Beyond Mere Application Structure: Thoughts on the Future of Cloud Orchestration Tools

Jörg Domaschka*[a], Frank Griesinger[a], Daniel Baur[a], and Alessandro Rossini[b]

[a]University of Ulm, Germany
[b]SINTEF, Oslo, Norway

Abstract

Managing cloud applications running on IaaS is complicated and error prone. This is why DevOps tools and application description languages have been emerging. While these tools and languages enable the user to define the application and communication structure based on application components, they lack the possibility to define sophisticated communication patterns including the wiring on instance level. This paper details these shortcomings and presents approaches to overcome them. In particular, they we propose *(i)* adding boundaries to wiring specifications and *(ii)* introducing a higher-level abstraction—called *facet*—on top of the application. The combination of both concepts allows specifying wiring on basis of logical units and their relations. Hence, the concepts overcome general wiring problems that currently exist in cloud orchestration tools. In addition to that, the introduction of facets improves the re-use of components across different applications.

*corresponding author:
Albert-Einstein-Allee 43, D-89081 Ulm, Germany
email: joerg.domaschka@uni-ulm.de

Contents

About this Document

This open access document has been published in the Proceedings of the *1st International Conference on Cloud Forward: From Distributed to Complete Computing*, 2015 with `doi:10.1016/j.procs.2015.09.231`.

1 Introduction

The fact that managing cloud applications running on IaaS is complicated and error prone, has led to the emerging of DevOps tools that attempt to provide a PaaS-like abstraction where users only have to deal with application components and IaaS aspects move to the background. These tools are sometimes referred to as PaaS (tools). Yet, as this usage is not aligned with the well-known NIST definition of cloud computing[29] we refrain from this terminology and use *cloud orchestration tool (COT)* instead.

COTs aim at abstracting the differences between various IaaS APIs (and providers). For that reason, they come with a description or even modelling language that is used to specify the application and interdependencies between components of that applications. The COT will then deploy the application as a whole to a cloud or even in a *cross-cloud* manner across multiple clouds. Many COTs also offer the possibility to adapt the deployed application, *e.g.*, by adding and removing instances of a particular application component (*i.e.*, horizontal scaling).

Even though the use of COTs offers a considerable advantage over deploying and operating cloud applications manually, we argue that current COTs still do not allow operators to take full advantage of the real benefits of cloud computing: dynamic and scale. This is because the concepts offered to specify applications are not sufficiently expressive on the one hand and too complex to use on the other hand. Moreover, we argue that the main weakness for all COTs is their limited capability to address communication needs within a single application. The approaches found assume that *(i)* all instances of a scaled component are independent from each other and that *(ii)* other components interacting with a scaled component may interact with any of the scaled instances (anycast).

These approaches favour applications with only a few tenants and are less focused on global-scale deployments with many tenants. They also neglect the challenge of growing data and logically disjunct data sets in the same application. In particular, the approaches followed by all COTs assume that a single application instance always refers to a single data set. Yet, this is not the case, as in practice the same application can be run with different data sets, *e.g.*, one per tenant and several components of the application can be shared between data sets.

In this paper, we discuss shortcomings of existing solutions with respect to communication and wiring. Moreover, we identify missing functionality regarding the support for logical application layers bound to data sets instead of code fragments, *i.e.*, code and binaries. Finally, we sketch early ideas on how to overcome these shortcoming and what additional language concepts have to be introduced in order to do so.

This paper is structured as follows: Section 2 introduces the terminology applied in the remainder of this paper and presents the state of the art in wiring components at the level of application specification. Section 3 presents a sample application and usage scenario that the we use as a running example throughout this paper. Section 4 introduces communication patterns and wirings that go beyond what current COTs offer and motivates why their support is needed in future platforms. Section 5 discusses multiple approaches to solving the shortcomings at language and the level of application specification. Section 6 discusses the implication of our findings with respect to future lines of research. Finally, Section 7 concludes.

2 Background and State of the Art

In this section, we introduce a terminology on cloud applications. Moreover, we discuss basic constraints and properties of application wiring and the deployment workflow. Finally, we revisit the related work and background on how COTs enable the specification of cloud applications and the life-cycle handling they apply.

2.1 Terminology

Throughout this paper we use the following terminology: *Cloud platform* refers to a software stack realising IaaS. Accordingly, it also defines the API offered by that stack (*e.g.*, OpenStack Nova v2.1). A *cloud provider* runs a cloud platform under a dedicated endpoint/URI (*e.g.*, Amazon[1], Rackspace[15]). A *cloud* refers to a cloud platform offered by a cloud provider as seen by a tenant. That is, besides the endpoint of the provider, it is also bound to log-in credentials.

A *(cloud) application* is a possibly distributed application consisting of multiple interlinked application components. In particular, an application is solely a description and does not represent anything enacted. An *(application) component* is the smallest indivisible element of an application. It is the unit of scale and the unit of failure. Just like an application, a component does not represent anything enacted. For illustration consider a blog application that consists of three components: load balancer, application server (along with business logic), and database management system (DBMS) (*cf.* Section 3).

