

Pre-Proceedings of the ES OCC 2018 Workshops

Joint CloudWays and OptiMoCS Workshop

Vasilios Andrikopoulos, Nane Kratzke, Zoltán Ádám Mann, Claus Pahl
(Editors)

14th International Workshop on Engineering Service- Oriented Applications and Cloud Services

Luciano Baresi, Willem-Jan van den Heuvel, Andreas S. Andreou, Guadalupe Ortiz, Christian Zirpins, George Feuerlicht, Winfried Lamersdorf (Editors)

Abstract

This volume contains the papers presented at the WESOACS and Joint CloudWays and OptiMoCS workshops associated with the 7th European Conference on Service-Oriented and Cloud Computing, ES OCC 2018. The workshops were held in Como, Italy, on 12th September 2018. The workshops covered specific topics in service-oriented and cloud computing-related domains:

- 4th International Workshop on Cloud Migration and Architecture (CloudWays 2018)
- 1st International Workshop on Optimization in Modern Computing Systems (OptiMoCS 2018)
- 14th International Workshop on Engineering Service-Oriented Applications and Cloud Services (WESOACS 2018)

All papers presented at the workshops were selected through a rigorous review process, in which each submission was reviewed by at least three members of the workshops' program committees.

We as the workshop chairs would like to thank all authors for their submissions, and the reviewers for their work.

Table of Contents

CLLOUDWAYS AND OPTIMOCS

Performance Engineering for Kubernetes-style Cloud Cluster Architectures using Model-Driven Simulation 4
[CloudWays track]

Federico Ghirardini, Areeg Samir, Claus Pahl and Ilenia Fronza

On enhancing the orchestration of multi-container Docker applications 16
[CloudWays track]

Antonio Brogi, Claus Pahl and Jacopo Soldani

Transactional Migration of Inhomogeneous Composite Cloud Applications 28
[CloudWays track]

Josef Spillner and Manuel Ramírez López

Secure apps in the Fog: Anything to declare? 40
[OptiMoCS track]

Antonio Brogi, Gian-Luigi Ferrari, Stefano Forti

WESOACS

Towards a Generalizable Comparison of the Maintainability of Object-Oriented and Service-Oriented Applications 55

Justus Bogner, Bhupendra Choudhary, Stefan Wagner and Alfred Zimmermann

Implementation of a Cloud Services Management Framework 67

George Feuerlicht and Thai Hong Tran

Decentralized Billing and Subcontracting of Application Services for Cloud Environment Providers 79

Wolf Posdorfer, Julian Kalinowski, Heiko Bornholdt and Winfried Lamersdorf

May Contain Nuts: The Case for API Labels 90

Cesare Pautasso and Erik Wilde

On Limitations of Abstraction-Based Deadlock-Analysis of Service-Oriented Systems 102

Mandy Weissbach and Wolf Zimmermann

Performance Engineering for Kubernetes-style Cloud Cluster Architectures using Model-Driven Simulation

Federico Ghirardini, Areeg Samir, Ilenia Fronza, and Claus Pahl

Free University of Bozen-Bolzano, Bolzano, Italy
`firstname.surname@unibz.it`

Abstract. We propose a performance engineering technique for self-adaptive container cluster management, often used in cloud environments now. We focus here on an abstract model that can be used by simulation tools to identify an optimal configuration for such a system, capable of providing reliable performance to service consumers. The aim of the model-based tool is to identify and analyse a set of rules capable of balancing resource demands for this platform. We present an executable model for a simulation environment that allows container cluster architectures to be studied. We have selected the Kubernetes cluster management platform as the target. Our models reflect the current Kubernetes platform, but we also introduce an advanced controller model going beyond current Kubernetes capabilities. We use the Palladio Eclipse plugin as the simulation environment. The outcome is a working simulator, that applied to a concrete container-based cluster architecture could be used by developers to understand and configure self-adaptive system behavior.

Keywords: Container · Cluster · Kubernetes · Performance Engineering · Simulation

1 Introduction

Container management techniques such as Docker or Kubernetes are becoming widely used in cloud and other environments. making container-based systems self-adaptive involves the continuous adjustment of their computing resources in order to provide a reliable performance under different workloads. To achieve this, a well-designed autonomous elastic system should be built considering the following three key aspects: scalability, the ability of the system to sustain workload fluctuation, cost efficiency, acquiring only the required resources by releasing initialized ones, time efficiency, acquiring and releasing resources as soon as a request is made [5]. Moreover, whenever it is possible the system should also be fault tolerant, meaning it detects and handles failures effectively.

Therefore, we focus on investigating container cluster architectures for exploring and analyzing different performance and workload patterns, capable of enhancing reliability and validity for cluster management in container-based cloud environments. The main goal of our study is obtaining a reliable tool to be used

as a simulation environment for autonomous elastic systems. We aim to help finding suitable settings for the management of container-based cloud resources. While various simulation tools such as CloudSim exist, we focus here on an architecture model driven approach that allows application and platform architecture settings to be modelled and changed easily.

We use the container cluster management tool Kubernetes here that is now widely used in cloud environments as our platform facilitating self-adaptive systems. We use Palladio as the platform for modeling and simulation here.

An adaptive container system architecture can be abstracted and viewed as the inter-collaboration of three main parts: an application (or service) that is provided by the system, the container platform, and a monitor for analyzing resources being used and overall performance. As a consequence, the most suitable and logically applicable architectural pattern for such a system has been the MAPE-K architecture pattern (i.e., using a Monitor, Analyze, Plan, Execute and Knowledge implementation). We will provide here an abstract, but executable model for the (i) architectural aspects of platform and application and (ii) the controller for self-management. The model is essentially the configuration of a simulation environment. The first set of models (i) reflect the current Kubernetes platform, which we also use in the experimental evaluation. However, we also introduce an advanced controller model (ii) aiming to link observable performance anomalies to underlying workload problems that is going beyond current Kubernetes capabilities.

The paper is structured as follows. We start with background technologies in Section 2, then introduce the backbone of the architecture model in Section 3. We present our experimental findings in Section 4 and discuss related work in Section 5. As an extension, we look at important aspects of a controller model in Section 6, before ending with conclusions and future work in Section 7.

2 Self-Adaptive Systems – Background

For our autoscaling investigation, we follow the MAPE-K control loop, i.e., monitoring the performance of the application environment used by public users and tenants (i.e. a group of users who share a common access with specific privileges to the software instance), analyzing the just planned corrective actions, using the knowledge part (i.e. the Rule base) containing the autoscaling rules of the system. Autoscaling an application involves specifying threshold-based rules to implement elasticity policies for acquiring and releasing resources [?]. To give an example, a typical autoscaling rule might look as follows: IF the workload is high (e.g., > 80%) AND the response time is slow (e.g., > 600msec) THEN add/remove n instances.

2.1 Kubernetes Container Cluster Management

There are two essential Kubernetes concepts: at a macroscopic level, a system consists of the pod and the service. The management of elastic applications in

Kubernetes consists of multiple microservices, which communicate with each other. Often those microservices are tightly coupled forming a group of containers that would typically, in a non-containerized setup, run together on one server. This group, the smallest unit that can be scheduled to be deployed through Kubernetes, is called a pod. These containers are co-located, hence share resources and are always scheduled together. Pods are not intended to live long. They are created, destroyed and re-created on demand, based on the state of the server and the service itself.

Since pods have a short lifetime, there is no guaranteed IP address they are served at. For that reason, Kubernetes uses the concept of a service: an abstraction on top of a number of pods, typically requiring to run a proxy for other services to communicate with it via a virtual IP address. This is used to configure load balancing for pods and expose them as a service [7].

Kubernetes has been chosen as the container system for our investigation because of its autoscaling feature. Kubernetes is able to automatically scale up and down clusters. Once a cluster is running, there is the possibility of creating a Horizontal Pod Autoscaler (HPA). When defining an HPA there is the possibility to declare the exact number of pod replicas that the system should maintain (e.g., between 1 and 10). So, the autoscaler will increase and decrease the number of replicas (via deployment) to maintain an average CPU utilization of 50% (default setting) across all pods [8]. We will create simulator for this.

2.2 Palladio Modelling and Simulation

Palladio has been chosen as the simulation platform, not only for its advanced simulation tools, but also for its architectural modeling capabilities. With Palladio we can prototype and adjust also the application and platform architecture of a system under investigation. Palladio provides several models capable of specifying the architecture and carrying out simulations, see Fig. 1. In Palladio, each model is built on top of the previous one. Palladio is Eclipse-based, thus requiring all models to be grouped inside a single Eclipse project directory.

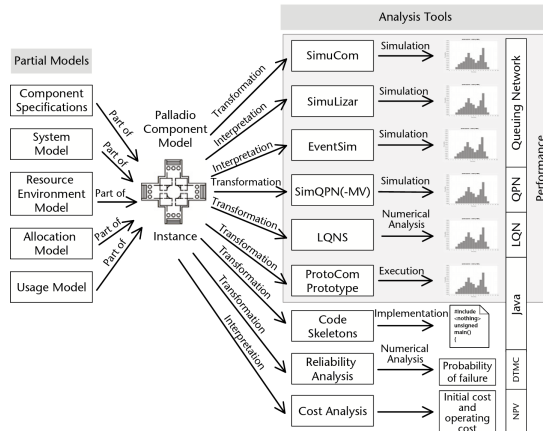


Fig. 1. Palladio Simulator Environment, from [15].

3 Architecture Model

The Palladio modeling tool allows us to specify a software system architecture in order to run simulations on these systems. We discuss now how the Kubernetes

containerized architecture has been abstracted and recreated inside Palladio in order to create a simulation tool (called KubeSim). In the following, we introduce the different structural and behavioural models.

3.1 Component Repository Model and SEFF diagrams

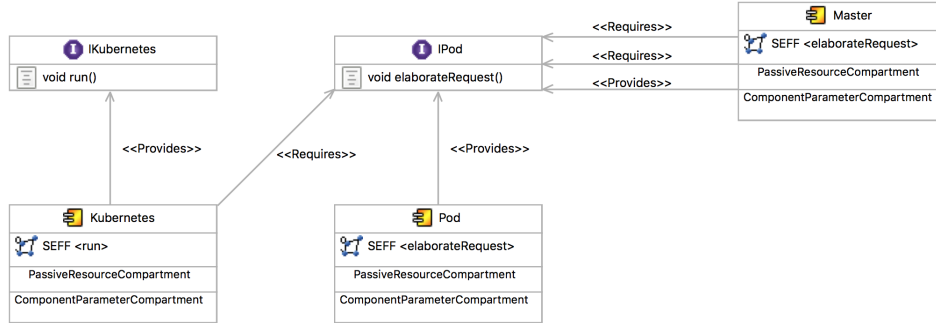


Fig. 2. Component Repository Model for all Simulations.

A Component Repository Model describes interfaces, components and dependencies between elements of the system architecture. Fig. 2 illustrates how the Kubernetes architecture has been abstracted and represented as a Repository model. In the model, a service is provided through Kubernetes, from pod governed databases, and accessible to a user via an internet connected device. For that reason, two interfaces are declared: one for the Kubernetes system (IKubernetes), performing a void run() action (simulate service up and running system call), and another for the pod component (IPod). Furthermore, one component has been created for the IKubernetes interface, named simply Kubernetes, and one for the IPod, named Pod. The Master component acts as the controller for load-balancing based on self-adaptive resource utilization rules.

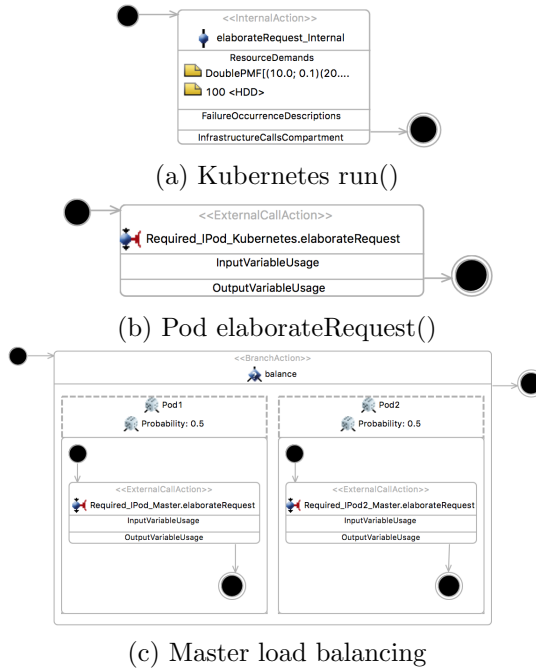


Fig. 3. System Behaviour Models (SEFFs).

Each component of the Component Repository Model has its own Service Effect Specification (SEFF), expressed in the form of a behavioral model. Fig. 3 shows the three SEFF models for the Kubernetes, Pod and Master component actions. Since the Kubernetes component requires the IPod interface to work, its behavior is reflected in an ExternalCallAction for the elaborateRequest() action. The action on the Pod component is performed internally. Moreover, in this SEFF diagram we specified actual resource demands of the system call. Resource demands are specified for the CPU (computational resource) and the HDD (storage resource) as hardware resources, in the form of stochastic expressions for work-units per second. The last Service Effect Specification models two components instead of only one: Pod1 and Pod2. This reconnects with Component Repository Model and the two arrows exiting from the Master component representing two instances of a Pod element. Note, that for simplicity, we present two sample pods here.

Here the execution flow is executed by a so called BranchAction, that has the task to distribute and balance the workload between the pod components. In this case, it is configured to reflect the default Kubernetes balancing rule that distributes the work evenly across all system pods. However, this setting could be varied in experiments and used like a virtual knob to tune balancing settings of the Master controller to whatever value of interest.

3.2 System Model

The System Model captures the composite structure of the whole Kubernetes systems architecture (not displayed as a diagram for space reasons). The system architecture for this model uses the available components declared in the Component Repository Model to constitute a complete component-based software system. This model includes dependencies between the various assembly contexts (i.e., components) of the Kubernetes architecture. The entire system provides its service over the Kubernetes platform, i.e., through the IKubernetes interface. This interface is connected to the assembly elements representing the Kubernetes component. Since Kubernetes requires pods to run the service, it is connected with a component providing the IPod interface. However, because the cluster is self-adaptive, we cannot directly connect the Kubernetes assembly to the pods, but use the Master controller node as an intermediary. The system also requires two pod interfaces. We only need to instantiate two assembly contexts for two pods and then connect the two with the Master component.

3.3 Execution Environment and Component Allocation Models

Based on the system model, we declare and allocate resources for our system environment. For that, there are the so called Deployment Models, which include the Execution Environment Model and the Component Allocation Model.

We have three resource components: Kubernetes, Pod1 and Pod2. Each pod has a CPU unit with scheduling policy set to Processor Sharing (that is an approximation of a Round-Robin resource management strategy), and an HDD

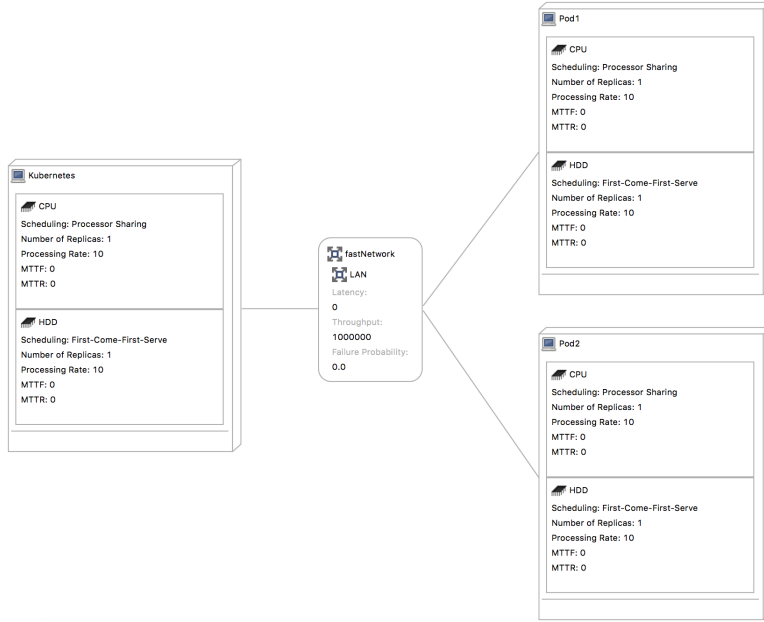


Fig. 4. Execution Environment Model configured for a Single Experiment.

with scheduling policy set to First-Come-First-Serve (that is a typical behavior for hard disk drives). The Processing Rates of CPUs and HDDs can vary, therefore the values in this model are purely indicative of one single experiment configuration. The Execution Environment Model, see Fig. 4, also provides other settings for resources, like the Number of Replicas, Mean Time To Failure (MTTF) and Mean Time To Repair (MTTR). In order to focus on performance, these have been set to standard values, i.e., resp., 1, 0.0 and 0.0. Kubernetes containers also have CPU and HDD declared resource demands (with Processing Rate set to 10 for both), with default settings for Number of Replicas, MTTF and MTTR. The three containers are connected via a LinkingResource component, that could act as a fast network, with Latency set to 0, Throughput set to 1,000,000 and Failure Probability set to 0.0.

3.4 Usage Model

The Usage Model contains a Service Effect Specification diagram specifying the system call. The Usage Model provides two different workloads for the system under study: an OpenWorkload and a ClosedWorkload. For the OpenWorkload, the user interarrival time could be specified in seconds, and the number of users coming to use the system will vary from one

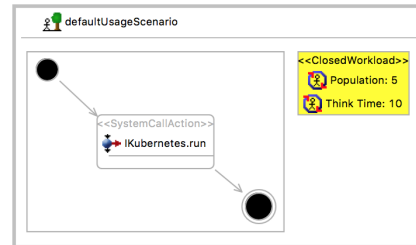


Fig. 5. Usage Model configured for one Experiment.

simulation run to the other. With the ClosedWorkload we can specify the user population (i.e. the number of active users in our system), and also the single user think time (i.e. the pause the user after each run() action, in seconds).

4 Evaluation and Discussion

In the experimental evaluation, we focus on simulations of the Kubernetes implementation as it is currently available, with the HPA component.

4.1 Experimental Evaluation

Our aim was to aid specification for container cluster scaling rules. The main experimental goal for the project focused on evaluating suitable system performance. We translate this into a simple rule: keep idle time less or equal to 50% (not to waste resource power) and concurrent active job time less or equal to 25% (not to experience long overload periods that impacts on performance), for both CPU and HDD components of the pods.

As a starting point, we considered different workload patterns, distinguished in terms of three qualitative values: low, medium and high, i.e., workloads without unexpected fluctuations in relation to the three main values.

To set a desired workload inside Palladio, we use the SEFF diagram describing the core system function, and specify the resource demand in the form of a stochastic expression. In our case, the SEFF diagram to be modified is the one of elaborateRequest() action, see Fig. 3 above. We keep the HDD expression fixed at 100 processing unit rate for all workload types, while for the CPU component the stochastic values (expressed in a joint Probability Mass Function with double values, i.e. DoublePMF) that have been used for the different workloads are: DoublePMF[(10.0;0.1)(20.0;0.8)(30.0;0.1)] for low, i.e. 10% of the time the CPU power used is being used at 10%, 80% of the time it is being used at 20% of power and in the remaining 10% of the time it is being used at 30% of power; and correspondingly DoublePMF[(40.0;0.1)(50.0;0.8)(60.0;0.1)] for medium and DoublePMF[(70.0;0.1)(80.0;0.8)(90.0;0.1)] for high.

The specification of a pod resource demand can be adjusted. Particularly, the CPU and HDD processing rate are the ones in which we are highly interested in, because they reflect the specification rules for assigning Kubernetes pods resource demands limits. The variable field that need to be changed for this time is the Execution Environment model. Fig. 6 shows the worst and best case results for CPU and HDD resources for different workload patterns and assumed processing rates. The values for CPU and HDD processing rate varied from 2 to 18 during different simulations where we followed an experimental progression based on observations obtained through the different simulating runs.

For another set of experiments, we also changed the population number, which describes the number of active users inside the system at simulation time, see Fig. 7,. The aim here was to better judge the impact the number of active users could have on overall system performance. We tested the system with

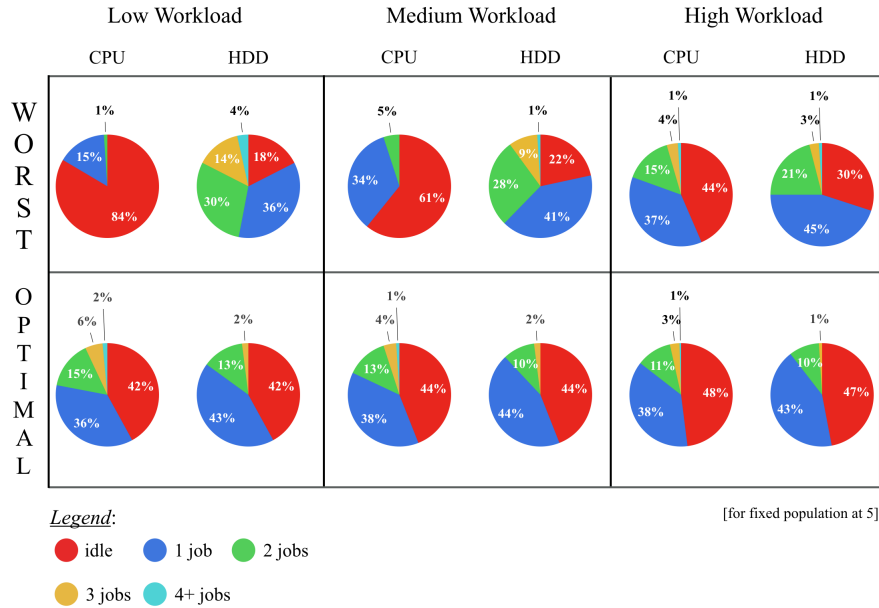


Fig. 6. Resource Utilization (idleness) for different Workloads – Best and Worst Case.

