our approach based on hiding all technical steps required for exchanging endpoints. To validate our architecture and TOSCA extension, we integrated the service bus into the existing open-source runtime environment OpenTOSCA. In future work, we plan to additionally integrate a message broker to support a wider range of IoT scenarios following our programming model. To improve the performance of our approach, we also plan to realize our approach in a decentralized manner avoiding a centralized component working as a service bus. Additionally, we plan to investigate other middleware technologies for enabling and coordinating the communication between components using TOSCA, e.g., by utilizing a tuple space.

[16]https://www.docker.com/products/docker-compose

[17]https://www.docker.com/products/docker-swarm

[18]http://kubernetes.io/

ACKNOWLEDGEMENTS

This work was partially funded by the BMWi project *SmartOrchestra* (01MD16001F).

REFERENCES

Bernstein, D. (2014). Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84.

Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., and Wagner, S. (2013). OpenTOSCA - A Runtime for TOSCA-based Cloud Applications. In *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013)*, pages 692–695. Springer.

Binz, T., Breitenbücher, U., Kopp, O., and Leymann, F. (2014). *TOSCA: Portable Automated Deployment and Management of Cloud Applications*, pages 527–549. Advanced Web Services. Springer.

Breitenbücher, U., Binz, T., Képes, K., Kopp, O., Leymann, F., and Wettinger, J. (2014a). Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA. In *International Conference on Cloud Engineering (IC2E 2014)*, pages 87–96. IEEE.

Breitenbücher, U., Binz, T., Kopp, O., and Leymann, F. (2014b). Vinothek - A Self-Service Portal for TOSCA. In *Proceedings of the 6th Central-European Workshop on Services and their Composition (ZEUS 2014)*, pages 69–72. CEUR-WS.org.

Breitenbücher, U., Binz, T., Kopp, O., Leymann, F., and Schumm, D. (2012). Vino4TOSCA: A Visual Notation for Application Topologies based on TOSCA. In *On the Move to Meaningful Internet Systems: OTM 2012 (CoopIS 2012)*, pages 416–424. Springer.

Breitenbücher, U., Binz, T., Kopp, O., Leymann, F., and Wettinger, J. (2013). Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies. In *On the Move to Meaningful Internet Systems: OTM 2013 Conferences (CoopIS 2013)*, pages 130–148. Springer.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, Omega, and Kubernetes. *Communications of the ACM*, 59(5):50–57.

Chappell, D. A. (2004). *Enterprise Service Bus*. O'Reilly.

Columbus, L. (2016). Roundup Of Cloud Computing Forecasts And Market Estimates, 2016.

http://www.forbes.com/sites/louiscolumbus/2016/03/13/roundup-of-cloud-computing-forecasts-and-market-estimates-2016.

da Silva, A. C. F., Breitenbücher, U., Képes, K., Kopp, O., and Leymann, F. (2016). OpenTOSCA for IoT: Automating the Deployment of IoT Applications based on the Mosquitto Message Broker. In *Proceedings of the 6th International Conference on the Internet of Things (IoT 2016)*, pages 181–182. ACM.

De Antonellis, V., Melchiori, M., De Santis, L., Mecella, M., Mussi, E., Pernici, B., and Plebani, P. (2006). A Layered Architecture for Flexible Web Service Invocation. *Software: Practice and Experience*, 36(2):191–223.

Guth, J., Breitenbücher, U., Falkenthal, M., Leymann, F., and Reinfurt, L. (2016). Comparison of IoT Platform Architectures: A Field Study based on a Reference Architecture. In *Proceedings of the International Conference on Cloudification of the Internet of Things (CIoT 2016)*. IEEE.

Happ, D. and Wolisz, A. (2016). Limitations of the Pub/Sub Pattern for Cloud Based IoT and Their Implications. In *2nd Int. Conf. on Cloudification of the Internet of Things (CIoT'16)*. IEEE.

Kopp, O., Binz, T., Breitenbücher, U., and Leymann, F. (2013). Winery – A Modeling Tool for TOSCA-based Cloud Applications. In *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013)*, pages 700–704. Springer.

Leitner, P., Rosenberg, F., and Dustdar, S. (2009). Daios: Efficient Dynamic Web Service Invocation. *Internet Computing, IEEE*, 13(3):72–80.

Leymann, F. (2009). Cloud Computing: The Next Revolution in IT. In *Proceedings of the 52th Photogrammetric Week*, pages 3–12. Wichmann Verlag.

Mohaan, M. and Raithatha, R. (2014). *Learning Ansible*. Packt Publishing.

Nagano, S., Hasegawa, T., Ohsuga, A., and Honiden, S. (2004). Dynamic Invocation Model of Web Services Using Subsumption Relations. In *Proceedings of the International Conference on Web Services (ICWS 2004)*, pages 150–156. IEEE.

OASIS (2007). *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*. Organization for the Advancement of Structured Information Standards (OASIS).

OASIS (2013a). *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*. Organization for the Advancement of Structured Information Standards (OASIS).

OASIS (2013b). *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*. Organization for the Advancement of Structured Information Standards (OASIS).

OMG (2011). *Business Process Model and Notation (BPMN) Version 2.0*. Object Management Group (OMG).

OMG (2012). *CORBA 3.3*. Object Management Group (OMG).

Oracle (2010). Java Remote Method Invocation - Distributed Computing for Java. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html.

Pahl, C. (2015). Containerisation and the PaaS Cloud. *IEEE Cloud Computing*, 2(3):24–31.

Taylor, M. and Vargo, S. (2014). *Learning Chef: A Guide to Configuration Management and Automation*. O'Reilly.

Tosatto, A., Ruiu, P., and Attanasio, A. (2015). Container-Based Orchestration in Cloud: State of the Art and Challenges. In *Ninth International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS 2015)*, pages 70–75. IEEE.

W3C (2001). *Web Service Definition Language (WSDL) Version 1.1*. World Wide Web Consortium.

W3C (2004). *Web Services Addressing (WS-Addressing)*. World Wide Web Consortium.

Wettinger, J., Binz, T., Breitenbücher, U., Kopp, O., Leymann, F., and Zimmermann, M. (2014). Unified Invocation of Scripts and Services for Provisioning, Deployment, and Management of Cloud Applications Based on TOSCA. In *Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER 2014)*, pages 559–568. SciTePress.