A component has several life-cycle handlers attached. A *life-cycle handler* is operating system-specific, self-contained executable software (*e.g.*, a bash script) to be run at dedicated points in time following the inversion-of-control principle [28]. The life-cycle handlers define for instance how to install, configure, and start/stop a component, but also how to detect its failure. They are the component and application providers' only mechanism to enact the component and to control its incarnation.

It is the *deployment* that enacts an application and creates an *application instance* consisting of one or more *component instances* for each of its components. Components may be connected with each other using directed, unidirectional *channels*. Deploying also has the task of *wiring* the component instances with each other. This results in *links* between instances and has to be such that the links between component instances reflect the channels between components. In practise, a link is most likely represented by messages sent over IP networks.

2.2 Wiring Component Instances

Without additional semantics on a channel specification, there is no deterministic mapping from channels to links. Connecting two components with a channel imposes that in the deployed application instance at least one component instance from the source component will have a link to at least one instance from the target component, which is a rather weak requirement. The concrete wiring between the source and target instances is subject to both the deployment and the scaling of the application, as well as the semantics set on channels by the COT. In order to realise non-trivial distributed applications, a COT has to enable the creation of links in an application-specific manner. Yet, to the best of our knowledge, there is no tool that provides means to specify advanced semantics (*cf.* Section 2.4).

In practice, we identify two basic approaches how COTs deal with wiring: *(i)* Wiring is supposed to be done by the users in the life-cycle handlers. Hence, the life-cycle handlers of two components are supposed to cooperate in order to establish the desired links. Many COTs realise this by providing access to an application-wide registry. Then the life-cycle handler of one component will write information there while the handler of the other component reads values from it. The necessary cooperation between handlers depends on the application structure and is hence, application-specific. In particular, it makes it hard to re-use a component in the context of another application, because the life-cycle handlers cannot be written an application-independent manner. This is even true for component reuse within a single COT.

(ii) The COT assumes anycast semantics and considers all scaled out instances of a single component to be equal. In that case a channel from component A to component B means

that all instances of A have links to all instances of B and can choose arbitrarily between them without violating any consistency and semantic constraints. While this approach makes component implementations application-independent, the imposed semantics may not necessarily fit the needs of all applications.

2.3 Deployment Workflow

As stated in Section 2.1, the life-cycle handling determines how to create a component instance from a component. Yet, a component instance may be dependent on an instance of another component and can only be started once the dependent instance is available. The action of running all life-cycle handlers of all components in the right order is the *deployment workflow* of a COT. The deployment functionality also provides the necessary mechanisms required by component instances in order to access the information needed for creating links, *e.g.*, the IP address and port number of a downstream instance.

Attribute and event passing is a straight forward approach where the COT offloads most complexity to the users. Here, the life-cycle scripts have to be written such that they lock/wait for attributes to become available. An improvement is a *manual workflow* definition. Here, the user defines a workflow taking care of the deployment order. Finally, for *automatic workflow* deduction the language used for specifying the application has to be sufficiently verbose to allow the COT to automatically derive the correct deployment workflow from the defined life-cycle actions and the channels.

2.4 Related Work

The state-of-the-art encompasses two modelling languages for the description of cloud application topologies: the Topology and Orchestration Specification for Cloud Applications (TOSCA)[31] and the Cloud Modelling Language (CloudML)[23, 21, 22, 34]. TOSCA is a specification developed by the OASIS consortium, which provides a language for specifying the components comprising the topology of cloud applications along with the processes for their orchestration. It is adopted by tools such as OpenTosca[11] and Cloudify[5]. CloudML is a domain-specific language to model the deployment of cross-cloud applications. It is supported by a models@run-time environment to enact the deployment and adaptation of these applications. The models@run-time environment provides a model-based representation of the underlying running system, which facilitates reasoning and adaptation of cross-cloud applications. CloudML is used in multiple projects, including PaaSage[12] and MODAClouds[9].

Both TOSCA and CloudML have a similar approach regarding the modelling of life-cycle handlers and the communications between components. The user models components and assigns life-cycle handlers (interfaces in TOSCA) that are responsible for installing, configuring, starting, and stopping the components on virtual machines. Moreover, the user models the communications between the components and assigns additional life-cycle handlers that are responsible for (re-)configuring the components to enable the communications (*e.g.* by setting the IP addresses and ports in the configuration of the components). Unfortunately, both TOSCA and CloudML lack support for modelling aspects related to the instantiation of channels as links between component instances. This means that the wiring of component instances may be different depending on the engine parsing the models and enacting the deployment of the cloud applications. In the case of CloudML, the models@run-time environment would still allow to manually change the wiring of components instances by manipulating the models at run-time. In the case of TOSCA, however, the specification does not cover instances, so the possibility to manually change the wiring of components instances at run-time depends on the capabilities

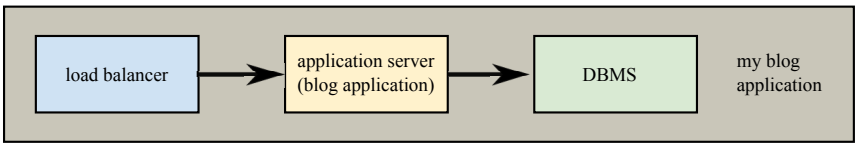


Figure 1: Our example application consisting of a load balancer, an application server with a blog application as business logic, and a database component as a storage backend.

offered by the tools. In both cases, an extension of the languages to incorporate some of the ideas presented in this paper would be desirable.