1, 3 and 5 users that were equally distributed between the pods, showing an increasingly reduced idle for higher CPU loads as the population increases.

4.2 Discussion

Our paper has focused on creating an environment to simulate the behavior of self-adapting (scaling) container cluster architectures. We presented the models implemented in the Palladio environment, thus creating a simulation bench by defining the architecture of a systems and its resource. We have demonstrated that running simulations of applications with Kubernetes autoscaling strategies allows investigating the architectural structure of a system and its behavioural properties. This can lead to greater efficiency in implementations as the sample resource utilization experiments have shown. KubeSim tool is useful when trying to obtain specification values to identify and configure for the controller. We were able to understand and investigate underlying functions and characteristics of self-adaption in Kubernetes – for example the case we experimentally observed that CPU and HDD performances were impacted by each other’s settings. KubeSim is thus beneficial for application developers aiming to use Kubernetes.

We also look at limitations and threats to the validity of our work. We can start our the threats to validity analysis by looking to a central and potential critical aspect of our work, the experiments sample field. While running simulations for KubeSim, we considered only a small portion of experimental values. Concretely, we restricted our sampling field as follows:

- Architecture: we took a scenario with a simplifying two pod system for illustrative purposes. We can, however, assume that our experimental results

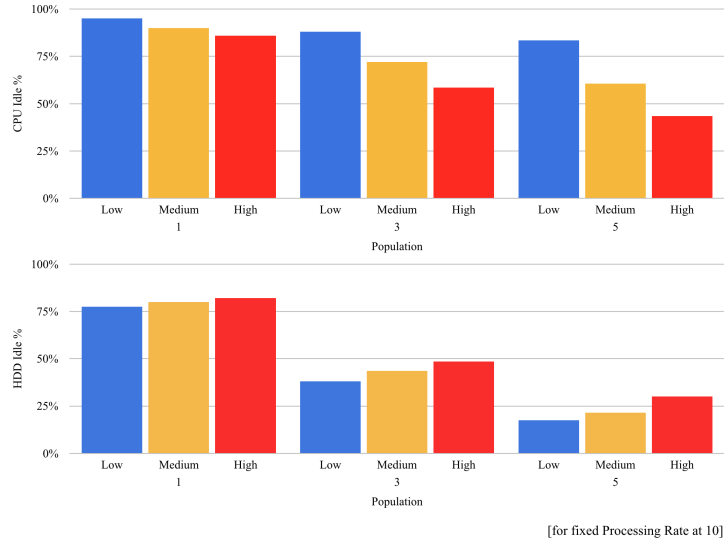


Fig. 7. Aggregated Workload Results for different User Populations.

- would apply also to bigger systems composed by more than two pods and used by a greater population of users as the results so far indicate linearity.
- Uncertainty: KubeSim was not exposed to unpredictable workloads and failures (as in real world platforms), thus restricting even more our sample fields and leading to more uncertainty in the validity of the results.
 - Scale: applies to all KubeSim settings, for which we used small numerical values for input variables (e.g., processing and user think time in the usage model). As argued above, linearity here is possible, but not yet proven.

A final remark on restricting the sample field is in regards to the load balancer policy, for which we only tested the implemented equal balancing load rule. Our advanced controller model is a first step towards proposing an improved Kubernetes scaling strategy, which however is beyond the scope of this paper.

However, evaluating our simulation environment under all possible scenarios was beyond the scope of this paper. The overall aim was to point out a valid alternative to already present performance engineering and evaluation methods for self-adaptive container cluster systems (i.e. those who separates software engineering analysis for the architecture and the autoscaling strategies parts).

5 Towards an Advanced Controller Model

The controller implemented in Kubernetes and modelled above uses equal workload distribution as the load balancing strategy. One of our goals is to explore advanced controller settings for Kubernetes that could be implemented in an improved HPA component. Our proposal shall take into account that platform and application are not controlled by the same organisation, i.e., that some load properties of platform resources (i.e., Kubernetes core components offered by a

cloud provider) are not visible for the Kubernetes user. The general situation is that in shared virtualised environments, third parties provides some resources that can be directly observed (e.g., using performance metrics such as response time) while others remain hidden from the consumer (e.g., the reason behind performance or workload anomalies, the dependency between the affected nodes and container, etc). In order to improve the workload balancing and autoscaling capability, we can enhance the MAPE-K based controller here. We introduce a core model for anomaly detection and analysis for a cluster environment that automatically manages resource workload fluctuations. This can be implemented as an extension of the Palladio model towards dynamic auto-scaling, which in the current version only considers a static load balancing strategy.

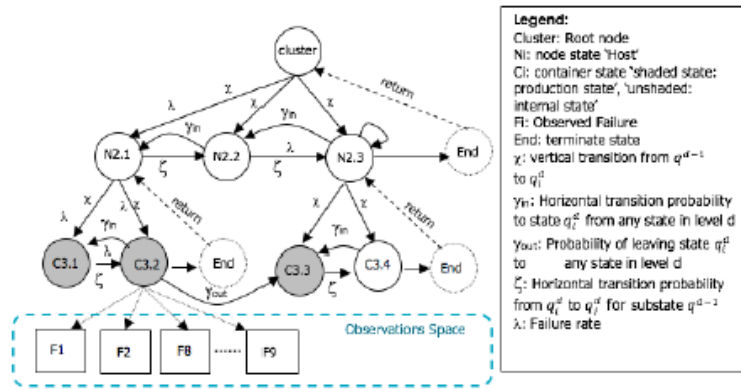


Fig. 8. HHMM for Auto-Scaling Workload for Cluster-Container Environments.

We differentiate two situations in which response time fluctuations occur:

- Hidden states might show anomalous behavior of the resource that might needs to be remedied by the controller (unwanted behavior such as overload, or appreciated behavior like underload).
- Emission or observation of behaviour for the user (indicating possible failure), which might result in failure if caused by a faulty hidden state.

To address this, we propose Hidden Markov Models (HMMs) to map the observed failure behaviors of a system resource to its hidden anomaly causes (e.g., overload) and to predict the occurrence of the anomaly in the future. A Hidden Markov Model (HMM) is a statistical Markov model in which the system being modeled is assumed to be a Markov process with hidden states. In simple Markov models, state transition probabilities are the only parameters, while in the hidden Markov model, there are observable emissions (e.g., in the form of response time data), dependent on the workload state.

To reflect the layered nature of application, platform and infrastructure in a Kubernetes system, we utilize a specific variant of HMM, that of Hierarchical Hidden Markov Models (HHMM) [2], see Fig. 8. HHMMs are better suited here than HMM in describing the uncertain probability distribution at node and

cluster level. HHMM generate sequences by a recursive activation of one of the substates (nodes) of a state (cluster). This substate might also be hierarchically composed of substates (here pods). The HHMM decides on transition probabilities a possible (hidden) cause for an observed anomaly and then decides how to transfer load between nodes to reduce undesired performance degradations.

Each state hidden state (internal node, production pod, root cluster) may be in an overload, underload or normal load state. Each workload is associated with response time observations that are emitted from a production state. The existence of anomalous workload in one state not only affects the current state, but possibly substates on the other level. The edge direction in the figure indicates the dependency between states. We can use vertical transitions for identifying the fault and we use horizontal transitions to show the fault that exists between states and to trace the fault/failure mapping between states at the same level.

6 Related Work

In [5,6], resource management and auto-scaling rules for self-adaptive systems was investigated, focused mainly on VM-based cloud platforms. As part of their experimental approach, a parameterized architecture model when running simulations has been used. That is, simulations were run on base of stochastic expressions, which reflected each systems component behavior. This allowed fine adjustments while setting adaptation rules for the simulation environment. We have followed these and similar approaches, but add a novel perspective in the advanced controller model here that takes the hidden/observable distinction into account for a hierarchically organised architecture.

To the best of our knowledge, there is no Kubernetes simulation environment. Models of performance concerns and resource management has been discussed [11], but a simulation tool has not been created.

In addition to work on models for self-adaptive cloud systems for performance and workload management, we also look at the simulation tool landscape. This allows us to justify our decision to choose Palladio as the model. Often, load balancing strategies can be formalised and simulated in tools like MatLab. However, this would not allow us to model the architecture in terms of applications, platform and infrastructure concerns. As we target the KubeSim tool to application developers and users of Kubernetes and similar tools, an explicit architecture model is of critical importance. The same argument also applies to other simulation tools such as CloudSim [1].

7 Conclusions

We investigated performance engineering solutions for self-adaptive container cluster systems, with the purpose of finding an efficient way to determine and express autoscaling rules for such systems, in order to improve platform settings.

We created a simulation tool for Kubernetes using the Palladio platform, capable of delivering an easy to use simulation bench. KubeSim offers a developer

the possibility of testing such a system by tuning different settings and metrics of the system. In fact, the novelty of KubeSim as a Kubernetes performance simulation tool is to enable reliable performance analysis with the effort of having to implement prototype implementations. With the advanced controller model, we also target a deeper investigation beyond application development.

As future work, KubeSim could include the possibility of considering faults types. Another improvement would be considering sensor noise. That is assuming and considering that the systems sensor is exposed to noise derived from hosting platform connection. Again, a last upgrade, always related to fault consideration, could be implementing a fault prediction algorithm, so that the system would be aware of an oncoming error and scale resource on its base.

Another aim is to integrate the advanced controller model in KubeSim. This would allow studying alternative strategies for the Kubernetes HPA component.

References

1. CloudSim: A Framework for Modeling and Simulation of Cloud Computing Infrastructures and Services. Retrieved from: <http://www.cloudbus.org/cloudsim/>.
2. S. Fine, Y. Singer, N. Tishby. The hierarchical hidden markov model: Analysis and applications. *Machine Learning*, 32:4162, 1998.
3. R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L.E. Lwakatare, C. Pahl, S. Schulte, J. Wettinger. Performance Engineering for Microservices: Research Challenges and Directions. *Intl Conference on Performance Engineering Companion*. 2017
4. P. Jamshidi, C. Pahl, N.C. Mendonca, J. Lewis, S. Tilkov. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* 35 (3), 24-35. 2018.
5. P. Jamshidi, A. Ahmad, C. Pahl, *Autonomic Resource Provisioning for Cloud-Based Software*, SEAMS, 2014.
6. P. Jamshidi, C. Pahl, N.C. Mendonca, *Managing Uncertainty in Autonomic Cloud Elasticity Controllers*, *IEEE Cloud Computing*, 2016.
7. Introduction to Kubernetes. Retrieved from: <https://x-team.com/blog/introduction-kubernetes-architecture/>. 2018.
8. Autoscaling in Kubernetes. Retrieved from: <http://blog.kubernetes.io/2016/07/autoscaling-in-kubernetes.html>. 2018.
9. H.C. Lim et al., *Automated Control in Cloud Computing: Challenges and Opportunities*, *Workshop Automated Control for Datacenters and Clouds*, 2009.
10. T. Lorido-Botran, J. Miguel-Alonso, J.A. Lozano, *A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments*, *J. Grid Comp* 12(4), 2014.
11. V. Medel, O. Rana, J.A.I Banares, U. Arronategui. *Modelling performance & resource management in kubernetes*. *Intl Conf on Utility and Cloud Comp*. 2016.
12. C. Pahl, S. Helmer, L. Miori, J. Sanin, B. Lee. *A container-based edge cloud PaaS architecture based on Raspberry Pi clusters*. *Future IoT & Cloud Workshop*. 2016.
13. C. Pahl, A. Brogi, J. Soldani, P. Jamshidi. *Cloud Container Technologies: a State-of-the-Art Review*. *IEEE Trans. on Cloud Computing*. 2017.
14. Palladio Simulator. Retrieved from: http://www.palladio-simulator.com/about_palladio/. 2018.
15. R.H. Reussner, et al. *Modelling and Simulating Software Architecture – The Palladio Approach*, 2016.
16. L.M. Vaquero, L. Rodero-Merino, R. Buyya, *Dynamically Scaling Applications in the Cloud*, *ACM SIGCOMM Computer Comm. Rev.* 41(51), 2011.

On enhancing the orchestration of multi-container Docker applications

Antonio Brogi¹, Claus Pahl², and Jacopo Soldani¹

¹ University of Pisa, Pisa, Italy {brogi,soldani}@di.unipi.it

² Free University of Bozen-Bolzano, Bolzano, Italy claus.pahl@unibz.it

Abstract. After introducing Docker containers in a nutshell, we discuss the benefits that can be obtained by supporting enhanced descriptions of multi-container Docker applications. We illustrate how such applications can be naturally modelled in TOSCA, and how this permits automating their management and reducing the time and cost needed to develop such applications (e.g., by facilitating the reuse of existing solutions, and by permitting to analyse and validate applications at design-time).

Keywords: Docker · TOSCA · orchestration · cloud applications

1 Introduction

Containers are emerging as a simple yet effective solution to manage applications in PaaS cloud platforms [18]. Containers are also an ideal solution for SOA-based architectural styles that are emerging in the PaaS community to decompose applications into suites of independently deployable, lightweight components, e.g., microservices [2,24]. These are natively supported by container-based virtualisation, which permits running components in independent containers, and allows containers to interact through lightweight communication mechanisms.

However, to fully exploit the potential of SOA, container-based platforms (e.g., Docker — www.docker.com) should enhance their support for selecting the containers where to run the components of an application, and for orchestrating containers to build up a multi-container application. To that end, there is a need for a modelling language to describe the features offered by a container (to satisfy the requirements of an application component), to orchestrate containers to build multi-container applications, and to deploy and manage them in clusters.

Our objective here is to highlight the need for such language, by illustrating its potential benefits on a concrete containerisation framework like Docker. In this context, the main contributions of this paper are:

1. We discuss the benefits, but also the limitations of Docker, specifically with respect to composition and orchestration in multi-container applications.
2. We propose a way to represent multi-container Docker applications in TOSCA [16], the OASIS standard for orchestrating cloud applications, as an example to discuss the advantages of enhancing their orchestration (e.g., easing the selection and reuse of existing containers, reducing time and cost for developing multi-container applications, automating their management, etc.).

This paper is organised as follows. Sect. 2 provides some background on TOSCA. Sects. 3 and 4 provide an introduction to Docker and discuss its current benefits and limitations, respectively. Sect. 5 discuss the advantages of enhancing the orchestration of multi-container Docker applications with TOSCA. Finally, Sects. 6 and 7 discuss related work and draw some concluding remarks, respectively.

2 Background: TOSCA

TOSCA [16] (*Topology and Orchestration Specification for Cloud Applications*) is an OASIS standard for specifying portable cloud applications and automating their management. TOSCA provides a modelling language to describe the structure of a cloud application as a typed topology graph, and its management tasks as plans. More precisely, each applications is represented as a *service template* (Fig. 1), consisting of a *topology template* and of optional management *plans*.

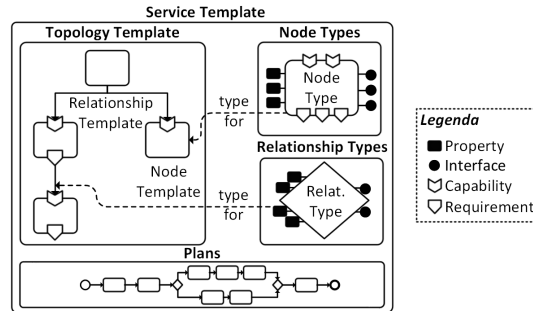


Fig. 1. TOSCA *service template*.

The topology template is a typed directed graph describing the structure of an application. Its nodes (*node templates*) model the application components, while its edges (*relationship templates*) model the relationship among those components. Both node templates and relationship templates are typed by means of *node types* and *relationship types*. A node type defines (i) the observable properties of a component, (ii) its requirements, (iii) the capabilities it offers to satisfy other components' requirements, and (iv) its management operations. Relationship types describe the properties of relationships occurring among components.

Plans permit describing the management of an application. A plan is a workflow orchestrating the management operations offered by the application components to address (part of) the management of the whole application.

3 Docker in a nutshell

Docker is a Linux-based platform for developing, shipping, and running applications through container-based virtualisation. Container-based virtualisation ex-

exploits the kernel of the host’s operating system to run multiple guest instances. Each guest instance is called a *container*, and each container is isolated from others (i.e., each container has its own root file system, processes, memory, devices and network ports).

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

A Docker image is a read-only template providing the instructions for creating a container. It is built by layering multiple images, with the bottom image being the *base* image, and with each image being the *parent* of the image right above it. A Docker image can be created by loading a base image, by performing the necessary updates to that image, and by *committing* the changes. Alternatively, one can write a *Dockerfile*, which is a configuration file containing the instructions for building an image³.

It is also possible to look for existing images instead of building them from scratch. Images are stored in *registries*, like Docker Hub (hub.docker.com). Inside a registry, images are stored in *repositories*. Each repository is devoted to a software application, and it contains different versions of such software. Each image is uniquely identified by the name of the repository it comes from and by the tag assigned to the version it represents, which can be used for retrieving it.

Volumes. Docker containers are volatile. A container runs until the stop command is issued, or as long as the process from which it has been started is running. By default, the data produced by a container is lost when a container is stopped, and even if the container is restarted, there is no way to access to the data previously produced. This is why Docker introduces *volumes*.

A volume is a directory in a container, which is designed to let data persist, independently of the container’s lifecycle. Docker therefore never automatically deletes volumes when a container is removed, and it never removes volumes that are no longer referenced by any container. Volumes can also be used to share data among different containers.

Docker orchestration. The term orchestration refers to the composition, coordination and management of multiple software components, including middleware and services [15]. In the context of container-based virtualisation, this corresponds to multi-component applications, whose components independently run in their own containers, and talk each other by exploiting lightweight communication mechanisms.

Docker supports orchestration with *swarm* and *compose*. Docker swarm permits creating a cluster of Docker containers and turning it into a single, virtual Docker container (with one or more containers acting “masters” and scheduling incoming tasks to “worker” containers). Docker compose permits creating multi-container applications, by specifying the images of the containers to run and the

³ The latter provides a more effective way to build images, as it only involves writing some configuration instructions (like installing software or mounting volumes), instead of having to launch a container and to manually perform and commit changes.

interconnections occurring among them. Docker compose and Docker swarm are seamlessly integrated, meaning that one can describe a multi-component application with Docker compose, and deploy it by exploiting Docker swarm.

4 Benefits and limitations of Docker

Docker containers feature some clear benefits. Firstly, they permit *separation of concerns*. Developers only focus on building applications inside containers and system administrators only focus on running containers in deployment environments. Previously, developers were building applications in local environments, passing them to system administrators, who could discover (during deployment) that certain libraries needed by the applications were missing in the deployment environments. With Docker containers, everything an application needs to run is included within its container.

Docker containers are also *portable*. Applications can be built in one environment and easily shipped to another. The only requirement is the presence of a Docker engine installed on the target host.

Furthermore, containers are *lightweight* and fast to launch. This reduces development, testing and deployment time. They also improve the horizontal scalability of applications, as it is easy to add or remove containers whenever needed.

On the other hand, limitations do exist. Docker currently does not support *search* mechanisms other than looking for the name and tag of an image inside a registry [9]. There is currently no way to describe the internals of an image, e.g., the features offered by a container instantiated from an image, or the software it supports. A more expressive description of images would enable more powerful reuse mechanisms (e.g., adaptation of existing images), hence reducing the time needed to retrieve images and develop container-based applications.

Further limitations affect the *orchestration* of complex applications. Consider, for instance, a multi-component application made of three components, i.e., a web-based GUI, which depends on a back-end API to serve its clients, which in turn connects to a database to store application data. Currently, Docker does not provide a way to describe the runtime environment needed to run each component. A developer is hence required to manually select the image where to run each component, to extend it (by adding the component and its runtime dependencies), and to package it into a new image. The obtained images can then be composed with Docker compose to build a multi-container Docker application, which however has no explicit information about which component is hosted on which image nor on which component is interconnected on each other. Everything is hidden in a kind of “black-box” view due to the lack of information on the internals of Docker containers [10].

It is not possible to distinguish between simple dependencies determining the deployment ordering from persistent connections to set up. For instance, in the aforementioned application, Docker compose would include two interconnections, one between the containers packaging the GUI and the API, and one between those packaging the API and the database. However, the former interconnection

may be unnecessary (especially in a multi-host deployment scenario), as the GUI may not require to set up a connection to the API. The GUI may indeed just require to be deployed after the API and to be configured so to forward user queries to the actual endpoint offered by the API.

Additionally, despite Docker compose and Docker swarm are seamlessly integrated, limitations do exist⁴. For instance, when a compose application is deployed with swarm, the latter may not be able to manage all interdependencies among containers, which may result in deploying all containers on the same host or in not being able to automatically deploy all containers. In the latter case, one would hence be required to manually complete the deployment.

A more expressive specification language (e.g., TOSCA [16]) would permit overcoming these limitations. By describing the environment needed to run an application component, it would be possible to specify what the component needs, and then to automatically derive the images of the underlying infrastructure (e.g., by exploiting existing reuse techniques [11,20]). It would also permit describing the management of a complex multi-component application by orchestrating the management of its components.