In CELAR[2], the user specifies cloud applications as compositions of components, and their relationships. A tool is provided to define the relationship as uni- or bidirectional interactions by one of the following types: master-slave, peer-peer and producer-consumer. Furthermore, the network interface can be defined for the connection[27]. Still, this definition happens on type level and not on instance level.

Moreover, there are multiple DevOps tools addressing the deployment of applications, each relying on their own domain specific language for application description. Some of them, such as Chef[3], mainly target single node deployment and are therefore missing any relationship definition. Others, like Juju[8], offer an approach for relationship handling. Juju allows to specify multiple relationships for each component (*e.g.* connection to database with master-slave). However, this forces the user to manually select the appropriate relationship type at runtime.

Finally, there are multiple COTs such Apache Stratos and Apache Brooklyn that also use their own domain specific language for application specification. Both of them outsource the wiring to the life-cycle actions and provide an attribute and event passing mechanism allowing the user to manually exchange attributes or execute actions on topology change events.

3 Running Example

Throughout this paper, we use a sample application as a running example in order to illustrate the challenges that arise when achieving a fine-grained wiring of components and their corresponding component instances. Figure 1 presents the component-oriented view of this application. We choose a blog application as representative for a classical three-tier application consisting of load balancer, application server, and DBMS. In the following, we address basic use cases for this application from which we derive more generic wiring scenarios in Section 4.

3.1 Single-tenant, Single-blog Usage

The common use case for running a blog is a single user hosting his own blog in a cloud. Using an application description derived from Figure 1, he triggers a deployment via his COT and ends up with one component instance of load balancer, application server, and database. When the load on the blog increases, the user will scale the deployment by adding more application servers. Ideally, the COT will support him by offering an auto-scale mechanism[25, 20]. Since instances of the application server are stateless, they are also independent from each other. Hence, adding one more instance of the application-server component will enable the application to handle more load.

This process of horizontally scaling the application server can continue until the single database instance is saturated. Then, the user will have to think of scaling the database. This is possible by adding more database instances, if the application has used a scalable NoSQL database such as Couchbase[6] from the very beginning. Alternatively, an add-on mechanism to support read replicas for classical relational databases such as PgPool[13] for PostgreSQL[14] is an option. In both cases, specification of the application as used by the COT has to be changed, unless it has already taken database scaling into account from the very beginning. In case of a clustering-enabled database (such as Couchbase), the application model has to cater for a reflexive channel that then maps to a cluster communication on link level.

With an ever increasing load on the application instance, the user may decide to add caches or similar mechanisms. Again, this changes the application specification as seen by the COT. Evolving the application right, the user will be able to scale along with the number of requests.

3.2 Single-tenant, Multi-blog Usage

When the user wants to host one or even multiple further independent blogs, the classical approach supported by COTs requires that for each instance of a blog a new application instance is created. As a consequence, for n blogs, he would have to run a set-up with n load balancers, n application servers, and n databases. Such a set-up is unfortunate: In one approach load balancers are run on n separate virtual machines, leading to n times the cost. In an alternative approach, multiple load balancers can be put on one virtual machine, leading to collisions on port numbers which require explicitly specifying ports in the blogs' URLs. Similar constraints apply to the DBMS component where either two virtual machines are required or two DBMS instances run on the same virtual machine. While re-using virtual machines is a feasible approach[33], the alternative of re-using application logic is even more appealing.

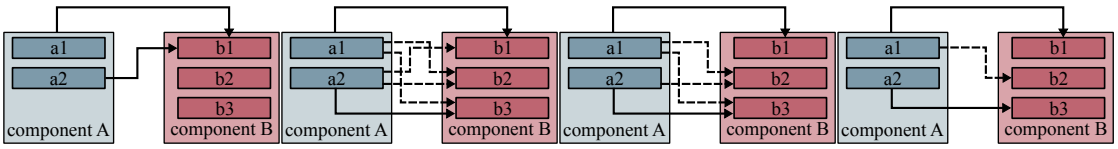
A single load balancer instance is clearly capable of balancing traffic for a larger set of clusters of application servers. Also, a DBMS is capable of hosting more than one database. Hence, a solution to the unsatisfactory approaches above is to extend the application description language such that the application infrastructure (load balancer and DBMS) is decoupled from the business logic, the message flow, and the data flow. This is addressed in Sections 4 and 5.