5 Orchestrating multi-container applications in TOSCA

In this section, we first show how multi-container applications can be represented in TOSCA (Sect. 5.1). We then illustrate how this permits enhancing the orchestration of multi-container Docker applications (Sect. 5.2), as well as better exploiting container-oriented design patterns (Sect. 5.3).

5.1 Multi-container applications in TOSCA

A multi-container application essentially corresponds to a multi-component application, where each component is hosted on a container. A multi-container Docker application can be represented by a TOSCA service template, whose topology nodes represent the application components, the containers they need to run, and the volumes that must be mounted by containers. The relationships instead model the dependencies between components, containers and volumes (e.g., hosting a component on a container, connecting components and/or containers, or attaching a volume to a container). Plans then orchestrate the operations offered by the nodes to describe the management of a whole application.

We hence need the types to include the above mentioned nodes and relationships in a topology template. For the nodes, we can exploit the TOSCA types defined in [10], which permit distinguishing (a) *Software* components, (b) *Containers* and (c) *Volumes* (Fig. 2). For the relationships, we can instead rely on the TOSCA normative relationship types [16].

Without delving into the details on the modelling (which can be found in [10]), we show how it can be exploited to represent the multi-container Docker

⁴ A thorough discussion on this is available at docs.docker.com/compose/swarm.

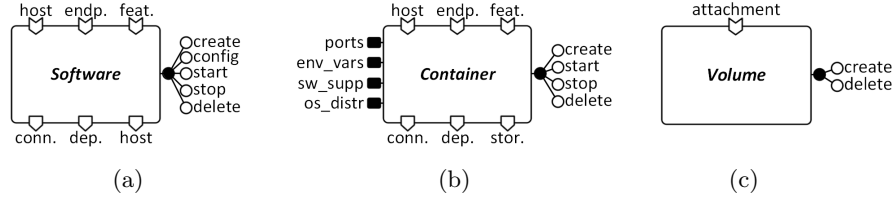


Fig. 2. TOSCA node types for multi-container Docker applications [10].

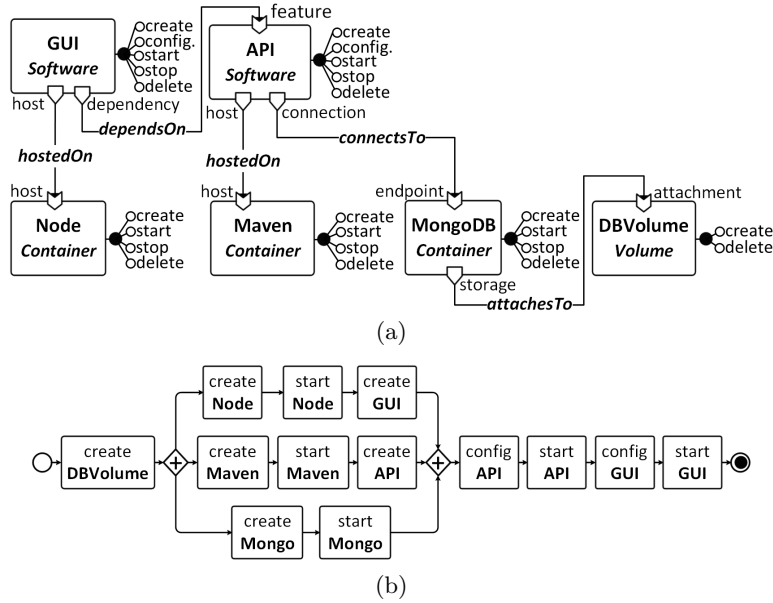


Fig. 3. Examples of (a) a topology template modelling a multi-container Docker application, and of (b) a plan orchestrating its deployment.

application mentioned in Sect. 4 (which consists of three interconnected components — i.e., a GUI, an API and a database). With the above mentioned TOSCA types, we can go beyond Docker compose (that only permits identifying the images of three containers, and specifying the interconnection occurring between them). As illustrated by Fig. 3.(a), we can describe components and containers separately, and explicitly specify which container is hosting a component (e.g., the API is hosted on the Maven container), the dependencies among components (e.g., the API connects to the database) and the necessary volumes.

We can also program the management of the application as a whole. Each component indeed exposes the operations that permit managing its lifecycle, which can then be orchestrated to accomplish application management tasks. A concrete example is given in Fig. 3.(b), which displays a BPMN-like plan orchestrating the deployment of the multi-container Docker application in Fig. 3.(a).

5.2 Orchestrating multi-container applications with TOSCA

One may argue whether the effort of defining multi-container Docker applications in TOSCA really pays off. As we anticipated in Sect. 4, a model like that discussed in the previous section permits enhancing the orchestration of multi-container Docker applications in three main ways.

Searching for images Docker search capabilities are currently limited, as Docker only permits looking for names and tags of images in registries [9]. TOSCA permits overcoming such limitation, as it also permits describing the internals of an image (like the features that will be offered by a container instantiated from an image, or the software distributions it will support). This enables more powerful discovery mechanisms.

For instance, in [8] we show how to automatically discover the Docker containers offering the runtime support needed by the components forming an application. The *host* requirements of GUI and API can be left pending, by only describing the runtime capabilities that must be provided by the container that can satisfy them (e.g., which software distribution they must support, which operating system they must run, etc.). Then, a concrete implementation like that in Fig. 3 can be automatically derived by reusing existing Docker containers. As we illustrated in [8], such an approach can drastically reduce the time and costs needed for developing and maintaining container-based applications

Design-time validation TOSCA permits explicitly indicating which are the requirements needed to run a component, which capabilities it provides to satisfy the requirements of other containers, and how requirements and capabilities are bound one another. This permits validating multi-container applications at design-time, by checking whether the requirements of a component have been properly satisfied (e.g., with the validator presented in [6]).

The same is currently not possible with Docker, which is not providing enough information to determine whether all interdependencies have been properly settled. This is because Docker compose only permits listing the images of containers to run, and the interconnections among them.

Automation of application management We can exploit TOSCA *plans* to describe only once all the management of an application (by orchestrating the management operations of the application components). For instance, we can program how to coordinate the deployment of all the components of an application with a single *plan* (as exemplified in Fig. 3.(b)).

The management of multi-container Docker applications can be further automated by exploiting management protocols [5], which permit specifying the behaviour of the nodes forming a TOSCA application, and to automatically derive the management behaviour of an application by composing the management protocols of its nodes. This permits automating various useful analyses, like determining whether management plans are valid, which are their effects (e.g., which application configuration is reached by executing a plan, or whether it generates faults), or which plans permit reaching certain application configurations or recovering faulted applications.

The above remarks highlight how Docker (and, more generally, a container-based framework) would enhance its orchestration capabilities by being integrated with expressive specification languages, such as TOSCA. This is also concretely illustrated and evaluated by the work in [8] and [10]. The latter present the components of the TOSKER open-source environment, by also empirically showing how the integration of Docker with a language like TOSCA can provide some of the aforementioned benefits.

It is also worth noting that, even if we exploited a simple application (with just three components) for illustration purposes, this is sufficient to illustrate the positive impact of an enhanced orchestration support for Docker-based frameworks. By considering complex enterprise applications [12], which can contain a much higher number of interdependent components, the potential and impact of allowing to search, reuse, orchestrate and verify multi-container applications can be even more significant. This is currently not supported by any of the frameworks that we will discuss in Sect. 6.

5.3 Container-oriented design patterns in TOSCA

TOSCA templates describe the structure of (parts of) multi-component applications. This aligns with the idea of design patterns, which also describe the structure of (parts of) software applications [1], and which can be provided as templates to be directly included into TOSCA applications [4].

From an architecture perspective, multi-container Docker applications are often designed according to the microservices architectural style [13] (as Docker perfectly matches the microservices' requirement of independent deployability). Hence, microservices design patterns constitute a concrete example of design patterns that can be provided as predefined TOSCA templates, to support and ease the development of new multi-container Docker applications.

A catalogue of such design patterns is presented in [21]. Three main categories of patterns emerge, i.e., *orchestration and coordination patterns* (capturing communication and coordination from a logical perspective), *deployment patterns* (reflecting physical deployment strategies for services on hosts through Docker containers), and *data management patterns* (capturing data storage options). All patterns falling within such categories align with our discussion on how to enhance the orchestration of multi-contained Docker applications with TOSCA. They indeed provide solutions for orchestrating the management of the components and containers forming multi-container applications, and for managing the Docker volumes storing their data.

We below provide a list of the patterns falling in the above mentioned categories, which can be provided as predefined templates to support the development of multi-container Docker applications in TOSCA.

Orchestration and coordination patterns Within this category, we have design patterns for service composition and discovery.

Service Composition \rightarrow *API Gateway*. An API Gateway is an entry point to a system. It provides a tailored API for each client to route requests

to appropriate containers, aggregate the required contents, and serve them back to the clients. The API Gateway can also implement some shared logic (e.g., authentication), and it can serve as load balancer. Its main goal is to increase system performance and simplify interactions, thus reducing the number of requests per client.

Service Discovery → *Client-Side Discovery*, *Server-Side Discovery*. Multiple instances of the same service usually run in different containers. The communication among them must be dynamically defined and the clients must be able to efficiently communicate to the appropriate microservice that dynamically change instances. For this purpose, service discovery dynamically supports the resolution addresses. For the Client-Side Discovery design pattern, clients query a registry, select an available instance, and make a request directly. For the Server-Side Discovery design pattern, clients make requests via a load balancer, which queries a registry and forwards the request to an available instance. Unlike the API-Gateway pattern, this pattern allows clients and containers to talk to each other directly.

Deployment patterns Within this category, we the most common pattern is the *Multiple Service per Host* design pattern. According to such pattern, each service in an application is deployed in a separate container, containers are distributed among a cluster of hosts, by allowing multiple containers to run on a same host. There is also a *Single Service per Host* design pattern, but it reflects a very uncommon deployment strategy.

Data management patterns This category of patterns focuses on data storage options for applications composed by multiple services and containers.

Database-per-Service. With this design pattern, each service accesses its private database.

Database cluster. The aim of this design pattern is storing data on a database cluster. This improves scalability, allowing to move the databases to dedicated nodes.

Shared database server. The aim of this design pattern is similar to that of the Database Cluster design pattern. However, instead of using a database cluster, all services access a single shared database.

To illustrate the usefulness of the above listed patterns, we refer back to example in Sect. 4. We discussed a layered architecture with GUI, API and database components. In such scenario, the API Gateway Pattern suggests an abstraction that permits routing user requests not only to a single container, but also to different containers (e.g., for different databases). Automatic deployment using Docker compose and Swarm, as also discussed, can also be modelled by exploiting the deployment patterns we introduced.

6 Related work

Despite Docker permits creating multi-container applications (with Docker compose) and ensuring high-availability of containers (with Docker swarm), its current orchestration capabilities are limited. Docker only permits specifying the

images to run, and the interconnections occurring among the containers instantiated from such images. It is not possible to search for images other than by looking for their names and tags in registries, there is no way to validate multi-container applications at design-time, and there is a lack of support for automating the management of the components in multi-container applications.

Multi-container Docker solutions can also be created by organising containers into *clusters*. Each of these clusters consists of host nodes that hold various containers with common services, such as scheduling and load balancing. This can be done with Mesos, Kubernetes, Marathon, and Cloud Foundry's Diego.

Mesos (mesos.apache.org) is an Apache cluster management platform that natively supports LXC and Docker. It organises distributed resources into a single pool, and includes a distributed systems kernel that provides applications with resource management and scheduling. However, Mesos does not allow to structure clusters, nor to orchestrate them.

Kubernetes (kubernetes.io) is a cluster management and orchestration solution at a higher level than Mesos, as it permits structuring clusters of hosts into pods, which are in charge of running containers. However, even though clusters can be structured into pods of containers, Kubernetes still lacks a proper support for orchestrating multi-container applications, by only permitting to specify the images to run through their names and tags. This results in limitations analogous to those we identified for Docker compose and Docker swarm (see Sect. 4): It is not possible to abstractly describe the runtime environment needed to run each component (to be then automatically implemented by adopting reuse techniques), and developers are hence required to manually select the image where to run each component, to extend it, and to package the obtained runtime into a new image. The obtained images can then be composed to build a multi-container application, which however has no explicit information about which component is hosted on which image nor about dependencies occurring among components (as the only dependencies that can be modelled are the interconnections occurring among containers).

Marathon ([mesosphere.github.io/marathon](https://github.com/mesosphere/marathon)) and *Diego* [22] are alternative solutions for managing and orchestrating clusters of containers (on top of Mesos and Cloud Foundry, respectively). Their objectives, as well as their limitations in orchestrating multi-container applications, are similar to those of Kubernetes.

Rocket (coreos.com/rkt) is a container framework alternative to Docker, which tries to address some of the limitations of Docker, like the search and composition of container images. Images of Rocket containers can indeed be composed to form complex applications, and there is a dedicated protocol for retrieving images. However, Rocket still lacks a way of specifying topology and orchestration of multi-container applications.

To summarise, currently existing platforms lack support for the high-level specification of multi-container (Docker) applications, and this limits their capabilities for searching and reusing container images, for orchestrating them to build up multi-container applications, and for verifying designed applications.

The possibility of employing TOSCA for enhancing the orchestration of multi-container applications was suggested in [17]. In this paper, we try to concretise such by providing a discussion of its benefits.

7 Conclusions

The management of multi-component applications across multiple and heterogeneous clouds is becoming commonplace [1,14]. Additionally, with the advent of fog and edge clouds, there is an increasing need for lightweight virtualisation support [7,19]. In this scenario, containers can play an important role, especially to decompose complex applications into suites of lightweight containers running independent services [13,23]. However, currently available platform offer limited support for specifying and orchestrating multi-container applications.

In this paper we have illustrated how a modelling language like TOSCA would enhance the orchestration of multi-container applications, thus overcoming current limitations. While utilising TOSCA for orchestrating container-based applications requires some additional initial effort, it permits composing and automatically orchestrating them to build and manage multi-container applications [10]. TOSCA can also empower the reuse of containers, by allowing developers to search and match them based on what they feature [8].

Additionally, despite both TOSKER [8,10] and Cloudify (getcloudify.org) provide a basic support for deploying multi-container Docker applications specified in a simplified profile of TOSCA, a full-fledged support for orchestrating multi-container applications is still lacking. Its development is in the scope of our future work.

It is finally worth noting that, while Docker is the de-facto standard for container-based virtualisation [18], the same does not hold for TOSCA [3] (which was exploited here just as an example). There exists promising alternatives to TOSCA that can be exploited, In the scope of our future work, we plan to comparatively assess existing topology languages to determine the most suited to our purposes, by also considering how Ansible or similar languages can be extended to develop the aforementioned full-fledged support for orchestrating multi-container applications.

References

1. Andrikopoulos, V.: Engineering cloud-based applications: Towards an application lifecycle. In: Mann, Z.Á., Stolz, V. (eds.) *Advances in Service-Oriented and Cloud Computing*. pp. 57–72. Springer (2018)
2. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software* **33**(3), 42–52 (2016)
3. Bergmayr, A., Breitenbücher, U., Ferry, N., Rossini, A., Solberg, A., Wimmer, M., Kappel, G., Leymann, F.: A systematic review of cloud modeling languages. *ACM Comput. Surv.* **51**(1), 22:1–22:38 (Feb 2018)
4. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: TOSCA: Portable Automated Deployment and Management of Cloud Applications, pp. 527–549. Springer (2014)

5. Brogi, A., Canciani, A., Soldani, J.: Fault-aware management protocols for multi-component applications. *Journal of Systems and Software* **139**, 189 – 210 (2018)
6. Brogi, A., Di Tommaso, A., Soldani, J.: Sommelier: A tool for validating toasca application topologies. In: Pires, L.F., Hammoudi, S., Selic, B. (eds.) *Model-Driven Engineering and Software Development*. pp. 1–22. Springer (2018)
7. Brogi, A., Forti, S., Ibrahim, A.: How to best deploy your fog applications, probably. In: *2017 IEEE Int. Conf. on Fog and Edge Computing (ICFEC)*. pp. 105–114. IEEE (2017)
8. Brogi, A., Neri, D., Rinaldi, L., Soldani, J.: Orchestrating incomplete TOSCA applications with Docker. *Science of Computer Programming* **166**, 194–213 (2018)
9. Brogi, A., Neri, D., Soldani, J.: A microservice-based architecture for (customisable) analyses of docker images. *Software: Practice and Experience* **48**(8), 1461–1474 (2018)
10. Brogi, A., Rinaldi, L., Soldani, J.: TosKer: A synergy between TOSCA and Docker for orchestrating multi-component applications. *Software: Practice and Experience* (2018). <https://doi.org/10.1002/spe.2625>, [In press]
11. Brogi, A., Soldani, J.: Finding available services in TOSCA-compliant clouds. *Science of Computer Programming* **115-116**, 177–198 (2016)
12. Fowler, M.: *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc. (2002)
13. Jamshidi, P., Pahl, C., Mendonca, N., Lewis, J., Tilkov, S.: Microservices: The journey so far and challenges ahead. *IEEE Software* **35**(3), 24–35 (2018)
14. Jamshidi, P., Pahl, C., Mendonca, N.: Pattern-based multi-cloud architecture migration. *Software: Practice and Experience* **47**(9), 1159–1184 (2017)
15. Liu, F., Tong, J., Mao, J., Bohn, R., Messina, J., Badger, L., Leaf, D.: *NIST Cloud Computing Reference Architecture: Recommendations of the National Institute of Standards and Technology (Special Publication 500-292)*. NIST (2012)
16. OASIS: *Topology and Orchestration Specification for Cloud Applications*. <http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf> (2013)
17. Pahl, C.: Containerization and the PaaS cloud. *IEEE Cloud Computing* **2**(3), 24–31 (2015)
18. Pahl, C., Brogi, A., Soldani, J., Jamshidi, P.: Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing* (2017). <https://doi.org/10.1109/TCC.2017.2702586>, [In press]
19. Pahl, C., Lee, B.: Containers and clusters for edge cloud architectures - a technology review. In: *Proc. of FiCloud 2015*, pp. 379–386. IEEE (2015)
20. Soldani, J., Binz, T., Breitenbcher, U., Leymann, F., Brogi, A.: ToscaMart: A method for adapting and reusing cloud applications. *Journal of Systems and Software* **113**, 395 – 406 (2016)
21. Taibi, D., Lenarduzzi, V., Pahl, C.: Architectural patterns for microservices: A systematic mapping study. In: *Proc. of the 8th Int. Conf. on Cloud Computing and Services Science, CLOSER 2018*. pp. 221–232. SciTePress (2018)
22. Winn, D.: *Cloud Foundry: The Cloud-Native Platform*. O’Reilly Media, Inc. (2016)
23. Yangui, S., Mohamed, M., Tata, S., Moalla, S.: Scalable service containers. In: *Proc. of the 2011 IEEE Third Int. Conf. on Cloud Computing Technology and Science (CloudCom 2011)*. pp. 348–356. IEEE Computer Society (2011)
24. Zimmermann, O.: Microservices tenets. *Computer Science - Research and Development* **32**(3), 301–310 (2017)

Transactional Migration of Inhomogeneous Composite Cloud Applications

Josef Spillner and Manuel Ramírez López

Zurich University of Applied Sciences, School of Engineering
Service Prototyping Lab (blog.zhaw.ch/icclab/), 8401 Winterthur, Switzerland
{josef.spillner,ramz}@zhaw.ch

Abstract. For various motives such as routing around scheduled downtimes or escaping price surges, operations engineers of cloud applications are occasionally conducting zero-downtime live migrations. For monolithic virtual machine-based applications, this process has been studied extensively. In contrast, for composite microservice applications new challenges arise due to the need for a transactional migration of all constituent microservice implementations such as platform-specific lightweight containers and volumes. This paper outlines the challenges in the general heterogeneous case and solves them partially for a specialised inhomogeneous case based on the OpenShift and Kubernetes application models. Specifically, the paper describes our contributions in terms of tangible application models, tool designs, and migration evaluation. From the results, we reason about possible solutions for the general heterogeneous case.

1 Introduction

Cloud applications are complex software applications which require a cloud environment to operate and to become programmable and configurable through well-defined and uniform service interfaces. Typically, applications are deployed in the form of virtual machines, containers or runtime-specific archives into environments such as infrastructure or platform offered as a service (IaaS and PaaS, respectively). Recently, container platforms (CaaS) which combine infrastructure and higher-level platform elements such as on-demand volumes and scheduling policies have become popular especially for composite microservice-based applications [1].

The concern of continuous deployment in these environments is then to keep the applications up to date from the latest development activities [2]. Another concern is to maintain flexibility in where the applications are deployed and how quickly and easily they can be re-deployed into another environment. When a new deployment from the development environment is not desired or simply not possible due to the lack of prerequisites, a direct migration from a source to a target environment may be a solution despite hurdles to full automation [3].

Cloud application migration from this viewpoint can be divided into different categories: Homogeneous and heterogeneous migrations, referring to differences

in the source and target environment technologies, same-provider and cross-provider migrations, referring to the ability to migrate beyond the boundaries of a single hosting services provider, as well as offline and online/live migrations, referring to the continuity of application service provisioning while the migration goes on. On the spectrum between homogeneity and heterogeneity, inhomogeneous migrations are concerned with minor automatable differences. This paper is concerned with *live, heterogeneous/inhomogeneous, cross-provider migrations* as shown in Fig. 1.

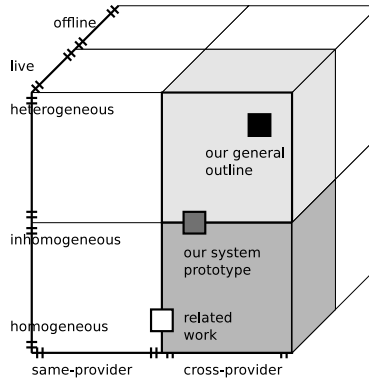


Fig. 1. Positioning within the multi-dimensional categories of cloud application migrations

An additional distinction is the representation of applications. Most of the literature covers monolithic applications which run as instances of virtual machine images where the main concern is pre-copy/post-copy main memory synchronisation [4]. Few emerging approaches exist for more lightweight compositions of stateless containers, where main memory is no longer a concern, and further platform-level components such as database services, volumes, secrets, routes and templates, some of which keep the actual state [5]. This paper is therefore concerned with migrating applications based on *container compositions* between diverse cloud platforms.

Consequently, the main contribution of the paper is a discussion of migration tool designs and prototypes for containerised Docker Compose, OpenShift and Kubernetes applications across providers. OpenShift is one of the most advanced open source PaaS stacks based on Kubernetes, a management and scheduling platform for containers, and in production use at several commercial cloud providers including RedHat’s OpenShift Online, the APPUiO Swiss Container Platform, and numerous on-premise deployments [6]. Additional pure Kubernetes hosting is offered by the Google Cloud Platform, by Azure Container Services and by the overlay platform Tectonic for AWS and Azure, among other providers [7]. Both platforms orchestrate, place, schedule and scale ideally-

stateless Docker containers, while simpler compositions can also be achieved with Docker Compose.

The possibility to have the same containerised application deployed and running in different cloud providers and using different container platforms or orchestration tools is useful for both researcher and for companies. It facilitates the comparison of different cloud providers or different orchestration tools. For companies, it facilitates to run the applications in the most attractive hosting options by cost or other internal constraints. Key questions to which the use of our tools gives answers typically are: Is the migration feasible? Is it lossless? How fast is it? Does the order matter?

The paper is structured as follows. First, we analyse contemporary application compositions to derive requirements for the generalised live heterogeneous migration process (Sect. 2), followed by outlining the tool design principles (Sect. 3) and architecture (Sect. 4) for a simpler subset, inhomogeneous migration. The implemented tools are furthermore described (Sect. 5) and evaluated with real application examples (Sect. 6). The paper concludes with a summary of achievements (Sect. 7) and a discussion on filling the gap to truly heterogeneous live migration.

2 Analysis

In the definition given in a ten-year review of cloud-native applications [8], such applications are designed using self-contained deployment units. In current applications the consensus is to use containers for reasonable isolation and almost native performance. Among the container technologies, Docker containers are the most common technology, although there are alternatives including Rkt, Containerd or CRI-O, as well as research-inspired prototypical engines such as SCONE [9]. In the following, we define a well-designed cloud application as a blueprint-described application, using containers to encapsulate the logic in microservices bound to the data confined in volumes. For deploying these applications in production into the cloud, just the container technology is not enough. Generally, a proper containerised application also uses an advanced container platform or an orchestration solution to add self-healing, auto-scaling, load balancing, service discovery and other properties which make it easier and faster to develop and deploy applications in the cloud. The platform also leverages more resilience, higher availability and scalability in the application itself. Among the most popular tools and platforms used to orchestrate containers are Docker Swarm, Docker Compose, Kubernetes, OpenShift, Rancher, and similar platforms. All of these can run in different cloud providers or on-premise. Moreover, usually each cloud provider has their own container platform. In Fig. 2 a diagram about the main container platforms and container orchestrators with their different associated composition blueprints is shown. The diagram also reveals relations and classifies the approaches by licencing (open source or proprietary) and by fitness for production. This complex technological landscape leads to different blueprints for the same containerised application depending which causes

practical difficulties for migrations. Despite fast ongoing consolidation, including the announced discontinuation of Docker Cloud in 2018, minor variations such as installed Kubernetes extensions continue to be a hurdle for seamless migration.

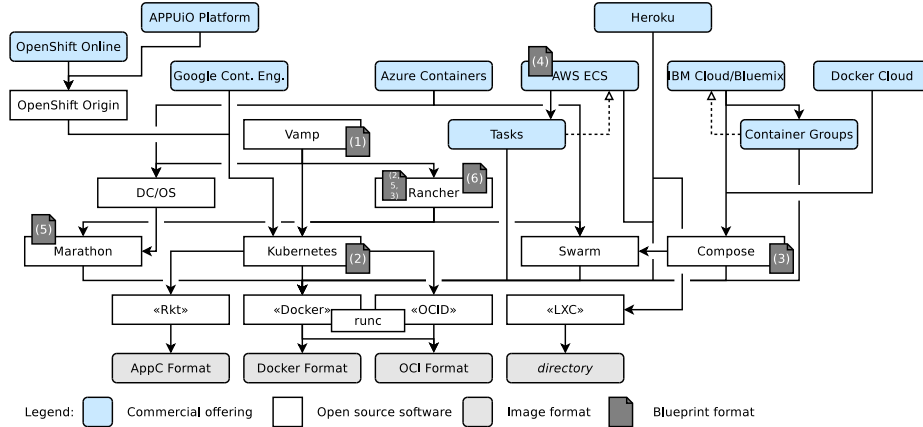


Fig. 2. Map of major container platforms and orchestration tools

The planning of the migration of a containerised application thus encompasses two key points which restrain the ability to automate the process:

- The blueprints: Even though containers encapsulate all the code in images which are meant to be portable and run everywhere, most of the real applications will need an orchestration tool to exploit all advantages that the cloud environment introduces: service discovery, definition of the number of replicas or persistence configuration. As most orchestration tools will introduce specific blueprints or deployment descriptors, the migration tool will need to convert between blueprint formats through transformation, perform minor modifications such as additions and removals of expressions, or rewrite limits and group associations (requirement R1).
- The data: Migration of the persisted data and other state information is non-trivial. In most container engines, the persistence of the data is confined to volumes. Depending of the cloud provider, the blueprints processed by the orchestration tools could reference volumes differently even for homogeneous orchestration tools, leading to slight differences and thus inhomogeneity (requirement R2).

To address these two points and increase the automation, the design of a suitable migration tool needs to account specifically for blueprint conversion and properly inlined data migration. We formalise a simplified composite application deployment as $D = \{b, c, v, \dots\}$, respectively, where: b : blueprint; c : set of associated containers; v : set of associated volumes. For example, a simplified OpenShift application is represented as $D_{openshift} = \{b, c, v, t, is, r, \dots\}$, where:

t : set of templates; is : set of image streams; r : set of routes. The goal of ideal heterogeneous migration m is to find migration paths from any arbitrary source deployment to any target deployment: $m = D \rightarrow D'$.

Fig. 3 summarises the different realistically resulting inhomogeneous migration paths between the three possible configurations $D_{kubernetes}$, $D_{openshift}$ and $D_{compose}$. Through various modifications applied to the orchestration descriptors, sources and targets can be largely different while mostly avoiding a loss of deployment information in fulfilment of R1.

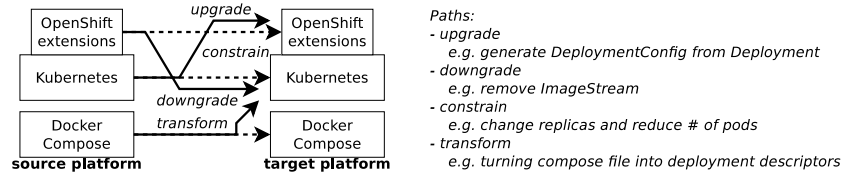


Fig. 3. Inhomogeneous application migration paths between three systems

3 General Application Migration Workflows

Requirement R1 calls for a dedicated blueprint extraction, conversion and re-deployment process. We consider four steps in this process (see Fig. 4) which shall be implemented by a migration tool:

- Step 1. Downloading the blueprints of the composite application: The tool will connect to the source platform the application is running on, will identify all the components of the application and download the blueprints to a temporary location.
- Step 2. Converting the blueprints: A conversion from source to target format takes place. Even when homogeneous technologies are in place on both sides, re-sizing and re-grouping of components can be enforced according to the constraints on the target side (fulfilling R1).
- Step 3. Deploying the application: The tool will connect to the new orchestration platform and deploy the application there.
- Step 4. Deleting the application: Once the new application instance is running in the new place, the tool can delete the old application instance from the previous place. This step is optional and only executed under move semantics as opposed to copy semantics.

A major issue is the transactional guarantee of achieving a complex running and serving application on the target platform which in all regards equals the source. To make this process successful in all cases, the tool algorithm must further fulfil the following three requirements:

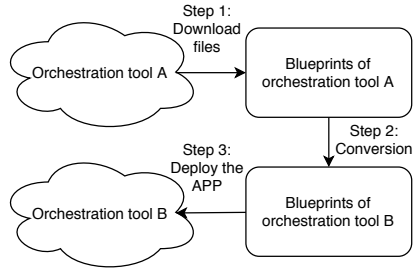


Fig. 4. Blueprints process diagram

- Connect to each of the different platforms in scope for heterogeneous migration.
- Convert between all the blueprints.
- Download and upload the application components from/to all the platforms, ensuring a re-deployment in the right order and a smooth hand-over by name service records which are typically external to both source and target platform.

With the previously described workflow, the tool can migrate stateless application or the stateless components of a stateful application. To complete the migration, the data in the containers needs to be migrated as well according to R2. In practice, this refers to volumes attached to containers, but also to databases and message queues which must be persisted in volume format beforehand. The process of the migration of a volume will be as follows:

- Step 1. Find the list of volumes linked to an application and for each one the path to the data.
- Step 2. Download the data to a temporary location. Due to the size, differential file transfer will be used.
- Step 3. Identify the same volume in the new deployment and pre-allocate the required storage space.
- Step 4. Upload the data to the new volume.

Now, we devise a fictive tool to express how the combined fulfilment of R1 and R2 in the context of heterogeneous application migration can be realised, expressed by Fig. 5 which highlights the separation into blueprints and data.

Although practitioners and researchers would benefit greatly from such a generic and all-encompassing tool, its conception and engineering would take many person months of software development work, needlessly delaying a prototype to answer the previously identified questions many companies in the field have right now. Instead, to focus on key research questions as outlined in the introduction follows a divide-and-conquer strategy. We subdivide the overall fictive tool into a set of smaller tools logically grouped into three categories, as shown in Fig. 6. Thus, we put our own prototypical work into context of a wider

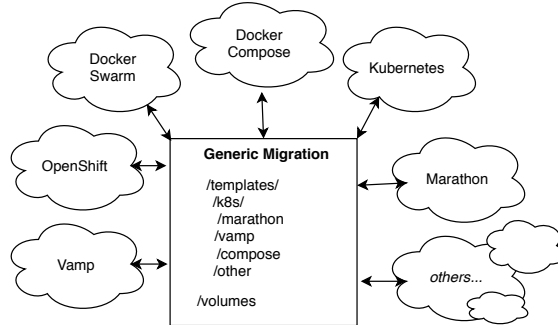


Fig. 5. Stateful application components diagram

ecosystem with some existing tools and further ongoing and future developments, making it possible to evaluate migration scenarios already now. The tools are:

- Homogeneously migrating containerised applications between multiple instances of the same orchestration tool: `os2os` (our work).
- Converting blueprints between the formats required by the platforms related to R1: `Kompose` (existing work).
- Rewriting Kubernetes blueprints to accomodate quotas: `descriptorrewriter` (our work).
- Migrating volumes related to R2: `volume2volume` (our work).
- Homogeneous transactional integration of volume and data migration for OpenShift as a service: `openshifter` (our early stage work).

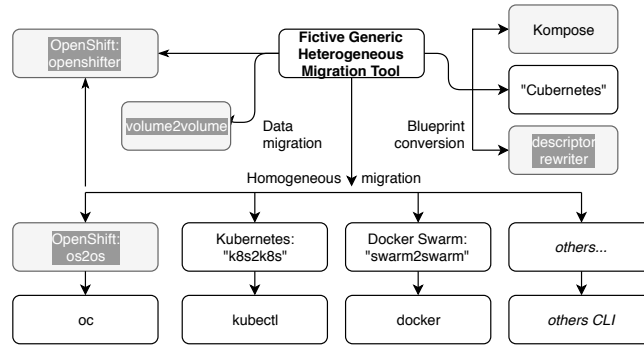


Fig. 6. Implementation strategy for fictive heterogeneous migration tool

We contribute in this paper the architectural design, implementation and combined evaluation of four tools referring to inhomogeneous OpenShift/Kubernetes/Docker Compose-to-OpenShift/Kubernetes migration. Use cases encompass intra-region replication and region switching within one provider, migration

from one provider to another, and developer-centric migration of local test applications into a cloud environment. All tools are publicly available for download and experimentation¹.

4 Migration Tools Design and Architecture

The general design of all tool ensures user-friendly abstraction over existing low-level tools such as `oc` and `kubect1`, the command line interfaces to OpenShift and Kubernetes, as well as auxiliary tools such as `rsync` for differential data transfer. Common migration and copy/replication workflows are available as powerful single commands. In Openshifter, these are complemented with full transaction support so that partial migrations can be gracefully interrupted or rolled back in case of occurring issues.

As Fig. 7 shows on the left side, `os2os` uses `oc` to communicate with the source and target OpenShift clusters and temporarily stores all artefacts in local templates and volumes folders. This choice ensures that only a single provider configuration file needs to be maintained and that any features added to `oc` will be transparently available. On the right side of Fig. 7, the Openshifter tool is depicted which follows a service-oriented design. This choice ensures that the migration code itself runs as stateless, resilient and auto-scaled service. A further difference between the tools is that for Openshifter, we have explored a conceptual extension of packaged template and configuration data archives, called Helm charts, into *fat charts* which include a snapshot of the data, closing the gap to monolithic virtual machines.

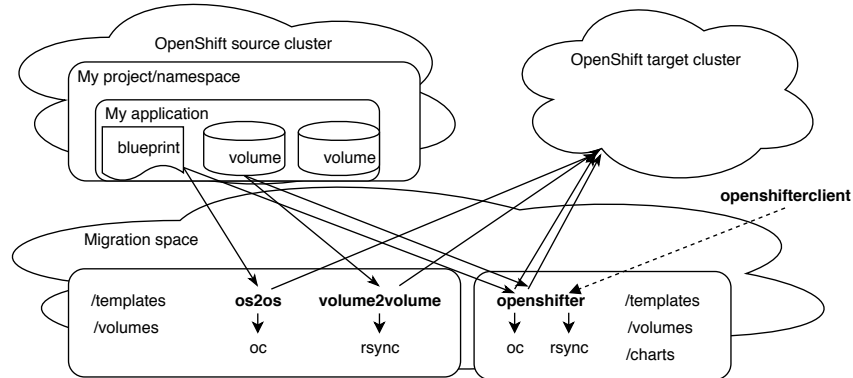


Fig. 7. OS2OS/Volume2Volume architectures (left); Openshifter architecture (right)

Exemplarily for all tools, `os2os` is composed of the following commands:

¹ Tools website: <https://github.com/serviceprototypinglab/>

- export: Connect to one cluster and export all the components (objects) of one application in one project, saved locally in a folder called `templates`.
- up: Connect to one cluster and upload all the components of one application in one project which are saved in `templates`.
- down: Connect to one cluster and delete all the components of one application in one project.
- migrate: Combine all the commands chronologically for a full migration in a single workflow.

The tools are implemented in different ways following the different designs. Both `os2os` and `volume2volume` are inspired by Kompose. They are implemented as command-line tools using Go with Cobra as library for handling the command-line parameters. Furthermore, the command names are derived from Kompose, making it easy to learn the tool for existing Kompose users. As usual in applications using Cobra, the configuration of the tool is stored in a YAML file. It contains the credentials to connect to the clusters, the cluster endpoints, the projects and the object types to migrate, overriding the default value of all object types. The `openshifter` prototype is implemented in Python using the AIO-HTTP web library to expose RESTful methods and works without any configuration file by receiving all parameters at invocation time.

5 Evaluation

When evaluating cloud migration tools, three important questions arise on whether the migration is lossless, performing and developer-acceptable. The measurable evaluation criteria are:

- C1 / Losslessness: The migration needs to avoid loss of critical application deployment information even after several roundtrips of migration between inhomogeneous systems. This is a challenge especially in the absence of features on some platforms. For instance, Kubernetes offers auto-scaling while Docker Swarm does not, leading to the question of how to preserve the information in case a migration from Kubernetes to Docker Swarm is followed by a reverse migration while the original source platform has vanished.
- C2 / Performance: A quantitative metric to express which time is needed both overall and for the individual migration steps. Further, can this time be pre-calculated or predicted in order to generate automated downtime messages, or can any downtime be alleviated.
- C3 / Acceptance: The migration needs to be easy to use for developers and operators as well as in modern DevOps environments.

A testbed with two local virtual machines running OpenShift 3.6 (setup S1) as well as a hosted OpenShift environment provided by the Swiss container platform APPUiO (S2) were set up to evaluate our tools experimentally according to the defined criteria C1 and C2. A synthetic scenario application consisting of three deployments and three services was prepared for that matter (A1), and the existing Snafu application (A2) was used for the comparison. The evaluation of C3 is left for future work.

5.1 Evaluation of Losslessness

For Kubernetes and OpenShift, the scenario service consists of shared Service and ConfigMap objects as well as platform-specific ones which are subject to loss; for Docker Compose, it consists of roughly equivalent directives. The deployed service was migrated from source to target and, with swapped roles between the platforms, back again from target to source. The following table reports on the loss of information depending on the system type. The Kompose tool incorrectly omits the lowercasing of object names and furthermore does not automatically complete the generated descriptors with information not already present in the Docker Compose files. To address the first issue, we have contributed a patch, whereas the second one would require a more extensive tool modification. The upgrade from Kubernetes to OpenShift works although OpenShift merely supports Deployment objects as a convenience whereas DeploymentConfig objects would be needed.

Table 1: Losslessness of blueprint transformations

Source	Target	Loss
OpenShift	OpenShift	none (assuming equal quotas)
OpenShift	Kubernetes (manual)	ImageStream,Route, DeploymentConfig
Kubernetes	OpenShift (manual)	(Deployment)
Docker Compose	Kubernetes (w/ Kompose)	none (yet incomplete & incorrect)

As a result, we have been able to automate all migrations except for the downgrade from OpenShift to Kubernetes using a combination of our tools which is invoked transparently when using Openshifter. The losslessness further refers to in-flight import and export of volume data. To avoid data corruption, applications need to perform modifications on the file level atomically, for instance by placing uploads into temporary files which are subsequently atomically renamed. Support for applications not adhering to this requirement is outside the scope of our work.

5.2 Evaluation of Performance

The synthetic scenario service A1 was exported from the source, re-deployed at the target, and torn down at the source 10 times with `os2os` in order to get information about the performance and its deviation in the local-to-local migration setup S1. Fig. 8 shows the results of the performance experiments. An evident characteristic is that exporting objects without changing them is more stable than running the down/up commands which modify the objects and cause changes to the scheduling of the remaining objects. A second observation is that, counter-intuitively, the `down` command consumes most of the time. A plausible explanation is that instead of simple deletions, objects are rather scheduled for deletion into a queue.

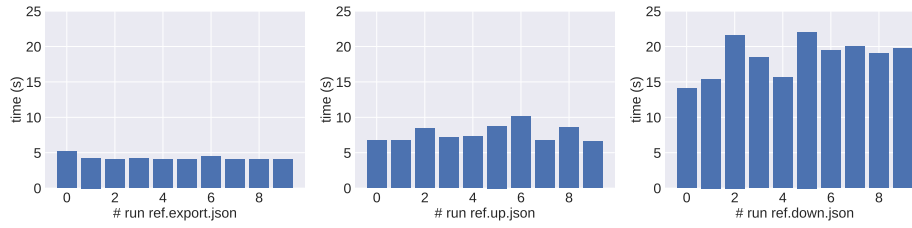


Fig. 8. Durations of the individual migration phases – export (left figure), up (middle), down (right) – between two local Kubernetes clusters

Service A2 was transformed automatically to measure the influence of the transformation logic on planned live migrations. The creation of Kubernetes descriptors with Kompose takes approximately 0.028 s. The adjustment of quotas and consolidation of pods, as performed by `descriptorrewriter`, takes approximately 0.064 s on the resulting Kubernetes descriptors. Both transformations are thus negligible which implies that apart from blueprint exports, the data transfer, which is primarily limited by the cluster connectivity, is the dominant influence on overall performance.

6 Conclusion

We have conducted a first analytical study on migrating cloud-native applications between inhomogeneous development and production platforms. The analysis was made possible through prototypical migration tools whose further development is in turn made possible by the results of the experiments. The derived findings from the experimental evaluation suggest that application portability is still an issue beyond the implementation (container) images. Future cloud platforms should include portability into the design requirements.

7 Future work

The current prototypes only support Kubernetes-based platforms. All functionality to convert other formats has been integrated into the experiments with external and existing tools. In the future, we want to integrate them in a unified way into `openshifter`. Further, we want to work on stricter requirements concerning a production-ready migration. They encompass improved user interfaces for easier inter-region/-zone migration within one provider, automatic identification of associated state and data formats, plugins for databases and message queues which keep non-volume state, data checksumming, and pre-copy statistics about both expected timing and resource requirements of the process and the subsequent deployment.

Acknowledgements

This research has been funded by Innosuisse - Swiss Innovation Agency in project MOSAIC/19333.1.