3.3 Multi-tenant, Multi-blog Usage

The multi-tenant, multi-blog use case extends the single-tenant, multi-blog use case by the fact that now the operator of the blog application operates it as a hosting platform for other people, hence, in a Software-as-a-Service manner. Beside all the implications already discussed for the single-tenant, multi-blog setting, here the number of further constraints is becoming tremendous. For instance, the COT operating such an application needs to be able to ensure sufficient isolation between tenants such that their work loads do not interfere beyond the acceptable. In particular, a peak load for one tenant shall not result in a degraded quality of service for other tenants. While we do not address any of these issues in this paper, this use case is the long term goal of our efforts and serves as an inspiration for future work.

4 Generic Communication Patterns

In this section, we investigate possible communication patterns of an application and its instances. These patterns may be derived from the various usage scenarios of our example application as discussed in Section 3. The fact that, on the one hand, a load balancer has to addressed different clusters of application servers, and, on the other hand, two logically separated databases may



(a) 1-1 mapping (b) 1-N mapping (c) 1-M mapping ($M = 3$) (d) Disj. 1-M mapping

Figure 2: Examples for mappings between component instances.

share a single DBMS as long as load allows, provides the initial direction. In the following, we generalise these scenarios from a more abstract point of view. Nevertheless, all discussions assume that all component instances belong to a single application instance.

Throughout this section, we use A and B as component names and denote the channel from A to B with $A{\rightarrow}B$. From our example, this covers the communication between load balancer and application server as well as the communication between application server and DBMS. By default, the specification of $A{\rightarrow}B$ has no semantics associated with it. The weakest possible semantics is that some instance a_i of A interacts with some instance b_j of B. For the discussion, we assume request-response interaction. The results, however, are not limited to such a scenario.

We distinguish between two different aspects of wiring: The static wiring as discussed in Section 4.1 defines for each a_i the set of b_j it is aware of. For each request to be sent, each a_i will then select one of these instances. Dynamic wiring that we investigate in Section 4.2 extends the static wiring to an extent where instances of A select an instance of B based on context information, *e.g.*, the kind of request to be sent.

4.1 Static Wiring of Component Instances

In the following, we describe several desirable wirings that are also sketched in Figure 2). While this list is certainly not exhaustive, we state that a mechanism that supports these structures most likely also supports other structures as well.

A *1-to-1 mapping* describes a mapping where every component instance of A is connected to only a single instance of B. In Figure 2a both instances of A use b_1. The other instances of B are not used and available to either other instances of A or to other components. A special form of the 1-to-1 mapping is the *fair 1-to-1 mapping*, where each instance of B has the same number of incoming connections. A *bijective mapping* is a special form of a 1-to-1 mapping where each target instance has exactly one source instance.

A *1-to-N mapping* describes the classical load balancer mapping. In Figure 2b both a_1 and a_2 are aware of all instances of B (dashed lines), but will only use one instance of B for their next request (solid line). Clearly, the number of instances of B that a_i will access is application-specific. In the case of load balancers, this is at least one instance while in the case of a distributed database it may be more than one depending on the distribution (sharding) of data.

A *1-to-M mapping* is a mapping where each instance of A is connected to up to M instances of component B. In Figure 2c a_1 is connected to all instances of B, while a_2 is connected to all instances but b_1 (1-to-M mapping with $M = 3$). A *disjunctive 1-to-M mapping* is a special type of 1-to-M mapping where the instances of B are grouped into disjunctive sets S_i with $1 \leq |S_i| \leq M$ and $\sum |S_i| = |B|$. Each instance of A is connected to exactly one of S_i (*cf.* Figure 2d).

4.2 Dynamic Wiring of Component Instances

Static wiring is useful when dealing with set-ups where the target set is at least temporarily fixed, but it is not sufficient for other situations. A load balancer shared between multiple blogs only knows which blog to address when receiving a request from a client visiting the blog. For instance, it might use forwarding rules that redirect requests to `http://myurl.com/blog1` to one set of application servers and requests to `http://myurl.com/blog2` to another one.

Obviously, the capability to forward HTTP requests is component-specific just like the capability to relay to different clusters depending on the incoming request. Similarly, the capability to use a fail-over instance of a DBMS is dependent on both the DBMS implementation, but also the component accessing the DBMS. We do not think that the specification of an application can be ready to support all such very special and even subtle capabilities as language concepts. Nevertheless, the language should be sufficiently expressive to allow generic access to such capabilities.

5 Towards Extended Specification Capabilities

In this section, we suggest two strategies to overcome the limitations of current approaches to specify, deploy, and operate cloud-based applications. First, we suggest to define boundaries on the endpoints of channels. Next, we suggest to introduce a new layer of abstraction that is applied on top of application instances and that represent logically separate parts of an application.

5.1 Port and Link Boundaries

The channels as they can be defined in current COTs assume an infinite cardinality on both ends of the relationship. That is, an arbitrary number of source instances can connect *to* a particular target instance while at the same time a source instance may also connect to an infinite number of target instances. By introducing upper and lower bounds on both outgoing and incoming ends of the channel (that we call *incoming* and *outgoing ports* of that relationship) this can be changed.