References

1. S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya. Efficient Virtual Machine Sizing for Hosting Containers as a Service (SERVICES 2015). In *2015 IEEE World Congress on Services*, pages 31–38, June 2015.
2. Pilar Rodríguez, Alireza Haghighatkah, Lucy Ellen Lwakatare, Susanna Teppola, Tanja Suomalainen, Juho Eskeli, Teemu Karvonen, Pasi Kuvaja, June M. Verner, and Markku Oivo. Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, 123:263–291, 2017.
3. Massimo Ficco, Christian Esposito, Henry Chang, and Kim-Kwang Raymond Choo. Live Migration in Emerging Cloud Paradigms. *IEEE Cloud Computing*, 3(2):12–19, 2016.
4. Petronio Bezerra, Gustavo Martins, Reinaldo Gomes, Fellype Cavalcante, and Anderson F. B. F. da Costa. Evaluating live virtual machine migration overhead on client’s application perspective. In *2017 International Conference on Information Networking, ICOIN 2017, Da Nang, Vietnam, January 11-13, 2017*, pages 503–508, 2017.
5. Jaemyoun Lee and Kyungtae Kang. Poster: A Lightweight Live Migration Platform with Container-based Virtualization for System Resilience. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys’17, Niagara Falls, NY, USA, June 19-23, 2017*, page 158, 2017.
6. C. Pahl. Containerization and the PaaS Cloud. *IEEE Cloud Computing*, 2(3):24–31, May 2015.
7. Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Commun. ACM*, 59(5):50–57, 2016.
8. Nane Kratzke and Peter-Christian Quint. Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. *Journal of Systems and Software*, 126:1–16, 2017.
9. Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O’Keeffe, Mark Stillwell, David Goltzsche, David M. Ebers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 689–703, 2016.

Secure Apps in the Fog: Anything to Declare?

Antonio Brogi, Gian-Luigi Ferrari, and Stefano Forti

Department of Computer Science,
University of Pisa, Italy
`name.surname@di.unipi.it`

Abstract. Assessing security of application deployments in the Fog is a non-trivial task, having to deal with highly heterogeneous infrastructures containing many resource-constrained devices. In this paper, we introduce: *(i)* a declarative way of specifying security capabilities of Fog infrastructures and security requirements of Fog applications, and *(ii)* a (probabilistic) reasoning strategy to determine application deployments and to quantitatively assess their security level, considering the trust degree of application operators in different Cloud/Fog providers. A lifelike example is used to showcase a first proof-of-concept implementation and to illustrate how it can be used in synergy with other predictive tools to optimise the deployment of Fog applications.

Keywords: Fog computing · Application Deployment · Security Assessment · Executable Specifications · Probabilistic Logic Programming · Trust.

1 Introduction

Fog computing [9] aims at better supporting the growing processing demand of (time-sensitive and bandwidth hungry) Internet of Things (IoT) applications by selectively pushing computation closer to where data is produced and exploiting a geographically distributed multitude of heterogeneous devices (e.g., personal devices, gateways, micro-data centres, embedded servers) spanning the continuum from the Cloud to the IoT. As a complement and an extension of the Cloud, the Fog will naturally share with it many security threats and it will also add its peculiar ones. On the one hand, Fog computing will increase the number of security enforcement points by allowing local processing of private data closer to the IoT sources. On the other hand, the Fog will be exposed to brand new threats for what concerns the trust and the physical vulnerability of devices. In particular, Fog deployments will span various service providers - some of which may be not fully trustable - and will include accessible devices that can be easily hacked, stolen or broken by malicious users [25]. Security will, therefore, play a crucial role in the success of the Fog paradigm and it represents a concern that should be addressed *by-design* at all architectural levels [26, 37]. The Fog calls for novel technologies, methodologies and models to guarantee adequate security (privacy and trust) levels to Fog deployments even when relying upon resource-constrained devices [8].

Meanwhile, modern computing systems are more and more made from distributed components – such as in service-oriented and micro-service based architectures – what makes it challenging to determine how they can be *best-placed* so to fulfil various application requirements. In our previous work, we proposed a model and algorithms to determine eligible deployments of IoT applications to Fog infrastructures [4] based on hardware, software and QoS requirements. Our prototype – `FogTorchΠ` – implements those algorithms and permits to estimate the QoS-assurance, the resource consumption in the Fog layer [5] and the monthly deployment cost [6] of the output eligible deployments. Various other works tackled the problem of determining “optimal” placements of application components in Fog scenarios, however, none included a quantitative security assessment to holistically predict security guarantees of the deployed applications, whilst determining eligible application deployments. Therefore, there is a clear need to evaluate *a priori* whether an application will have its security requirements fulfilled by the (Cloud and Fog) nodes chosen for the deployment of its components. Furthermore, due to the mission-critical nature of many Fog applications (e.g., e-health, disaster recovery), it is important that the techniques employed to reason on security properties of deployed multi-component applications are configurable and well-founded.

In this paper, we propose a methodology (`SecFog`) to (quantitatively) assess the security level of multi-component application deployments in Fog scenarios. Such quantitative assessment can be used both alone – to maximise the security level of application deployments – and synergically with other techniques so to perform multi-criteria optimisations and to determine the *best* placement of application components in Fog infrastructure. This work allows application deployers to specify security constraints both at the level of the components and at the level of the application as a whole. As per recent proposals in the field of AI [3], it exploits probabilistic reasoning to account for reliability and trust, whilst capturing the uncertainty typical of in Fog scenarios. Therefore, we propose: *(i)* a declarative methodology that enables writing an executable specification of the security policies related to an application deployment to be checked against the security offerings of a Fog infrastructure, *(ii)* a reasoning methodology that can be used to look for secure application deployments and to assess the security levels guaranteed by any input deployment, and *(iii)* a first proof-of-concept implementation of `SecFog` which can be used to optimise security aspects of Fog application deployments along with other metrics.

The rest of this paper is organised as follows. After reviewing some related work (Section 2), we offer an overview of `SecFog` and we introduce a motivating example (Section 3). Then, we present our proof-of-concept implementation of `SecFog` and we show how it can be used to determine application deployment whilst maximising their security level (Section 4). Finally, we show how `SecFog` can be used with `FogTorchΠ` to identify suitable trade-offs among QoS-assurance, resource usage, monthly cost and security level of eligible deployments (Section 5), and we briefly conclude with some directions for future work (Section 6).

2 Related Work

Among the works that studied the placement of multi-component applications to Cloud nodes, very few approaches considered security aspects when determining eligible application deployments, mainly focussing on improving performance, resource usage and deployment cost [18, 21], or on performing identification of potential data integrity violations based on pre-defined risk patterns [28]. Indeed, existing research considered security mainly when treating the deployment of business processes to (federated) multi-Clouds (e.g., [23, 12, 36]). Similar to our work, Luna et al. [19] were among the first to propose a quantitative reasoning methodology to rank single Cloud providers based on their security SLAs, and with respect to a specific set of (user-weighted) security requirements. Recently, swarm intelligence techniques [21] have been exploited to determine eligible deployments of composite Cloud applications, considering a risk assessment score based on node vulnerabilities.

Fog computing introduces new challenges, mainly due to its pervasive geo-distribution and heterogeneity, need for QoS-awareness, dynamicity and support to interactions with the IoT, that were not thoroughly studied in previous works addressing the problem of application deployment to the Cloud [32, 35]. Among the first proposals investigating these new lines, [15] proposed a Fog-to-Cloud search algorithm as a first way to determine an eligible deployment of (multi-component) DAG applications to tree-like Fog infrastructures. Their placement algorithm attempts the placement of components *Fog-to-Cloud* by considering hardware capacity only. An open-source simulator – iFogSim – has been released to test the proposed policy against Cloud-only deployments. Building on top of iFogSim, [20] refines tries to guarantee the application service delivery deadlines and to optimise Fog resource exploitation. Also [33] used iFogSim to implement an algorithm for optimal online placement of application components, with respect to load balancing. Recently, exploiting iFogSim, [13] proposed a distributed search strategy to find the best service placement in the Fog, which minimises the distance between the clients and the most requested services, based on request rates and available free resources. [17, 30] proposed (linearithmic) heuristic algorithms that attempt deployments prioritising placement of applications to devices that feature with less free resources.

From an alternative viewpoint, [16] gave a Mixed-Integer Non-Linear Programming (MINLP) formulation of the problem of placing application components so to satisfy end-to-end delay constraints. The problem is then solved by linearisation into a Mixed-Integer Linear Programming (MILP), showing potential improvements in latency, energy consumption and costs for routing and storage that the Fog might bring. Also [29] adopted an ILP formulation of the problem of allocating computation to Fog nodes so to optimise time deadlines on application execution. A simple linear model for the Cloud costs is also taken into account. Finally, dynamic programming (e.g., [27]), genetic algorithms (e.g., [29]) and deep learning (e.g., [31]) were exploited promisingly in some recent works.

Overall, to the best of our knowledge, no previous work included a quantitative assessment of the security level of candidate Fog application deployments.

3 Methodology Overview

The OpenFog Consortium [1] highlighted the need for Fog computing platforms to guarantee privacy, anonymity, integrity, trust, attestation, verification and measurement. Whilst security control frameworks exist for Cloud computing scenarios (e.g., the EU Cloud SLA Standardisation Guidelines [2] or the ISO/IEC 19086), to the best of our knowledge, no standard exists yet that defines security objectives for Fog application deployments. Based on recent surveys about security aspects in Fog computing (i.e., [21], [22], [25]), we devised a simple example of taxonomy¹ (Figure 1) of security features that can be offered by Cloud and Fog nodes and therefore used for reasoning on the security levels of given Fog application deployments.

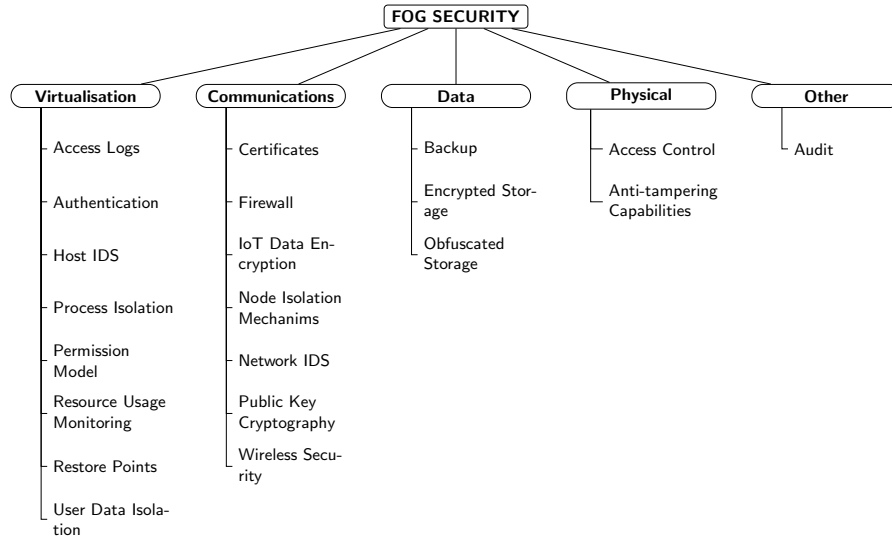


Fig. 1. An example of taxonomy of security capabilities in Fog computing.

Security features that are common with the Cloud might assume renewed importance in Fog scenarios, due to the limited capabilities of the available devices. For instance, guaranteeing physical integrity of and user data isolation at an access point with Fog capabilities might be very difficult. Apropos, the possibility to encrypt or obfuscate data at Fog nodes, along with encrypted IoT communication and physical anti-tampering machinery, will be key to protect those application deployments that need data privacy assurance.

Figure 2 shows the ingredients needed to perform the security assessment by means of the SecFog methodology. On the one hand, we assume that infrastruc-

¹ The proposed taxonomy can be easily modified, extended and refined so as to include new security categories and third-level security features as soon as normative security frameworks will get established for the Fog.

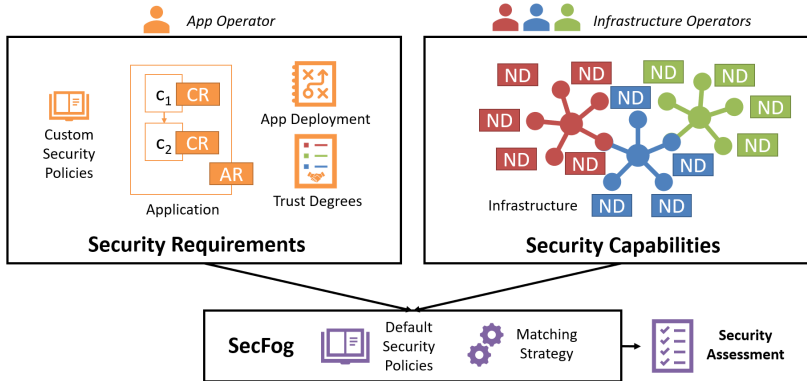


Fig. 2. Bird's-eye view of SecFog.

ture operators declare the *security capabilities* featured by their nodes². Namely, for each node she is managing, the operator publishes a **Node Descriptor (ND)** featuring a list of the node security capabilities along with a declared measure of their reliability (in the range $[0, 1]$), as shown in Figure 4. On the other hand, based on the same common vocabulary, application operators can define (non-trivial) *custom security policies*. Such properties can complete or override a set of *default security policies* available in SecFog implementation. Custom security policies can be either existing ones, inferred from the presence of certain node capabilities, or they can be autonomously specified/enriched by the application deployers, depending on business-related considerations.

For instance, one can derive that application components deployed to nodes featuring **Public Key Cryptography** capabilities can communicate through **End-to-End Secure** channel. A different stakeholder might also require the availability of **Certificates** at both end-point to consider a channel **End-to-End Secure**. Similarly, one can decide to infer that a node offering **Backup** capabilities together with **Encrypted Storage** or **Obfuscated Storage** can be considered a **Secure Storage** provider. Custom and default properties are used, along with ground facts, to specify the *security requirements* of a given application as **Component Requirements (CR)** and **Application Requirements (AR)**, or both. For instance, application operators can specify that a certain component c is securely deployed to node n when n features **Secure Storage** and when the communication with component c' happens over an **End-to-End Secure** channel.

Finally, the security level of an *application deployment* can be assessed by matching the security requirements of the application with the security capabilities featured by the infrastructure and by multiplying the reliability of all exploited security capabilities, weighting them as per *trust degrees*, which may

² For the sake of simplicity, in this paper, we assume that operators exploit the vocabulary of the example taxonomy in Figure 1. In reality, different operators can employ different vocabulary and then rely on mediation mechanisms.

be assigned by application deployers to each infrastructure operator. This last step can be used both to assess the security level of a single (possibly partial) input application deployment and to generate and test all eligible deployments according to the declared security requirements. We now go through a motivating example that we will retake later on by exploiting the SecFog prototype.

3.1 Motivating Example

We retake the application example of [6]. Consider a simple Fog application (Figure 3) that manages fire alarm, heating and A/C systems, interior lighting, and security cameras of a smart building. The application consists of three microservices:

- IoTController, interacting with the connected cyber-physical systems,
- DataStorage, storing all sensed information for future use and employing machine learning techniques to update sense-act rules at the IoTController so to optimise heating and lighting management based on previous experience and/or on people behaviour, and
- Dashboard, aggregating and visualising collected data and videos, as well as allowing users to interact with the system.

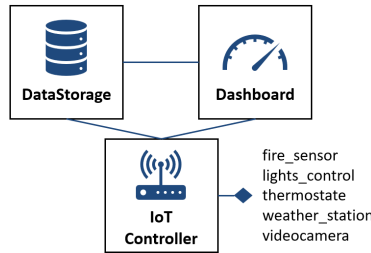


Fig. 3. Fog application.

Each microservice represents an independently deployable component of the application [24] and has hardware and software requirements³ in order to function properly. Application components must cooperate so that well-defined levels of service are met at runtime. Hence, communication links supporting component-component and component-thing interactions should provide suitable end-to-end latency and bandwidth.

Figure 4 shows the infrastructure – two Cloud data centres, three Fog nodes – to which the smart building application is deployed. For each node, the available security capabilities and their reliability (as declared by the infrastructure operator) are listed in terms of the taxonomy of Figure 1.

³ For the sake of readability, we omit the application requirements. The interested reader can find all the details in [6].

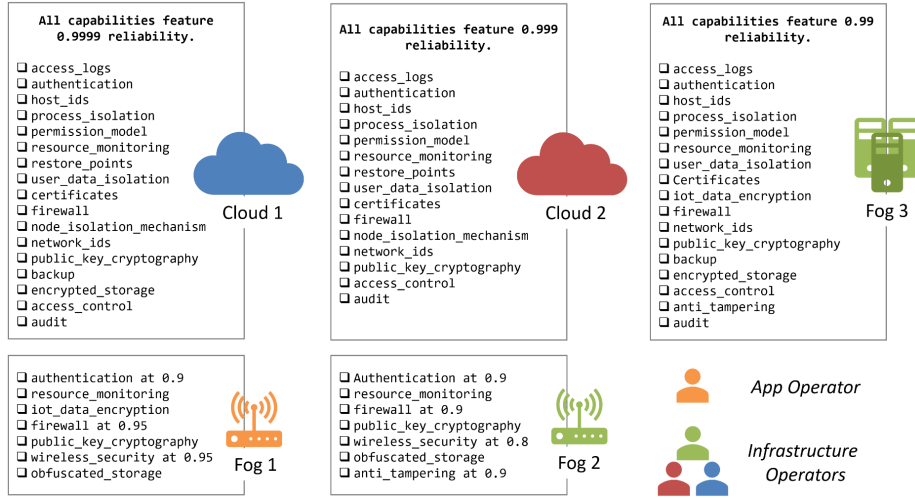


Fig. 4. Fog infrastructure: security view.

Table 1 lists all the deployments of the given application to the considered infrastructure which meet all set software, hardware and network QoS requirements, as they are found by FogTorchΠ in [6]. For each deployment, FogTorchΠ outputs the QoS-assurance (i.e., the likelihood it will meet network QoS requirements), an aggregate measure of Fog resource consumption, and an estimate of the monthly cost for keeping the deployment up and running. Deployments annotated with * are only available when Fog 2 features a 4G connection which costs, however, 20 € a month in addition to the costs reported in Table 1.

In [6], the deployments Δ2 and Δ16 are selected as the best candidates depending on the type of mobile connection (i.e., 3G vs 4G) available at Fog 2. As the majority of the existing approaches for application placement, [6] focuses on finding deployments that guarantee application functionality and end-user preferences, currently ignoring security aspects in the featured analysis.

Nevertheless, the application operators are able to define the following Component Requirements:

- IoTController requires Physical Security guarantees (i.e., Access Control ∨ Anti-tampering Capabilities) so to avoid that temporarily stored data can be physically stolen from the deployment node,
- DataStorage requires Secure Storage (viz., Backup ∧ (Obfuscated Storage ∨ Encrypted Storage)), the availability of Access Logs, a Network IDS in place to prevent distributed Denial of Service (dDoS) attacks, and
- Dashboard requires a Host IDS installed at the deployment node (e.g., an antivirus software) along with a Resource Usage Monitoring to prevent interactions with malicious software and to detect anomalous component behaviour.

Table 1. Eligible deployments of the example application.

Dep. ID	IoTController	DataStorage	Dashboard	QoS	Resources	Cost
$\Delta 1$	Fog 2	Fog 3	Cloud 2	98.6%	48.4%	€856.7
$\Delta 2$	Fog 2	Fog 3	Cloud 1	98.6%	48.4%	€798.7
$\Delta 3$	Fog 3	Fog 3	Cloud 1	100%	48.4%	€829.7
$\Delta 4$	Fog 2	Fog 3	Fog 1	100%	59.2%	€844.7
$\Delta 5$	Fog 1	Fog 3	Cloud 1	96%	48.4%	€837.7
$\Delta 6$	Fog 3	Fog 3	Cloud 2	100%	48.4%	€887.7
$\Delta 7$	Fog 3	Fog 3	Fog 2	100%	59.2%	€801.7
$\Delta 8$	Fog 3	Fog 3	Fog 1	100%	59.2%	€875.7
$\Delta 9$	Fog 1	Fog 3	Cloud 2	96%	48.4%	€895.7
$\Delta 10$	Fog 1	Fog 3	Fog 2	100%	59.2%	€809.7
$\Delta 11$	Fog 1	Fog 3	Fog 1	100%	59.2%	€883.7
$\Delta 12^*$	Fog 2	Cloud 2	Fog 1	94.7%	16.1%	€870.7
$\Delta 13^*$	Fog 2	Cloud 2	Cloud 1	97.2%	5.4%	€824.7
$\Delta 14^*$	Fog 2	Cloud 2	Cloud 2	98.6%	5.4%	€882.7
$\Delta 15^*$	Fog 2	Cloud 1	Cloud 2	97.2%	5.4%	€785.7
$\Delta 16^*$	Fog 2	Cloud 1	Cloud 1	98.6%	5.4%	€727.7
$\Delta 17^*$	Fog 2	Cloud 1	Fog 1	94.7%	16.1%	€773.7

Furthermore, the Application Requirements require guaranteed end-to-end encryption among all components (viz., all deployment nodes should feature Public Key Cryptography) and that deployment nodes should feature an Authentication mechanism. Finally, application operators assign a trust degree of 80% to the infrastructure providers of Cloud 1 and Cloud 2, and of 90% to the infrastructure providers of Fog 3 and Fog 2. Naturally, they consider their management of Fog 1 completely trustable.

4 Proof-of-Concept

Being SecFog a declarative methodology based on probabilistic reasoning about declared infrastructure capabilities and security requirements, it was natural to prototype it relying on probabilistic logic programming. To implement both the model and the matching strategy we used a language called *ProbLog* [10]. ProbLog is a Python package that permits writing logic programs that encode complex interactions between large sets of heterogeneous components, capturing the inherent uncertainties that are present in real-life situations. Problog programs are composed of *facts* and *rules*. The facts, such as

$$p::f.$$

represent a statement f which is true with probability p ⁴. The rules, like

$$r :- c1, \dots, cn.$$

⁴ A fact declared simply as $f.$ is assumed to be true with probability 1.

represent a property r inferred when $c_1 \wedge \dots \wedge c_n$ hold⁵. ProbLog programs are logic programs in which some of the facts are annotated with (their) probabilities. Each program defines a probability distribution over logic programs where a fact $p:f.$ is considered true with probability p and false with probability $1 - p$. The ProbLog engine [11] determines the success probability of a query q as the probability that q has a proof, given the distribution over logic programs.

Our prototype offers three main default security policies that can be used to compose more complex application security requirements. First

```
secure(C, N, D) :-
    member(d(C,N), D),
    node(N, Op),
    trustable(Op).
```

that checks if a component C is actually deployed to an existing node N (as per deployment D) and that the infrastructure operator Op managing N is trustable according to the application operator. Then

```
secureApp(A,D) :-
    app(A,L),
    deployment(L,D),
    secureComponents(A,L,D).

secureComponents(A, [], _).
secureComponents(A, [C|Cs], D) :-
    secureComponent(C,N,D),
    secureComponents(A,Cs,D).
```

that checks whether, according to an input deployment D , each component of a given application A can be securely deployed, i.e. if $\text{secureComponent}(C, N, D)$ holds for all components C of A . The application operator is therefore asked to define a $\text{secureComponent}(C, N, D)$ for each of the application components, always including the default predicate $\text{secure}(C, N, D)$.

4.1 Motivating Example Continued

In this section, we retake the example of Section 3.1 and we show how ProbLog permits to naturally express both security capabilities of an infrastructure and security requirements of an application.

Node Descriptors can be expressed by listing ground facts, possibly featuring a probability that represents their reliability according to the infrastructure provider. For instance, `fog1` directly operated by the application operator `appOp` is described as

```
node(fog1,appOp).
0.9::authentication(fog1).
```

⁵ Both r and $\{ci\}$ can have variable (upper-case) or constant (lower-case) input parameters.


```

resource_monitoring(fog1).
iot_data_encryption(fog1).
0.95::firewall(fog1).
public_key_cryptography(fog1).
0.95::wireless_security(fog1).
obfuscated_storage(fog1).

```

All the Node Descriptors made following this template form a description of the *security capabilities* available in the infrastructure.

Application operators can define the topology of an application by specifying an identifier and the set of its components. For instance, the application of Figure 3 can be simply denoted by the fact

```
app(smartbuilding, [iot_controller, data_storage, dashboard]).
```

Then, they can define the *security requirements* of the application both as Component Requirements and Application Requirements. In our example, the Component Requirements can be simply declared as

```

secureComponent(iot_controller, N, D) :-
    physical_security(N),
    secure(iot_controller, N, D).

secureComponent(data_storage, N, D) :-
    secure_storage(N),
    access_logs(N),
    network_ids(N),
    secure(data_storage, N, D).

secureComponent(dashboard, N, D) :-
    host_ids(N),
    resource_monitoring(N),
    secure(dashboard, N,D).

```

where the custom security policies `physical_security(N)` and `secure_storage(N)` are defined as

```

secure_storage(N) :-
    backup(N),
    (encrypted_storage(N); obfuscated_storage(N)).

physical_security(N) :- anti_tampering(N); access_control(N).

```

Analogously, the Application Requirements that concern the application as a whole can be specified by extending the default policy `secureApp(A,D)` as follows

```

mySecureApp(A,D) :-
    secureApp(A,D),
    deployment(L,D),
    extras(D).

```

where the custom security policy `extras(N)` checking for Public Key Cryptography and Authentication at all nodes are (recursively) defined as

```

extras([]).
extras([d(C,N)|Ds]) :-
    public_key_cryptography(N),
    authentication(N),
    extras(Ds).

```

Finally, application operators can express their *trust degrees* towards each infrastructure operator as the probability of trusting it (i.e., $t \in [0, 1]$). In our example, we have

```

0.8::trustable(cloudOp1).
0.8::trustable(cloudOp2).
0.9::trustable(fogOp).
trustable(appOp).

```

Our prototype can be used to find (via a *generate & test* approach) all deployments that satisfy the security requirements of the example application to a given infrastructure, by simply issuing the query⁶

```

query(mySecureApp(smartbuilding,L)).

```

As shown in Figure 5, relying on ProbLog out-of-the-box algorithms, SecFog prototype returns answers to the query along with a value in $[0, 1]$ that represents the aggregate *security level* of the inferred facts, i.e. the probability that a deployment can be considered secure both according to the declared reliability of the infrastructure capabilities and to the trust degree of the application operator in each exploited infrastructure provider.

If the application operator is only considering security as a parameter to lead her search, she would try to maximise the obtained metric and, most probably, deploy all three components to Fog 3. However, security might need to be considered together with other parameters so to find a trade-off among them. In the next section, we propose a simple multi-objective optimisation and we apply it to our motivating example.

5 Multi-Objective Optimisation

Naturally, the quantitative results obtained with ProbLog can be used to optimise the security level of any application deployment, by simply taking the maximum value for our query. As we will show over an example in the next section, it is possible to exploit the SecFog methodology to optimise the security level together with other metrics. In this work, as in [14], given a deployment Δ , we will try to optimise the objective function

$$r(\Delta) = \sum_{m \in M} \omega_m \cdot \widehat{m}(\Delta)$$

⁶ Naturally, it is also possible to specify one particular deployment and assess its security level only.

```

mySecureApp(smartbuilding,[d(iot_controller,cloud1), d(data_storage,cloud1), d(dashboard,cloud1)]): 0.79928029
mySecureApp(smartbuilding,[d(iot_controller,cloud1), d(data_storage,cloud1), d(dashboard,cloud2)]): 0.63699776
mySecureApp(smartbuilding,[d(iot_controller,cloud1), d(data_storage,cloud1), d(dashboard,fog3)]): 0.69114513
mySecureApp(smartbuilding,[d(iot_controller,cloud1), d(data_storage,fog3), d(dashboard,cloud1)]): 0.67752684
mySecureApp(smartbuilding,[d(iot_controller,cloud1), d(data_storage,fog3), d(dashboard,cloud2)]): 0.53996463
mySecureApp(smartbuilding,[d(iot_controller,cloud1), d(data_storage,fog3), d(dashboard,fog3)]): 0.66417689
mySecureApp(smartbuilding,[d(iot_controller,cloud2), d(data_storage,cloud1), d(dashboard,cloud1)]): 0.63757163
mySecureApp(smartbuilding,[d(iot_controller,cloud2), d(data_storage,cloud1), d(dashboard,cloud2)]): 0.63642441
mySecureApp(smartbuilding,[d(iot_controller,cloud2), d(data_storage,cloud1), d(dashboard,fog3)]): 0.55131415
mySecureApp(smartbuilding,[d(iot_controller,cloud2), d(data_storage,fog3), d(dashboard,cloud1)]): 0.54045108
mySecureApp(smartbuilding,[d(iot_controller,cloud2), d(data_storage,fog3), d(dashboard,cloud2)]): 0.67448315
mySecureApp(smartbuilding,[d(iot_controller,cloud2), d(data_storage,fog3), d(dashboard,fog3)]): 0.66238504
mySecureApp(smartbuilding,[d(iot_controller,fog2), d(data_storage,cloud1), d(dashboard,cloud1)]): 0.5827336
mySecureApp(smartbuilding,[d(iot_controller,fog2), d(data_storage,cloud1), d(dashboard,cloud2)]): 0.46441781
mySecureApp(smartbuilding,[d(iot_controller,fog2), d(data_storage,cloud1), d(dashboard,fog3)]): 0.55988355
mySecureApp(smartbuilding,[d(iot_controller,fog2), d(data_storage,fog3), d(dashboard,cloud1)]): 0.54885163
mySecureApp(smartbuilding,[d(iot_controller,fog2), d(data_storage,fog3), d(dashboard,cloud2)]): 0.54687823
mySecureApp(smartbuilding,[d(iot_controller,fog2), d(data_storage,fog3), d(dashboard,fog3)]): 0.67268088
mySecureApp(smartbuilding,[d(iot_controller,fog3), d(data_storage,cloud1), d(dashboard,cloud1)]): 0.70503715
mySecureApp(smartbuilding,[d(iot_controller,fog3), d(data_storage,cloud1), d(dashboard,cloud2)]): 0.56188935
mySecureApp(smartbuilding,[d(iot_controller,fog3), d(data_storage,cloud1), d(dashboard,fog3)]): 0.69114513
mySecureApp(smartbuilding,[d(iot_controller,fog3), d(data_storage,fog3), d(dashboard,cloud1)]): 0.67752684
mySecureApp(smartbuilding,[d(iot_controller,fog3), d(data_storage,fog3), d(dashboard,cloud2)]): 0.67509079
mySecureApp(smartbuilding,[d(iot_controller,fog3), d(data_storage,fog3), d(dashboard,fog3)]): 0.83038718

```

Fig. 5. Results of the motivating example.

where M is the set of metrics to be optimised, ω_m is the weight⁷ assigned to each metrics (so that $\sum_{m \in M} \omega_m = 1$) and $\widehat{m}(\Delta)$ is the normalised value of metric m for deployment Δ , which – given the set D of candidate deployments – is computed as:

$$\begin{aligned}
- \widehat{m}(\Delta) &= \frac{m(\Delta) - \min_{d \in D} \{m(d)\}}{\max_{d \in D} \{m(d)\} - \min_{d \in D} \{m(d)\}} \text{ when the } m(\Delta) \text{ is to be maximised, and} \\
- \widehat{m}(\Delta) &= \frac{\max_{d \in D} \{m(d)\} - m(\Delta)}{\max_{d \in D} \{m(d)\} - \min_{d \in D} \{m(d)\}} \text{ when } m(\Delta) \text{ is to be minimised.}
\end{aligned}$$

Therefore, since we assumed that the higher the value of $r(\Delta)$ the better is deployment Δ , we will choose $\overline{\Delta}$ such that $r(\overline{\Delta}) = \max_{\Delta \in D} \{r(\Delta)\}$. In what follows, we solve the motivating example by employing this optimisation technique on all attributes of Table 1 along with the security levels computed in Section 4.

5.1 Motivating Example Continued

In our motivating example, we will attempt to maximise QoS-assurance and security, whilst minimising cost (in which we include the cost for the 4G connection at Fog 2 when needed). However, different application operators may want to either maximise or minimise the Fog resource consumption of their deployment, i.e. they may look for a Fog-ward or for a Cloud-ward deployment. Hence, concerning this parameter, we will consider both situations. Table 2 show the values of the Fog-ward (i.e., $r_F(\Delta)$) and of the Cloud-ward (i.e., $r_C(\Delta)$) objective function.

⁷ For the sake of simplicity, we assume here $\omega_m = \frac{1}{|M|}$, which can be tuned differently depending on the needs of the application operator.

Table 2. Ranking of eligible deployments.

Dep. ID	IoTController	DataStorage	Dashboard	$r_F(\Delta)$	$r_C(\Delta)$
$\Delta 1$	Fog 2	Fog 3	Cloud 2	0.53	0.28
$\Delta 2$	Fog 2	Fog 3	Cloud 1	0.63	0.38
$\Delta 3$	Fog 3	Fog 3	Cloud 1	0.85	0.60
$\Delta 6$	Fog 3	Fog 3	Cloud 2	0.75	0.50
$\Delta 15^*$	Fog 2	Cloud 1	Cloud 2	0.15	0.40
$\Delta 16^*$	Fog 2	Cloud 1	Cloud 1	0.51	0.76

In the Fog-ward case, when looking for the best trade-off among QoS-assurance, resource consumption, cost and security level, the most promising deployment is not $\Delta 2$ anymore (as it was in [5]). Indeed, $\Delta 3$ scores a much better ranking when compared to $\Delta 2$. Furthermore, in the Fog-ward case, the 4G upgrade at Fog 2, which makes it possible to enact $\Delta 15$ and $\Delta 16$, is not worth the investment due to the low score of both deployments. Conversely, in the Cloud-ward case (even though $\Delta 3$ would still be preferable), $\Delta 16$ features a good ranking value, despite requiring to upgrade the connection available at Fog 2.

6 Concluding Remarks

In this paper, we proposed a declarative methodology, **SecFog**, which can be used to assess the security level of multi-component application deployments to Fog computing infrastructures. With a proof-of-concept implementation in ProbLog, we have shown how **SecFog** helps application operators in determining secure deployments based on specific application requirements, available infrastructure capabilities, and trust degrees in different Fog and Cloud providers. We have also shown how **SecFog** can be used synergically with other predictive methodologies to perform multi-objective optimisation of security along with other metrics (e.g., deployment cost, QoS-assurance, resource usage). In our future work we plan to:

- enhance **SecFog** by combining it with existing strategies that have been used to quantify trust degrees (e.g., Bayesian or Dempster–Shafer theories as in [34]) based on direct experience, possibly considering also the mobility of Fog nodes and IoT devices,
- evaluate the possibility to use **SecFog** with meta-heuristic optimisation techniques (e.g., genetic or swarm intelligence algorithms), also taming the time complexity of the generate & test approach we prototyped, and
- further engineer our proof-of-concept implementation and show its applicability to actual use cases (e.g., based on the Fog application of [7]).

References

1. OpenFog Consortium. <http://www.openfogconsortium.org/>

2. EU Cloud SLA Standardisation Guidelines (2014), <https://ec.europa.eu/digital-single-market/en/news/cloud-service-level-agreement-standardisation-guidelines>
3. Belle, V.: Logic meets probability: towards explainable ai systems for uncertain worlds. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI. pp. 19–25 (2017)
4. Brogi, A., Forti, S.: QoS-Aware Deployment of IoT Applications Through the Fog. *IEEE Internet of Things Journal* **4**(5), 1185–1192 (Oct 2017)
5. Brogi, A., Forti, S., Ibrahim, A.: How to best deploy your Fog applications, probably. In: Rana, O., Buyya, R., Anjum, A. (eds.) Proceedings of 1st IEEE Int. Conference on Fog and Edge Computing (2017)
6. Brogi, A., Forti, S., Ibrahim, A.: Deploying fog applications: How much does it cost, by the way? In: Proceedings of the 8th International Conference on Cloud Computing and Services Science. pp. 68–77. SciTePress (2018)
7. Brogi, A., Forti, S., Ibrahim, A., Rinaldi, L.: Bonsai in the fog: An active learning lab with fog computing. In: Fog and Mobile Edge Computing (FMEC), 2018 Third International Conference on. pp. 79–86. IEEE (2018)
8. Choo, K.K.R., Lu, R., Chen, L., Yi, X.: A foggy research future: Advances and future opportunities in fog computing research (2018)
9. Dastjerdi, A.V., Buyya, R.: Fog computing: Helping the internet of things realize its potential. *Computer* **49**(8), 112–116 (Aug 2016)
10. De Raedt, L., Kimmig, A.: Probabilistic (logic) programming concepts. *Machine Learning* **100**(1), 5–47 (2015)
11. De Raedt, L., Kimmig, A., Toivonen, H.: Problog: A probabilistic prolog and its application in link discovery. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence. pp. 2468–2473 (2007)
12. Goettelmann, E., Dahman, K., Gateau, B., Dubois, E., Godart, C.: A security risk assessment model for business process deployment in the cloud. In: Services Computing (SCC), 2014 IEEE International Conference on. pp. 307–314. IEEE (2014)
13. Guerrero, C., Lera, I., Juiz, C.: A lightweight decentralized service placement policy for performance optimization in fog computing. *Journal of Ambient Intelligence and Humanized Computing* (Jun 2018)
14. Guerrero, C., Lera, I., Juiz, C.: Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications. *The Journal of Supercomputing* **74**(7), 2956–2983 (Jul 2018)
15. Gupta, H., Vahid Dastjerdi, A., Ghosh, S.K., Buyya, R.: iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments. *Software: Practice and Experience* **47**(9), 1275–1296 (2017)
16. Hamid Reza Arkian, Abolfazl Diyanat, A.P.: Mist: Fog-based data analytics scheme with cost-efficient resource provisioning for IoT crowdsensing applications. *Journal of Network and Computer Applications* **82**, 152 – 165 (2017)
17. Hong, H.J., Tsai, P.H., Hsu, C.H.: Dynamic module deployment in a fog computing platform. In: 2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS). pp. 1–6 (Oct 2016)
18. Kaur, A., Singh, M., Singh, P., et al.: A taxonomy, survey on placement of virtual machines in cloud. In: 2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS). pp. 2054–2058. IEEE (2017)
19. Luna, J., Taha, A., Trapero, R., Suri, N.: Quantitative reasoning about cloud security using service level agreements. *IEEE Transactions on Cloud Computing* **5**(3), 457–471 (July 2017)

20. Mahmud, R., Ramamohanarao, K., Buyya, R.: Latency-aware application module management for fog computing environments. *ACM Transactions on Internet Technology (TOIT)* (2018)
21. Mezni, H., Sellami, M., Kouki, J.: Security-aware SaaS placement using swarm intelligence. *Journal of Software: Evolution and Process* (2018)
22. Mukherjee, M., Matam, R., Shu, L., Maglaras, L., Ferrag, M.A., Choudhury, N., Kumar, V.: Security and privacy in fog computing: Challenges. *IEEE Access* **5**, 19293–19304 (2017)
23. Nacer, A.A., Goettelmann, E., Youcef, S., Tari, A., Godart, C.: Obfuscating a business process by splitting its logic with fake fragments for securing a multi-cloud deployment. In: *Services (SERVICES), 2016 IEEE World Congress on*. pp. 18–25. IEEE (2016)
24. Newman, S.: *Building microservices: designing fine-grained systems.* ” O’Reilly Media, Inc.” (2015)
25. Ni, J., Zhang, K., Lin, X., Shen, X.: Securing fog computing for internet of things applications: Challenges and solutions. *IEEE Comm. Surveys & Tutorials* (2017)
26. OpenFog: OpenFog Reference Architecture (2016)
27. Rahbari, D., Nickray, M.: Scheduling of fog networks with optimized knapsack by symbiotic organisms search. In: *2017 21st Conference of Open Innovations Association (FRUCT)*. pp. 278–283 (Nov 2017)
28. Schoenen, S., Mann, Z.Á., Metzger, A.: Using risk patterns to identify violations of data protection policies in cloud systems. In: *International Conference on Service-Oriented Computing*. pp. 296–307. Springer (2017)
29. Skarlat, O., Nardelli, M., Schulte, S., Dustdar, S.: Towards qos-aware fog service placement. In: *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*. pp. 89–96 (May 2017)
30. Taneja, M., Davy, A.: Resource aware placement of iot application modules in fog-cloud computing paradigm. In: *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. pp. 1222–1228 (May 2017)
31. Tang, Z., Zhou, X., Zhang, F., Jia, W., Zhao, W.: Migration modeling and learning algorithms for containers in fog computing. *IEEE Transactions on Services Computing* (2018)
32. Varshney, P., Simmhan, Y.: Demystifying Fog Computing: Characterizing Architectures, Applications and Abstractions. In: *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*. pp. 115–124 (2017)
33. Wang, S., Zafer, M., Leung, K.K.: Online placement of multi-component applications in edge computing environments. *IEEE Access* **5**, 2514–2533 (2017)
34. Wei, Z., Tang, H., Yu, F.R., Wang, M., Mason, P.: Security enhancements for mobile ad hoc networks with trust management using uncertain reasoning. *IEEE Transactions on Vehicular Technology* **63**(9), 4647–4658 (Nov 2014)
35. Wen, Z., Yang, R., Garraghan, P., Lin, T., Xu, J., Rovatsos, M.: Fog Orchestration for Internet of Things Services. *IEEE Internet Computing* **21**(2), 16–24 (2017)
36. Wen, Z., Cala, J., Watson, P., Romanovsky, A.: Cost effective, reliable and secure workflow deployment over federated clouds. *IEEE Transactions on Services Computing* **10**(6), 929–941 (2017)
37. Zhang, P., Zhou, M., Fortino, G.: Security and trust issues in fog computing: A survey. *Future Generation Computer Systems* **88**, 16–27 (2018)

Towards a Generalizable Comparison of the Maintainability of Object-Oriented and Service-Oriented Applications

Justus Bogner^{1,2}, Bhupendra Choudhary², Stefan Wagner², and Alfred Zimmermann¹

¹ University of Applied Sciences Reutlingen, Germany
{justus.bogner,alfred.zimmermann}@reutlingen-university.de

² University of Stuttgart, Germany
{justus.bogner,stefan.wagner}@informatik.uni-stuttgart.de
bhupendra.choudhary@gmx.de

Abstract. While there are several theoretical comparisons of Object Orientation (OO) and Service Orientation (SO), little empirical research on the maintainability of the two paradigms exists. To provide support for a generalizable comparison, we conducted a study with four related parts. Two functionally equivalent systems (one OO and one SO version) were analyzed with coupling and cohesion metrics as well as via a controlled experiment, where participants had to extend the systems. We also conducted a survey with 32 software professionals and interviewed 8 industry experts on the topic. Results indicate that the SO version of our system possesses a higher degree of cohesion, a lower degree of coupling, and could be extended faster. Survey and interview results suggest that industry sees systems built with SO as more loosely coupled, modifiable, and reusable. OO systems, however, were described as less complex and easier to test.