Here, it is desirable to differentiate between boundaries that exist due to technical limitations and those that relate to the way the application shall be constructed. Again, we have to distinguish between *component-specific aspects*, *application-specific aspects*, and *application instance-specific aspects* (*i.e.*, deployment-specific aspects).

Component-specific aspects define technical and other constraints that are inherent to the software of that component and its intended use. For instance, the fact that a component with an outgoing HTTP port (to a web server or similar) can only handle one possible URI is a technical limitation of that component. Hence, this limitation should also be specified at component level. Obviously, boundaries on this level make technical constraints on the application and components visible that previously had to be known implicitly by the DevOps of the application, as they cannot be enforced by nowadays COTs.

Application-specific aspects define the basic wiring for all instances of a particular application. This can for instance capture the insight that a target instance should not handle more than ten source instances even though it might be possible on a technical level.

Application instance-specific aspects define upper and lower bounds on ports for a particular application instance. This limits the possible deployment as well as scaling and further enables the realisation of multiple of the communication patterns from Section 4.1. In contrast to application-specific aspects, application instance-specific aspects enable usage-specific definitions and may for instance take into account the quality of service.

Examples

Again, assume a channel $A \rightarrow B$ from component A to component B. The tuple $([out_{low}, out_{up}], [in_{low}, in_{up}]$ defines the upper and lower boundaries of the incoming and outgoing ends of $A \rightarrow B$. The asterisk $*$ denotes an infinite cardinality.

Setting the boundaries of $A \rightarrow B$ to $([*, *], [0, *])$ means that a component instance of A should be connected to the maximum allowed set of component instances of B, while any instance of B can be addressed by 0 up to an infinite number of instances of A. Hence, any instance of A will connect to all instances of B (1-to-N mapping). Similarly, using $([*, *], [0, M])$ allows that all instances of A connect to by as many instances of B as possible. In contrast, any instance of B only allows up to M connections, so that all instances of A connect to M instances of B. Using $([1, *], [0, M])$ weakens this and allows that instances of A use up to M instances of B (1-to-M mapping). Setting the bounds to $([1, 1], [1, 1])$ achieves a bijective 1-to-1 mapping, whereas $([1, 1], [0, 1])$ allows that instances of B are not addressed by any of the instances of A.

5.2 Facets as Logical Layers

As we have seen, setting upper and lower boundaries on ports can achieve almost all patterns of grouping introduced in Section 4.1. What is missing, though, is a statement of how to group instances deterministically and further mechanisms that define how clusters of instances shall be formed. While we will not entirely solve these issues in this paper (*cf.* Section 6), this section introduces *facets* as a new layer of abstraction that may in the long run be beneficial to achieve these goals.

The main observation we make from our example is that in the multi-blog use cases, entities exist (individual blogs) that are not reflected in any application description. Summarising from Section 2 the application specification focuses on the mere structure of the application and with it, the high-level interaction between components that are a representation of software artefacts. In consequence, the specification has to leave out anything that is not related to software artefacts, but rather represents data or logically separated entities within the application. Also the fact that all instances of a single component are considered equal, introduces limitations.

In order to overcome these limitations, we suggest the concept of a *facet* as a new level of abstraction that represents the logical entities within an application. Conceptually, a facet resides on top of an application and consequently one or multiple *facet instances* can be created on top of application instances. Just like components and applications, facets need a mechanism to introduce properties to the runtime system and at the same time, the components and applications need a way to specify what knobs they offer to create facets. For that purpose, we introduce *facet ports* that represent hooks on the application and component level.

Facet ports have to be supported by the component definition and hence also the component life-cycle handlers. On the other hand, it is the application semantics that defines whether facets are needed or not. In the single-blog, single tenant use case, there is no need to use facets, because only one blog will be run. In any of the multi-blog scenarios, however, each blog together with its data and URI is represented as a facet. Hence, in order to not sacrifice re-usability of component definitions, it is necessary to allow their specification in a way that can deal with both alternatives. It is only when an application is built from components that the mode is selected. Supporting both alternatives may be achieved either by using a facet-based and facet-less component definition or components may support the notion of a default facet.

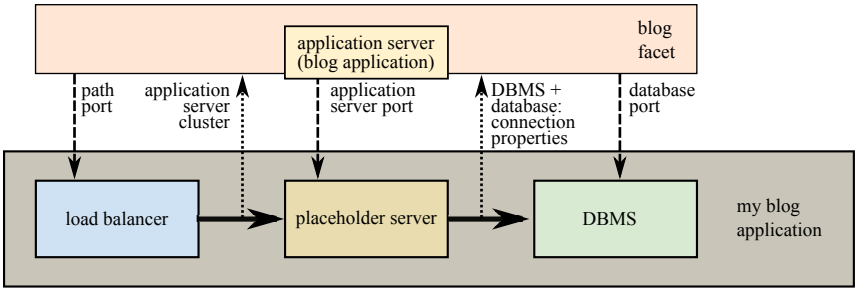


Figure 3: Example application enriched with a facet to allow creating multiple ports over the same application infrastructure.