Keywords: Maintainability · Service Orientation · Object Orientation · Metrics · Experiment · Survey · Interviews

1 Introduction

The ability to quickly and cost-efficiently change applications and services due to new or redacted requirements is important for any company relying on custom software. The associated quality attribute is maintainability: the degree of effectiveness and efficiency with which software can be changed [5], e.g. to adapt or extend it. The introduction of Object Orientation (OO) lead to maintainability-related benefits like encapsulation, abstraction, inheritance, or increased support for modularization [3]. In today's enterprise world, however, systems built on Service Orientation (SO) are increasingly more common. By introducing a higher level of abstraction, Service-Based Systems (SBSs) consist of loosely coupled distributed components with well defined technology-agnostic interfaces [7]. SO

aims to promote interoperability, reuse of cohesive functionality at a business-relevant abstraction level, and encapsulation of implementation details behind published interfaces [4].

So while Service Orientation seems to surpass Object Orientation w.r.t. maintainability from a theoretical point of view, this comparison is very hard to generalize in a practical setting. Developers can build systems of arbitrary quality in both paradigms, although the inherent properties of both paradigms may make it easier or harder to build well maintainable systems. Very little empirical research exists on the topic of comparing the maintainability of OO and SO (see Sect. 2). Results from such studies can bring valuable insights into the evolution qualities of these two paradigms. Research in this area can also highlight potential deficiencies and weaknesses, which helps raising awareness for developers as well as providing decision support for choosing a paradigm for a project.

This is why we conducted a study to compare the maintainability of object-oriented and service-oriented applications from different perspectives. For a practical empirical point of view, we constructed two functionally equivalent systems (one based on OO and the other on SO) and compared them with metrics as well as by means of a controlled software development experiment. To gain insight into software professionals' subjective estimation of the two paradigms, we conducted an industry survey as well as expert interviews. In the remainder of this paper, we first introduce related work in this area. Then we present the details of our 4-part study including the methods, results, and limitations. Lastly, we conclude by summarizing our results and putting them into perspective.

2 Related Work

A small number of scientific publications exists that compare Service Orientation and Object Orientation. In 2005 when SBSs were still very young, Baker and Dobson [1] published a theoretical comparison of Service-Oriented Architecture (SOA) and Distributed Object Architectures (DOA) based on literature and personal experience. Their comparison is very high-level and not focused specifically on maintainability. While they highlight a large number of similarities, they also point out the more coarse-grained interfaces of SOA that lead to simplified communication and less cognitive overhead for developers of service consumers. Moreover, they point out the missing notion of inheritance and interface specialization in SOA, which they acknowledge as initially less complex, but potentially limiting in the long term.

Stubbings [10] provided another theoretical comparison that also emphasizes the direct line of evolution from OO to SO. Beneficial OO concepts like encapsulation and reuse have been adapted to a higher abstraction level in Service Orientation that is closer to the business domain. He further assessed the structural and technological complexity to be higher in a system based on Service Orientation. Concerning communication, he reported the focus for OO to be primarily internal while SO would be more aimed at external interoperability.

One of the few empirical studies on the subject was performed by Perepletchikov et al. [8] on two versions of a fictional Academic Management System (one service-oriented version, the other one object-oriented). To compare the maintainability of the two, they employed traditional source code metrics like *Lines of Code*, *Cyclomatic Complexity*, as well as the OO metrics suite from Chidamber and Kemerer. They focused on the structural properties size, complexity, coupling, and cohesion. As findings, they reported that the SO version provides better separation of business and implementation logic and a lower degree of coupling. The OO system, however, would be overall less complex.

Lastly, Mansour and Mustafa [6] conducted a similar empirical study. They constructed a service-oriented version of an existing OO Automated Teller Machine system and compared the two versions with a set of metrics, very similar to the ones in [8]. They reported that the SO version of their system inhibited a higher degree of reusability and a lower degree of coupling while the complexity of the OO version was lower. Additionally, they described difficulties when trying to apply OO metrics to a Service-Based System and advocated the need for a set of service-oriented maintainability metrics.

Existing studies are either of a theoretical nature or solely focused on metrics. While the presented empirical studies provide first valuable support for a comparison with metrics, they also reported difficulties due to a lack of mutually applicable metrics. Not all OO metrics can be used for SBSs. Moreover, additional metric evaluations with other systems will be of value while new approaches can bring different perspectives to the discussion.

3 Study Design

Based on the results and lessons learned of the related work, we therefore conducted a study with four different parts. First, we constructed a service-oriented and an object-oriented version of a simple Online Book Store (OBS) that provided functionality to register as a user as well as to browse and order books. The service-oriented version was implemented with RESTful NodeJS services using the Express framework³ and an Angular frontend⁴ while the object-oriented version is a Java monolith relying on JavaServer Pages (JSP) as a web UI. These two systems were compared using a set of **coupling and cohesion metrics**. To respect the two system versions, we needed metrics that can be applied both to service- as well as object-oriented systems. This is often difficult to achieve, since coupling and cohesion metrics are usually designed for either of the two paradigms. We therefore chose two metrics for each structural property that could be adapted to be mutually applicable.

For coupling, we chose *Absolute Importance of the Service* (AIS) and *Absolute Dependence of the Service* (ADS). Both have been specifically designed for SBSs and represent the number of clients invoking a service (AIS) and the number of

³ <https://expressjs.com>

⁴ <https://angular.io>

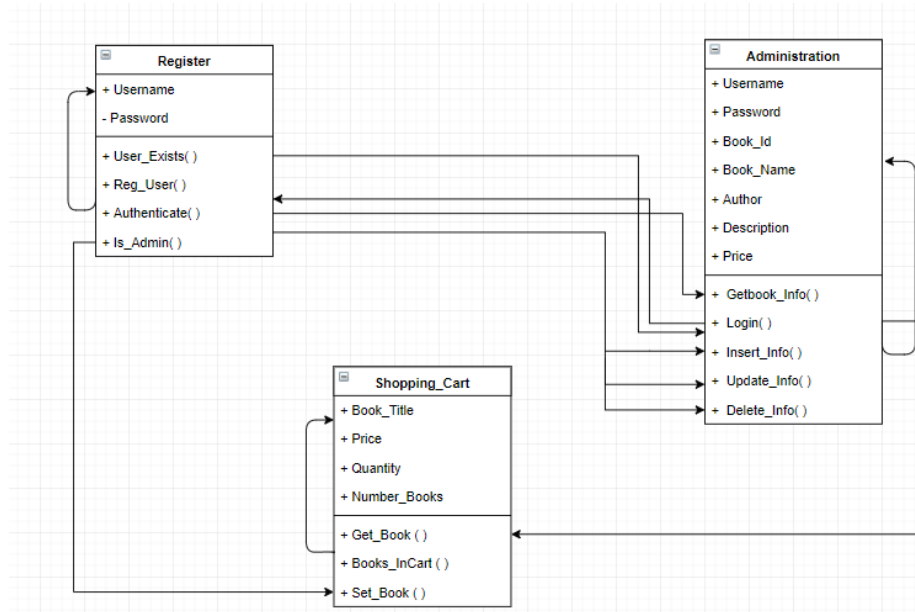


Fig. 1. Object-Oriented Version of OBS

other services a service depends on respectively (ADS) [9]. They can be easily adapted to object-oriented systems by substituting *services* with *classes*.

For cohesion, we selected two object-oriented metrics, namely *Tight Class Cohesion* (TCC) and *Loose Class Cohesion* (LCC) [2]. These metrics attempt to measure the relatedness of class functionality based on common class attributes that the methods operate on. TCC represents the relative number of directly connected methods while LCC also includes indirectly connected methods (via other intermediate methods). To adapt these metrics to a service-oriented context, class methods are substituted by service operations.

While the majority of maintainability metrics use structural properties as a proxy, industry is really interested in something else: how fast can changes or features be implemented for the system? To account for this, the same systems were used in a **controlled experiment**. Software practitioners had to implement search functionality for books while the time was measured. We then analyzed whether the version made a noticeable difference. 8 software developers participated in the experiment, four per system version of OBS. 7 of the 8 developers were from Germany. They had an average of ~ 4.1 years of experience (OO AVG: 4.5 years, SO AVG: 3.75 years). All of them had worked with their respective paradigm before. We measured the time necessary to complete the exercise as well as the changed Lines of Code for the backend part.

To complement these two empirical approaches, we also conducted an **industry survey** to capture the general sentiment of developers towards the two

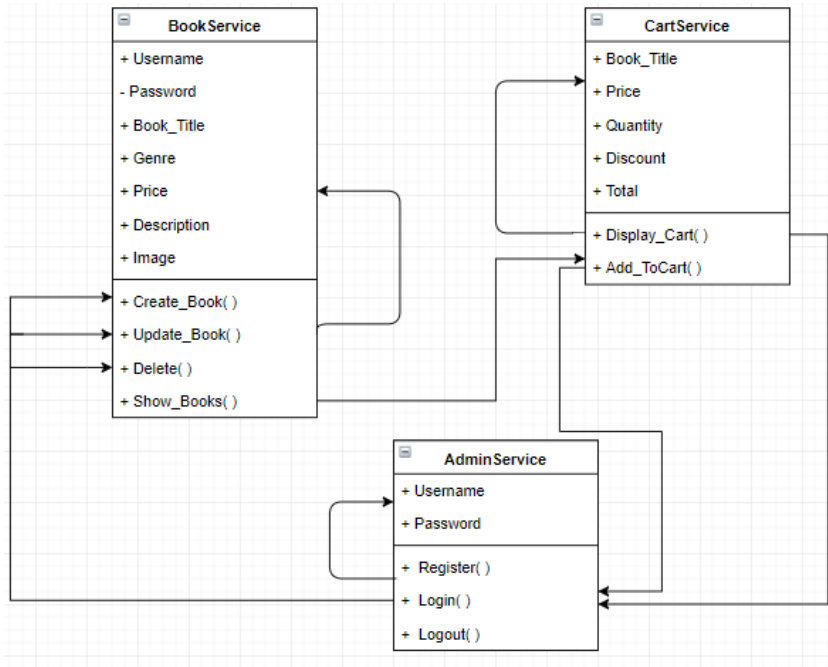


Fig. 2. Service-Oriented Version of OBS

paradigms. Software professionals filled out an online questionnaire where they were asked to compare structural and maintainability-related properties of the two paradigms based on their personal experience. 32 participants completed our web-based questionnaire that was distributed via personal industry contacts, mailing lists, and social media. The survey was hosted from 2018-04-19 until 2018-05-06 and consisted of 12 questions, mostly with Likert scale answers. Most participants were from Germany and India and all had at least three years of professional experience. They had to comment on the average condition of different structural properties (e.g. coupling) and subquality attributes of maintainability in SW projects based on either OO or SO. Lastly, they had to answer some questions where they ranked the three paradigms *Object Orientation*, *Service Orientation*, and *Component-Based* for similar attributes.

As a more in-depth follow-up to the survey, we conducted **qualitative interviews** with several experts to complement the broader scope of the survey and to dive more deeply into some of the topics. Similar to the survey, we also asked for their personal experience and preference w.r.t. the maintainability of the two paradigms under study. This was the fourth and final part of our study. All 8 experts had an IT or Engineering background and had previously worked with object-oriented as well as service-oriented systems. 7 of the 8 experts were older than 30 years, i.e. had considerable professional experience. The interviews started with an introduction of the two OBS versions and a discussion

about their strengths and weaknesses. This was followed by similar questions as in the survey about properties of the two paradigms and the participants’ experience.

Please refer to our GitHub repository for the source code of the systems as well as the detailed survey questions and results ⁵.

4 Results

For the metric-based part of the study, we measured all four **component-level metrics** for both the object-oriented (Fig. 1) and the service-oriented version (Fig. 2) of the Online Book Store (OBS). Since each version of the system includes three components (services or classes respectively), we have a total of 12 measurements (see Table 1). When looking at the AVG values per version and metric (see Fig. 3), we can see that the service-oriented version overall has slightly better values, i.e. on average lower coupling and higher cohesion per component.

Table 1. Coupling and Cohesion Metric Values per Component

	Component	AIS	ADS	TCC	LCC
OO Version	Administration	1	2	0.00	0.40
	Register	1	2	0.16	0.50
	Shopping_Cart	2	0	0.33	0.33
SO Version	AdminService	1	1	0.67	0.67
	BookService	1	1	0.33	0.50
	CartService	1	1	1.00	1.00

During the **controlled experiment**, it took less time and effort to extend the service-oriented version of OBS (see Fig. 4). The mean duration for the SO version was 0.8 h while it was 0.99 h for the OO version. Respectively, the mean effort was 7.25 LoC for SO and 12.5 LoC for OO. When analyzing the significance of the mean differences in our sample with an unpaired t-test, we found two-tailed p-values smaller than 0.05 (p-value_{duration}: 0.0479, p-value_{effort}: 0.005).

The following part highlights the results of the **survey questions**. For Likert scale question, we also present the aggregated score per paradigm (Strongly Disagree: -2, Disagree: -1, Neutral: 0, Agree: 1, Strongly Agree: 2).

Question: *In my experience, software based on <paradigm> has a comparatively low degree of **coupling**.*

⁵ <https://github.com/xJREB/research-oo-vs-so>

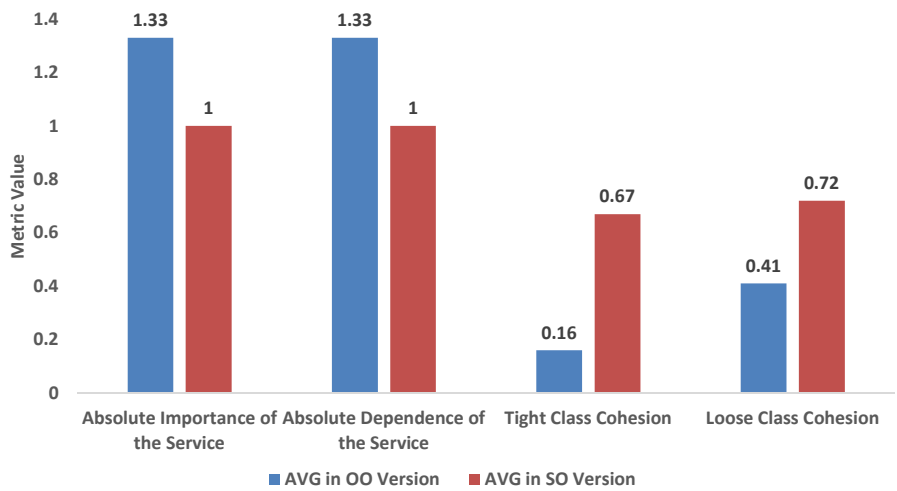


Fig. 3. Average Coupling and Cohesion Metric Values per Version

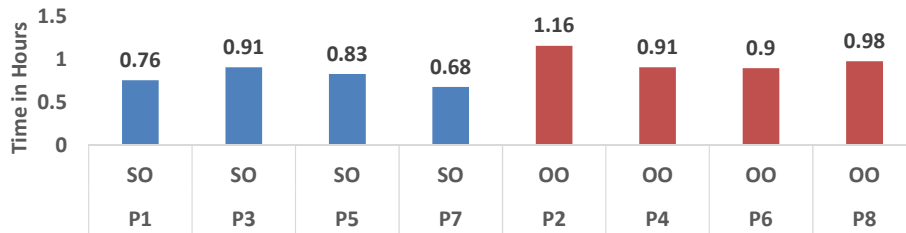


Fig. 4. Experiment: Duration per Participant

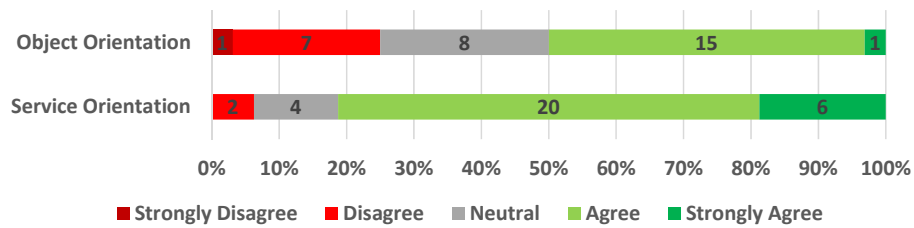


Fig. 5. Question: In my experience, software based on <paradigm> has a comparatively low degree of coupling.

For coupling, participants clearly favored Service Orientation (score: 30) over Object Orientation (score: 8). Over 80% reported that service-oriented systems were in their experience of a more loosely coupled nature while only 50% reported the same for object-oriented systems (see Fig. 5). This result was to be expected, since loose coupling and the reduction of dependencies is a major driver in SBSs.

Question: *In my experience, software based on <paradigm> facilitates a comparatively high degree of **cohesion**.*

When it came to cohesion, the results were less decisive (SO: 18, OO: 14). Overall, roughly 13% more participants agreed with this statement for Service Orientation (SO: ~63%, OO: 50%). This does not seem to be a lot, when we consider the prevalence of the “cohesive services grouped around business capabilities” theme in an SOA and especially in a Microservices context.

Question: *In my experience, software based on <paradigm> promises a significant extent of **reusability**.*

Participants reported higher reusability for their service-oriented software than for their object-oriented software. While the scores were pretty even (SO: 25, OO: 22), ~78% of participants agreed to this statement for SO while only ~59% agreed for OO. Absolute scores are so close because two more people disagreed for SO and one more strongly agreed for OO (see Fig. 6). Overall, these results seem to support the SO principle of business-relevant reuse granularity.

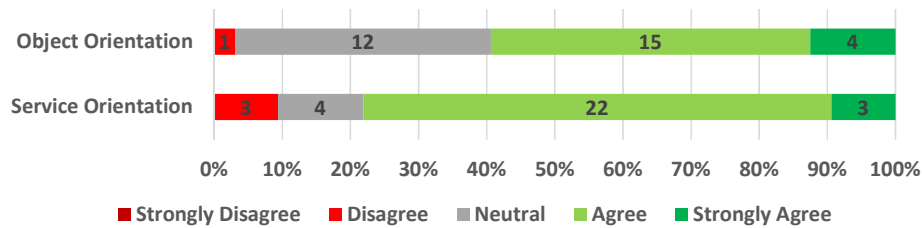


Fig. 6. Question: In my experience, software based on <paradigm> promises a significant extent of **reusability**.

Question: *In my experience, software based on <paradigm> reduces the complexity of **testing**.*

In the case of testability, Object Orientation (score: 24) was seen as more beneficial than Service Orientation (score: 14) to reduce complexity. Roughly 72% of participants agreed with this statement for OO while only ~53% agreed for SO together with 6 disagreements (see Fig. 7). This is the first category where OO decisively wins out in the opinion of participating developers.

Lastly, developers were asked to rank the three paradigms *Object Orientation*, *Service Orientation*, and *Component-Based* from their experience for three further properties: modifiability, encapsulation/abstraction, and size/complexity. Ranking a paradigm first provided three points, ranking it second provided two, ranking it last provided one point respectively. The results (see Table 2) indi-

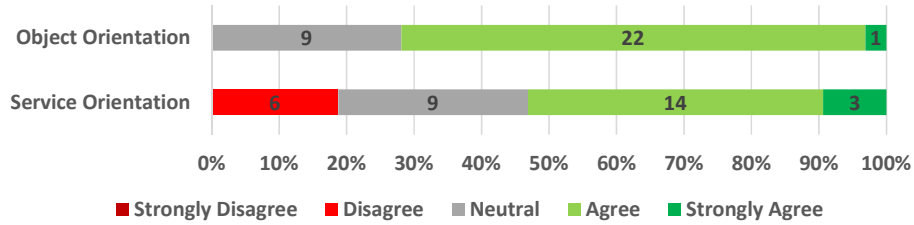


Fig. 7. Question: In my experience, software based on <paradigm> reduces the complexity of testing.

Table 2. Question: In your experience, which of the three paradigms provides on average the most favorable degree of <attribute>?

	Object Orientation	Component-Based	Service Orientation
Modifiability	63	43	86
Encapsulation and Abstraction	58	43	85
Size and Complexity	74	39	73

cate that participants experienced systems based on Service Orientation as more modifiable and with a better degree of encapsulation and abstraction as for the other two paradigms. For size and complexity, however, participants reported that they believed the manageability of these properties to be roughly equal for OO and SO, with OO winning out by one point.

We compiled results from the **qualitative interviews** in several areas. For the topic of *modifiability*, 5 of the 8 experts reported that on average in their experience service-oriented systems are more beneficial than object-oriented ones when it comes to evolving already developed systems. Participants emphasized the advantages of service-based modularity, which would increase independence in the system and reduce costs in the long run. Some experts highlighted that SO is more convenient when requirements frequently change.

Concerning *complexity*, most experts indicated based on their past software projects that systems based on Object Orientation are on average less complex than SBSs from a structural and technological point of view. They also mentioned mature tool support in the field of object-oriented SW development that would ease some of the difficulties. In the service-oriented space, however, tool support would be lacking.

When comparing the average *analyzability* of the two paradigms, the majority of participants favored Service Orientation over Object Orientation. The structure of the system would be easier to grasp when referring to services as coarse-grained components. Moreover, experts experienced less dependencies in SBSs, which also helped to comprehend the structure of a system.

Lastly, in addition to the lack of mature tool support for Service Orientation, participants reported the danger of ripple effects when changing services, especially with service interface changes that require updates of all service consumers. Some experts also stressed that Object Orientation was a valuable paradigm to be used for the inner low-level design of single services and that it would nicely complement the service-based high-level architecture of a system. So the choice would not always be either Service or Object Orientation.