Examples

With respect to our example, a new blog instance is represented by a facet instance. This blog facet maps to the blog application such that the facet bundles the forwarding rule of the load balancer, a cluster of application servers, and a set of databases. The application, in turn, captures the load balancer hosting the forwarding rules as well as the DBMS hosting the database. Figure 3 captures static set-up with the application, the blog facet and the facet ports being used.

Hence, the load balancer component offers an incoming facet port *path* that the outgoing facet port of the blog facet binds for instance its identifier to. When a new facet instance is created, the identifier (*e.g.*, `blog1`) is used in the load balancer instance to create a forwarding URI for that blog (*e.g.*, `http://myurl.com/blog1`). In this path, `http://myurl.com/` is a property of the load balancer instance, while `blog1` is a property of the facet instance. For the databases/DBMS part of the application, the DBMS is a part of the application definition. As a consequence, also the application defines how many databases a blog requires as well as their data schema. The facet in turn provides facet ports for specifying the actual database names to be used with a particular facet instance.

Things are more complex for the application server part. In our particular case with Ghost[7] running on Node.js[10], the full deployment of the application server including the business logic is dependant on a facet instance and cannot exist outside of it. This is due to the fact that the application server will generally require a connection to a database (and not just a DBMS) to even be able to boot. Hence, a deployment of an application without any facets defined, results in the deployment of a DBMS and a load balancer, but will not contain an instance of an application server. Yet, this limitation is a particular consequence of the set-up we run, as Node.js does not support running multiple applications. It may be different if we choose to run a servlet[30] within an Enterprise Java Bean (EJB) [16] container, as one container can run multiple servlets. In such a case, the container would be part of the application instance, but the servlets and their connections to different databases would be part of the facets.

In order to compensate the missing application server in the application description, we introduce a placeholder service that provides an *application server* facet port. Then, the facet can bind to this port and instances will be created when a facet is created. Figure 3 also indicates that all component instances need to have facet instance-specific information for the wiring.

6 Beyond Facets and Port Boundaries

The introduction of port boundaries and facets is an important step towards the realisation of much more powerful COTs. They strengthen the self-containedness of component specifications, they allow establishing a clustering of component instances and hence support richer communication patterns than nowadays COTs. Finally, facets allows the re-using of an application infrastructure to support multiple data sets or communication routes. While all these aspects are important, the realisation of the two features opens up a set of further challenges and possibilities that has to be tackled in follow-up work. In the following paragraphs, we identify some topics that are interesting for ongoing and future work.

6.1 Resource Allocation

The fact that facets constitute an own layer of abstraction on top of applications introduces the need for mapping facet instances to component instances. In particular, this boils down to the following questions: *(i)* How to map facet instances over component instances. *(ii)* When to add new component instances and which ones to add.

To a large extent the possible mappings are dependent on the application semantics and the involved application components. For instance, a single load balancer instance can handle much more facets than a single DBMS instance. Taking the blog example, this means that a new DBMS instance has to be created and the existing facets be redistributed over the old and new DBMS instances.

The pattern of how to map different layers of abstractions on others is well known in the area of cloud deployments, because COTs already have to face it when dealing with component instances. When deployed, one component is conceptually bound to other components. On a technical level, however, it has to be mapped to a virtual machine instance. For cross-cloud deployments, it further has to be decided on what cloud this virtual machine shall be run. Such work may server as a starting point for developing mapping heuristics[24, 26, 32].

6.2 Clustering and Auto-scaling

Strongly related to the question of how to map facet resources to application resources (and these to virtual machine resources) is the questions of auto-scaling. With facets in place, scaling operations now need to be executed based on semantic knowledge: Instead of cloning an overloaded DBMS including all its databases, it might be beneficial to create a new DBMS and only migrate particular databases there, followed by a rewiring of affected application servers. In general, the following scenarios are worth considering: *(i)* when to add new application instance-level resources to better serve the needs on facet level; *(ii)* when to move facet resources to a different application instance-level resource in order to ensure quality levels or non-interference of tenants; *(iii)* when to distribute facet resources over multiple application-level and even IaaS resources;

We are aware that the heuristics needed to determine the best fitting point for performing one of the actions are strongly dependent on the application semantics as well as the available facets. Hence, there will be no general insight. Nevertheless, a further open question that remains is which language is powerful enough to on the one hand cater for all these requirements and on the other hand is sufficiently understandable to be used.

6.3 Advanced Communication Patterns

More complex distributed applications or components that can work in a distributed manner not always use request-response interaction patterns, but rely on other approaches such as multic-

asting or groupcasts[17, 18]. While the actual way to interaction is realised by the component implementation, the model of an application and its facets has to be able to reflect the needs of these interactions such that during deployment the correct links can be created from channels. In particular, many of these advanced communication protocols will be run within a cluster of component instances. For instance, distributed databases come with their own internal communication mechanism to distribute load and data amongst all members of the cluster. Hence, an application model would have to support channels from one component to itself.

Other scenarios that modelling shall be able to deal with is fault-tolerance. For instance, two clusters of application servers may both make use of their own database including an own DBMS. In turn, both DBMSs synchronise their state with a third DBMS in order to create a backup. Now, the application server clusters should only use the third DBMS when their actual DBMS has failed.

6.4 Deployment Workflow and Life-cycle Handling

The extensions introduced in Section 4 and 5 have a large impact on the deployment workflows and the life-cycle handling of both application instances and component instances, but also the wiring between instances. In particular, by using facets, it becomes necessary to instantiate a new application-dependent entity on the state of an already existing deployment. Moreover, it is unavoidable to introduce an own life cycle for facets and integrate it seamlessly in the life cycle of the other entities. Such a facet life cycle has to be able to deploy facet instance-related aspects on top of application instance-related aspects.

Considering the then four levels of deployment activities (virtual machines, applications, facets, components) makes it harder users/DevOps to implement the entire deployment workflow by hand. Instead, it should be the goal of a mature application specification language to provide a single, yet sufficiently powerful and well-defined deployment workflow including well-defined life-cycle handlers and the pre- and post-conditions to hold when they are invoked. Only such a deployment workflow caters for the best possible re-usability of components, freedom of errors, and abandons the need for a manual implementation of workflows.

In addition, it seems that the introduction of facets makes supporting migration of stateful applications (including databases) much more important, as facets to a large extent unveil the data aspects that so far have been ignored by most COTs. Supporting statefulness and data leads to the need to enhance the life cycle of components such that they support their own migration, but also to support the migration of facets hosted on them.

7 Conclusions and Future Work

Managing cloud applications running on IaaS is complicated and error prone. This is why cloud orchestration tools (COTs) have been emerging. COTs provide an application description language allowing the user to specify its application and application structure. Based on this specification, the tool will then deploy and manage the application.

In this paper, we identified two basic shortcomings of current COTs that were motivated through a sample three-tier application. Both of them are related to the way COTs compose applications from components and further allow the specification of communication channels between two components, *e.g.*, an application server and the DBMS. At runtime such a channel has to be mapped to dedicated links wired between component instances. Here, COTs either outsource the wiring logic to user-provided scripts or assume a one-to-all (anycast) mapping (any application server instance is linked to all DBMS instances). While the first approach

bears the risk of errors and also complicates automation, the latter hinders the introduction of application-specific wirings and with it, tenant-aware structures on a single application instance.

In order to identify the challenges, we took the sample application from a single-user, single-instance set-up to a multi-user, multi-instance set-up. In the latter case, an operator of the application is hosting the application on behalf of his customers. Naturally, the operator has the motivation to save resources and prefers, *e.g.*, to not run multiple DBMS with one database when one DBMS with multiple databases is sufficient. In order to do that, the operator has to exploit the internal structure of the application.

The discussion led to the insight that such scenarios indeed make it necessary that the application specification allows to logically separate instances of the business logic from the instances of the software component and to further separate instances of business logic from each other. Currently, this is not supported by any existing COTs. The discussion also made clear that the standard anycast assumption made by all current COTs is not powerful enough to support many multi-instance scenarios. Instead, mechanisms to express a clustering of component instances are needed.

In this paper, we introduced lower and upper bounds on both ends of channels. This approach allows clustering of component instances according to patterns such as 1-to-1, 1-to-N, and 1-to-M. Such patterns enable that technical constraints of the respective component can be expressed as well as those on the application structure.

Secondly, we introduced the concept of an application facet. A facet represents a logical layer of an application that is conceptually independent from the mere application infrastructure. A facet encapsulates mostly data aspects that may be instantiated multiple times within an application. Often, these data aspects will be isolated from each other and the decision of which facet (*i.e.*, which part of the entire data set) shall be accessed is dependent on the message flow through the system. For instance, a database (with its tables) uses the infrastructure provided by a DBMS and the decision which database to use is dependent on which user accesses the application. In addition, facet instances are widely independent from each other and multiple facets can be instantiated within an application and placed on top of the application infrastructure.

Finally, we discussed possible extensions of our proposals and clarified their impact on COTs, *e.g.*, their life-cycle handling, describing future directions of research.

Our future work centres around the realisation of the proposals made in this paper as well as to drive extensions thereof: In the next step, we will mainly experiment with the demands of large application deployments such as the multi-tenant, multi-blog scenario. This happens on two tracks: In the first track, we will extend our custom Cloudiator COT[19, 4] with the necessary capabilities to deal with facets and channel boundaries. Based on the experiments gained, we will refine both concepts. In the second track, we will also introduce the concepts into CloudML.

In parallel to this work, we investigate other use cases with possibly different demands. One of our favoured use case would incorporate disaster recovery or a backup strategy. We will also closely investigate the impact of facets on scaling rules and the auto-scaling capabilities of applications and slip the insights in the Scalability Rules Language[25] and approaches how different facet instances can share state.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 317715 (PaaS-age) and from the European Community's Framework Programme for Research and Innovation

HORIZON 2020 (ICT-07-2014) under grant agreement number 644690 (CloudSocket). We also thank the PaaSage consortium for valuable feedback on earlier versions of this paper.

References

[1] Amazon Web Services, 2015.

[2] CELAR EU project, 2015.

[3] Chef, 2015.

[4] Cloudiator Code Repository, 2015.

[5] Cloudify, 2015.

[6] Couchbase NoSQL database, 2015.

[7] Ghost — Just a blogging platform, 2015.

[8] Juju, 2015.

[9] MODAClouds EU project, 2015.

[10] Node.js, 2015.

[11] OpenTosca - Open Source TOSCA Ecosystem, 2015.

[12] PaaSage EU project, 2015.

[13] Pgpool, 2015.

[14] PostgreSQL, 2015.

[15] Rackspace: Leverage our cloud expertise to run fast and lean, 2015.

[16] Roberto Chinnici and Bill Shannon. Java platform, enterprise edition (Java EE) specification. Technical Report JSR-000316, Sun Microsystems, Inc., December 2009. Version 6.0.

[17] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33:427–469, December 2001.

[18] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36:372–421, December 2004.

[19] Jörg Domaschka, Daniel Baur, Daniel Seybold, and Frank Griesinger. Cloudiator: A cross-cloud, multi-tenant deployment and runtime engine. In *9th Workshop and Summer School On Service-Oriented Computing 2015*, 2015. accepted.

[20] Jörg Domaschka, Kyriakos Kritikos, and Alessandro Rossini. Towards a Generic Language for Scalability Rules. In *Proceedings of CSB 2014: 2nd International Workshop on Cloud Service Brokerage*, pages 206–220, 2014 (To Appear).

[21] Nicolas Ferry, Franck Chauvel, Alessandro Rossini, Brice Morin, and Arnor Solberg. Managing multi-cloud systems with CloudMF. In Arnor Solberg, Muhammad Ali Babar, Marlon Dumas, and Carlos E. Cuesta, editors, *NordiCloud 2013: 2nd Nordic Symposium on Cloud Computing and Internet Technologies*, pages 38–45. ACM, 2013.

[22] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In Lisa O'Conner, editor, *CLOUD 2013: 6th IEEE International Conference on Cloud Computing*, pages 887–894. IEEE Computer Society, 2013.

[23] Nicolas Ferry, Hui Song, Alessandro Rossini, Franck Chauvel, and Arnor Solberg. CloudMF: Applying MDE to Tame the Complexity of Managing Multi-Cloud Applications. In Randall Bilof, editor, *UCC 2014: 7th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 269–277. IEEE Computer Society, 2014.

[24] Abhishek Gupta, Dejan Milojicic, and Laxmikant V. Kalé. Optimizing vm placement for hpc in the cloud. In *Proceedings of the 2012 Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit*, FederatedClouds '12, pages 1–6, New York, NY, USA, 2012. ACM.

[25] Kyriakos Kritikos, Jörg Domaschka, and Alessandro Rossini. SRL: A Scalability Rule Language for Multi-Cloud Environments. In Juan E. Guerrero, editor, *CloudCom 2014: 6th IEEE International Conference on Cloud Computing Technology and Science*, pages 1–9. IEEE Computer Society, 2014.

[26] Gunho Lee, Byung-Gon Chun, and H. Katz. Heterogeneity-aware resource allocation and scheduling in the cloud. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pages 4–4, Berkeley, CA, USA, 2011. USENIX Association.

[27] Nicholas Loulloudes, Stalo Sofokleous, Demetris Trihinas, George Pallis, and Marios D. Dikaiakos. D2.1 – Application Description Tool V1. Celar project deliverable, July 2013.

[28] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[29] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Special Publication 800-145, National Institute of Standards and Technology, September 2001.

[30] Rajiv Mordani. Java servlet specification. Technical Report JSR-000315, Sun Microsystems, Inc., December 2009. Version 3.0.

[31] Derek Palma and Thomas Spatzier. Topology and Orchestration Specification for Cloud Applications (TOSCA). Technical report, Organization for the Advancement of Structured Information Standards (OASIS), June 2013.

[32] P. Pawluk, B. Simmons, M. Smit, M. Litoiu, and S. Mankovski. Introducing stratos: A cloud broker service. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 891–898, June 2012.

[33] Kaveh Razavi, Ana Ion, Genc Tato, Kyuho Jeong, Renato Figueiredo, Guillaume Pierre, and Thilo Kielmann. Kangaroo: A Tenant-Centric Software-Defined Cloud Infrastructure. In *Proceedings of the IEEE International Conference on Cloud Engineering*, pages 106–115, Tempe, AZ, USA, United States, March 2015.

[34] Alessandro Rossini, Juan de Lara, Esther Guerra, and Nikolay Nikolov. A Comparison of Two-Level and Multi-level Modelling for Cloud-Based Applications. In Gabriele Taentzer and Francis Bordeleau, editors, *ECMFA 2015: 11th European Conference on Modelling Foundations and Applications*, volume 9153 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2015.