5 Threats to Validity

Several things have to be mentioned to put our results into appropriate perspective. For the **metric-based evaluation**, the tested systems were artificially constructed and are not real industry or open source systems. While we tried to design and implement them as close to a real use case as possible, we also needed something of manageable size and complexity, which may impact the generalizability of the comparison (e.g. the AVG metric values were computed from only three components). The chosen technology for both versions may also be a limitation. Results with other programming languages or frameworks could be different. Moreover, we only used a small number of metrics and targeted only two structural properties (coupling and cohesion). Other metrics, e.g. for size or complexity, could have yielded additional insights, but were neglected due to project time constraints. Finally, we calculated the metric values manually due to missing tool support. Since the systems are of limited size and we double-checked each value, the error probability should still be very small.

In the case of the **controlled experiment**, the same limitations of the constructed systems as described above hold true. The two different programming languages (Java and NodeJS/JavaScript) also limit the comparability of the LoC effort. Additionally, we only had a small number of participants. Potentially different development experience and skill levels could not be accounted for when assigning the participants to the two versions of OBS. Lastly, the experiment consisted of only one exercise, which can only test the modifiability of certain parts of the system.

As with most **quantitative surveys**, a number of limitations have to be mentioned. First, the number of participants (32) only provides limited generalizability, as a different population subset may have different views on the subject. Moreover, we could not guarantee that the participating developers indeed had sufficient experience with all three software paradigms. Lastly, the subjective estimation of the inherent qualities of a paradigm may be skewed by a particularly bad experience with a suboptimally designed system. Overall, it is important to keep in mind that personal preference of developers is not necessarily of a rational nature.

As opposed to our survey participants, we could select our **interview experts** based on their experience with the two paradigms under evaluation, at least up to a certain degree. However, there is still a chance that some experts were less proficient with one of the paradigms or were heavily influenced by one

specific project of theirs. Moreover, there is a chance that we slightly influenced the experts by posing questions that should direct the conversation to the properties under evaluation. Lastly, our interviews were conducted and analyzed in a fairly loosely structured manner without a rigorous methodology.

6 Conclusion

To provide additional evidence for a generalizable comparison of the maintainability of Service Orientation and Object Orientation, we conducted a study with four parts: a metric-based comparison of two functionally equivalent systems (one SO and one OO version); a controlled experiment where practitioners had to extend the same systems; an industry survey with comparative questions about OO and SO; and expert interviews as a more in-depth follow-up to the survey.

The empirical results indicate that the service-oriented version of our Online Book Store system consists of more cohesive and more loosely coupled components and could also be extended faster and with less effort (LoC) by experiment participants. Survey and interview results seem to go in the same direction: industry professionals experienced higher modifiability, lower degrees of coupling, higher reusability, and stronger encapsulation and abstraction in their service-oriented projects. For their average object-oriented systems, however, they reported comparatively lower complexity and better testability.

While these results can aid in the decision process for a paradigm and can highlight important maintainability-related focus points when designing systems with either paradigm, it is still important to remember that we can build software of arbitrary quality in both paradigms. Moreover, Object Orientation can be a useful complement for the inner architecture of services.

Acknowledgments This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program “Services Computing” (<http://www.services-computing.de/?lang=en>).

References

1. Baker, S., Dobson, S.: Comparing Service-Oriented and Distributed Object Architectures. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3760 LNCS, pp. 631–645 (2005)
2. Bieman, J.M., Kang, B.K.: Cohesion and reuse in an object-oriented system. In: *Proceedings of the 1995 Symposium on Software reusability - SSR '95*. pp. 259–262. ACM Press, New York, New York, USA (1995)
3. Booch, G.: *Object Oriented Analysis & Design with Application*. Pearson Education (2006)
4. Erl, T.: *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA (2005)

5. International Organization For Standardization: ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Tech. rep. (2011)
6. Mansour, Y.I., Mustafa, S.H.: Assessing Internal Software Quality Attributes of the Object-Oriented and Service-Oriented Software Development Paradigms: A Comparative Study. *Journal of Software Engineering and Applications* 04(04), 244–252 (2011)
7. Papazoglou, M.: Service-oriented computing: concepts, characteristics and directions. In: *Proceedings of the 7th International Conference on Properties and Applications of Dielectric Materials (Cat. No.03CH37417)*. pp. 3–12. IEEE Comput. Soc (2003)
8. Perepletchikov, M., Ryan, C., Frampton, K.: Comparing the impact of service-oriented and object-oriented paradigms on the structural properties of software. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 3762 LNCS, 431–441 (2005)
9. Rud, D., Schmietendorf, A., Dumke, R.R.: Product Metrics for Service-Oriented Infrastructures. In: *IWSM/MetriKon* (2006)
10. Stubbings, G.: Service-Oriented and Object-Oriented: Complementary Design Paradigms. *SPARK: The ACES Journal of Postgraduate Research* 1 (2010)

Implementation of a Cloud Services Management Framework

Hong Thai Tran¹ and George Feuerlicht^{1,2,3}

¹ Faculty of Engineering and Information Technology,
University of Technology Sydney, Australia

² Unicorn College, V Kapslovně 2767/2, 130 00 Prague 3, Czech Republic

³ Prague University of Economics, W. Churchill Square. 4, 130 67 Prague 3, Czech Republic

`HongThai.Tran@uts.edu.au`, `George.Feuerlicht@uts.edu.au`

Abstract. Rapid growth of various types of cloud services is creating new opportunities for innovative enterprise applications. As a result, enterprise applications are increasingly reliant on externally provided cloud services. It can be argued that traditional systems development methods and tools are not adequate in the context of cloud services and that new methods and frameworks that support these methods are needed for management of lifecycle of cloud services. In this paper, we describe the implementation of a Service Consumer Framework (SCF) – a framework for the management of design-time and runtime activities throughout the lifecycle of enterprise applications that use externally provided cloud services. The SCF framework has been evaluated during the implementation of a large-scale project and is being continuously improved to incorporate additional types of cloud services.

Keywords: Cloud Computing, Service Management, Frameworks

1 Introduction

Most enterprise applications today use third party cloud services to implement a significant part of their functionality. This results in hybrid environments that require the integration of on-premises services with public cloud services made available on a pay-per-use basis by external cloud providers. The use of third party cloud services (e.g. payment services, storage services, etc.) in enterprise applications has many benefits, but at the same time presents challenges as both the functional and non-functional characteristics of cloud services are controlled by autonomous cloud service providers. Service consumers are primarily responsible for the selection of services, integration of cloud services into on-premises enterprise applications and managing continuity of operation during runtime. With the increasing use of cloud services, it is important that cloud service consumers use suitable methods and tools to manage the entire lifecycle of enterprise applications [1]. A comprehensive framework is needed to support all

phases of service consumer lifecycle including the selection of cloud services, integration of services with enterprise applications and runtime monitoring and management of services.

Cloud services management has been an active area of research with numerous publications addressing different cloud service lifecycle phases, in particular cloud service selection [2-7] and service integration and monitoring [8-11]. However, most of these research efforts take service provider perspective and do not address the issues that arise when on-premises enterprise applications consume externally provided cloud services. A typical scenario illustrating this situation involves an on-premises application that consumes a range of cloud services (e.g. payment services: PayPal and eWay, storage services: DropBox, Google Drive and AWS S3, mapping services: Google Maps, etc.) via published APIs (Application Programming Interfaces) [12]. Management of such heterogeneous environments requires both design-time and run-time support to minimize the software maintenance effort and to ensure continuity of operation.

The main motivation for our research is to provide a detail description of the service development lifecycle as it applies to cloud service consumers (as distinct from cloud service providers) and to implement a prototype framework that supports this lifecycle. In our previous work we have proposed a Service Consumer Framework (SCF) [13] and described a cloud Service Consumer System Development Lifecycle (SC-SDLC) [14] for managing cloud services from a service consumer perspective. In this paper, we describe how the SCF supports design-time and run-time activities throughout the SC-SDLC (section 3), and detail the implementation of this framework (section 4). In the next section (section 2), we review related work on the methods and frameworks for the management of cloud services. Section 5 are our conclusions and directions for future work.

2 Related Work

While the management of cloud services in enterprise applications is still a subject of extensive investigation, there is a general agreement in the literature about the individual lifecycle phases. A method for managing integrated lifecycle of cloud services was proposed by Joshi et al. [15]. The authors have identified performance metrics associated with each lifecycle phase that include data quality, cost, and security metrics based on SLA (Service Level Agreement) and consumer satisfaction, and they have proposed a service repository with a discovery capability for managing cloud services lifecycle [16]. The authors divide cloud services lifecycle into five phases: requirements specification, discovery, negotiation, composition, and consumption. During the service discovery phase, service consumers search for services using service description and provider policies in a simple services database. Service information is stored as a Request for Service (RFS) that contains functional specifications, technical specifications, human agent policy, security policy and data quality policy. Field et al. [17] present a European Middleware Initiative (EMI) Registry that uses a decentralized architecture to support service discovery for both hierarchical and peering topologies. The objective of the EMI Registry is to provide robust and scalable service discovery that contains

two components: Domain Service Registry (DSR) and Global Service Registry (GSR). Service discovery is based on service information stored in service records that contain mandatory attributes such as service name, type of service, service endpoint, service interface, and service expiry date.

Cloud-based application development frameworks and architectures have been the subject of intense recent interest in the context of microservices and DevOps [18], [19]. According to Rimal et al. [20] the most important current challenge is the lack of a standard architectural approach for cloud computing. The authors explore and classify architectural characteristics of cloud computing and identify several architectural features that play a major role in the adoption of cloud computing. The paper provides guidelines for software architects for developing cloud architectures. Another notable effort in this area is the Seaclouds project [21, 22] that aims to develop a new open source framework for Seamless Adaptive Multi-Cloud management of service-based applications. The authors argue that lack of standardization results in vendor lock-in that affects all stages of the cloud applications' lifecycle, forcing application developers to have a deep knowledge of the providers' APIs. Seaclouds is a software platform and a reference architecture designed to address the heterogeneity of cloud service APIs at IaaS (Infrastructure as a Service) and SaaS (service as a Service) levels. The Seaclouds platform supports Discovery and Matchmaking, Cloud Service Optimization, Application Management, Monitoring and SLA Enforcement, and Application Migration. The authors of the Nucleous project [23] have investigated the practicability of abstracting the differences of vendor specific deployment and management APIs and creating an intermediary abstraction layer based on four selected PaaS platforms (cloudControl, Cloud Foundry, Heroku, and OpenShift), and concluded that the diversities among the platforms can be successfully harmonized. Using the Nucleous platform the effort involved in switching providers can be minimized, increasing the portability and interoperability of PaaS applications, helping to avoid critical vendor lock-in.

Unlike the above-mentioned initiatives, we do not aim to implement a framework for multi-cloud deployment, monitoring and orchestration of cloud services across multiple cloud platforms. Our focus is on designing a framework that improves the manageability and reliability of enterprise applications that consume cloud services from different providers with varied QoS (Quality of Service) characteristics.

3 Service Consumer Framework and SC-SDLC

Service Consumer Framework is a research prototype designed for the purpose of evaluating the functionality required for supporting the SC-SDLC. SCF constitutes a layer between on-premises enterprise applications and external cloud services and consists of four main components: Service Repository, Service Adaptors, Workflow Engine and Monitoring Centre. The service repository records information about enterprise applications and related cloud services throughout the entire service lifecycle. The service adaptor module contains adaptors for various categories of services. The function of a service adaptor is to present a unified API for services from different cloud providers for the same type of service (e.g. a payment service), transforming outgoing application

requests into the format supported by the current version of the corresponding external service, and incoming responses into format compatible with on-premises applications. The main function of the workflow engine is to provide failover capability in the event of a cloud service not being available by routing application requests to an alternative cloud service. The monitoring centre uses log data collected from service adaptors and the workflow engine to monitor cloud services and to analyze their runtime performance.

We have described the SC-SDLC in previous publications [24, 25]; in this section we briefly describe the main SC-SDLC lifecycle phases and discuss how the Service Consumer Framework supports activities during these phases. We have identified the following five phases of SC-SDLC: Requirements Specification, Service Identification, Service Integration, Service Monitoring and Service Optimization. We classify these phases into design-time activities: requirements specification, service identification and service integration, and run-time activities: service monitoring and service optimization. Typically, business analysts are involved with the requirements specification phase, while service identification, integration and optimization phases are the domain of application developers. Service monitoring phase is the responsibility of system administrators.

The SC-SDLC is closely interrelated with the Service Consumer Framework that provides support for lifecycle phases and activities. Figure 1 illustrates how the Service Consumer SDLC is supported by the Service Consumer Framework.

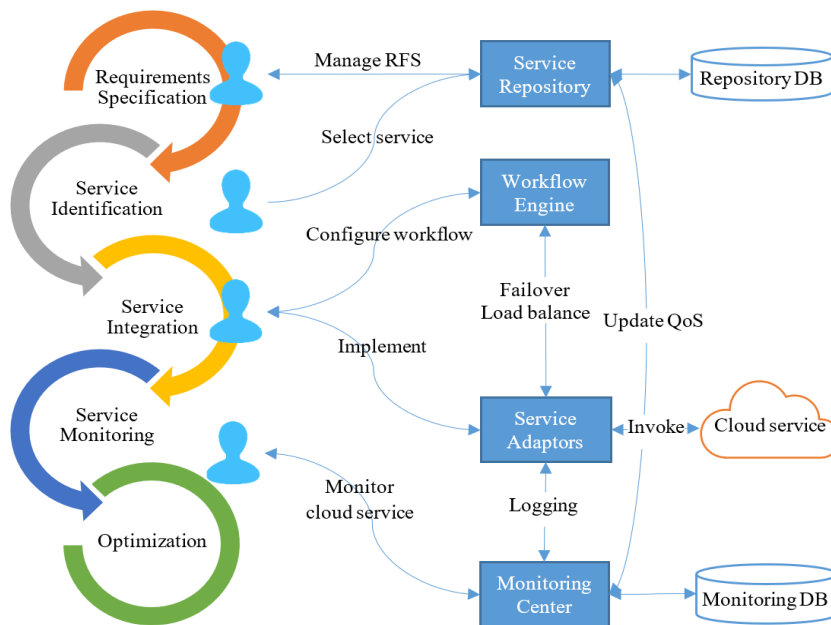


Fig. 1 Service Consumer Framework support for SC-SDLC phases

During the service requirements specification phase, business analysts record functional and non-functional requirements of the services in the service repository. Functional specification of the service describes what functions the service should provide and its characteristics may vary according to the type of service (i.e. application service, infrastructure service, etc.). The QoS (Quality of Service) non-functional attributes include service availability, response time, security requirements, and may also include requirements such as data location and the maximum cost of the service. Once the service is fully described and classified, the service consumer creates a Request for Service (RFS) and records this information in the service repository. Services are categorized according to service type (e.g. payment, storage, mapping, etc.) and this information is used during the service identification phase to search the service repository.

The service identification phase involves searching the service repository for services that match the RFS attempting to identify an existing service that is already certified for use (e.g. payment service with availability of 99.99 and sub-second response time). The SCF incorporates an API that supports a repository query function (described in section 4.1) that searches the service repository database for suitable candidate services. Service repository database stores detail information that includes service features available in different versions (i.e. functional description of the service) as well as non-functional parameters, including service reliability information recorded during runtime. Service repository can be searched based on various parameters to identify candidate services that are then checked for compatibility with the service specifications. If no suitable certified service is found, the service consumer will attempt to identify the service from the services available from external cloud providers. Following verification of the functionality and performance, the service is certified and recorded in the service repository. Certification involves extensive testing of the functionality and performance of the service. If no suitable cloud service is found, the service may have to be developed internally (i.e. as an on-premises service).

The service integration phase involves the integration of cloud services with on-premises enterprise applications. This activity varies depending on the type of cloud service and may involve the development of a service adaptor and design of specialized workflows to improve the reliability of applications by incorporating failover capability. The SCF repository records the relationships between services (service versions) and corresponding enterprise applications. The final activity of the cloud service integration phase comprises integration testing, provisioning, and deployment, similar to activities during the implementation of on-premises application.

The service monitoring phase involves measuring runtime QoS attributes and comparing their values with those specified in the corresponding SLA. System administrators use the monitoring centre to identify performance issues. Local monitoring is required as QoS values measured at the consumer site may differ from the values published by cloud service providers. Data generated during the monitoring phase is stored in the monitoring database.

The final SC-SDLC phase is concerned with service optimization and continuous service improvement. Service optimization may involve replacing existing services with new versions as these become available, or by identifying alternative cloud services from a different provider with improved QoS characteristics.

4 SCF Implementation

This section describes the implementation details of the components of the SCF framework. Additional implementation details are available in [26] and the SCF source code has been published on GitHub (<https://github.com/tranhongthai/SCF>). The SCF prototype is developed using .Net technologies: Microsoft SQL Server [27] was used to implement the service repository and monitoring center databases, ASP.Net MVC 5 [28] was used to build the service repository and the service monitoring center tools, and Windows Communication Foundation (WCF) [29] was used to implement service repository and monitoring centre APIs. The SCF is deployed on an AWS (Amazon Web Services) EC2 server and the databases are implemented as AWS RDS (Relational Database System) services. The workflow engine and service adaptors are implemented as Class Libraries (DLL) in C# programming language and released using NuGet - Microsoft package manager for .NET (<https://www.nuget.org/packages>). Table 1 lists the main SCF modules, technologies used for their implementation and deployment platforms.

Table 1. SCF implementation technologies and deployment platforms

SCF Modules	Implementation Technology	Deployment Platform
Service Repository	ASP.NET MVC	AWS EC2
	Microsoft SQL Server	AWS RDS
Service Adaptors	Class Library (.DDL)	Nuget
Workflow Engine	Class Library (.DDL)	Nuget
Monitoring Center	Windows Service Application	AWS EC2
	Microsoft SQL Server	AWS RDS
	Window Foundation Communication	

4.1 Service Repository

Service repository is a key component of SCF framework that maintains information about cloud services throughout the entire service lifecycle. A simplified data model (Entity-Relationship Diagram) of the service repository is shown in Figure 2. *Service* is a central entity of the service repository with attributes that describe services and include service requirements as captured by the SLA. In order to manage service evolution and keep track of changes in service functionality, information about service versions is stored in the repository. The *ServiceVersion* entity includes functional and non-functional descriptors of the service that are further described by the information in the related *QoS* and *ServiceFeature* entities. This allows service versions to have different QoS values and features. *ServiceCategory* is used to categorize services according to their type (e.g. payment, storage, etc.); the self-referencing relationship produces a service type hierarchy, so that for example, a storage service constitutes a subtype of an infrastructure service. *ServiceProvider* entity represents service providers and contains

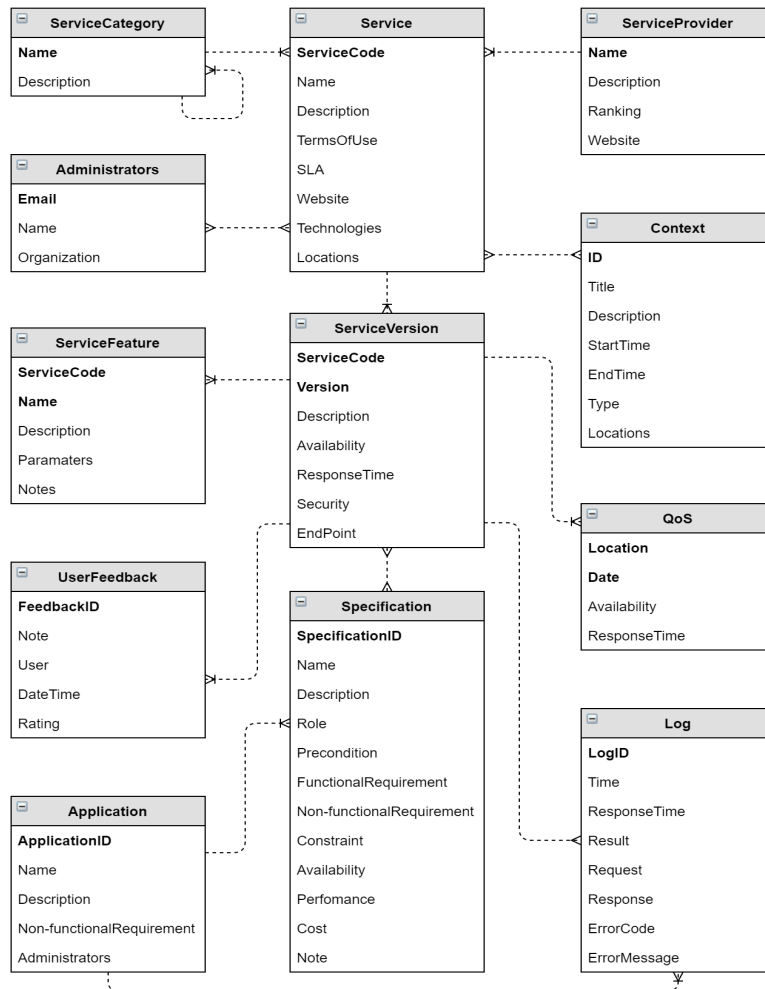


Fig. 2. Service Repository Entity Relationship Diagram

service provider attributes including provider description and provider ranking (indication of provider reputation). The *Application* entity represents on-premises applications that are associated with requirements specifications (*Specification*) that are matched with services (*ServiceVersion*) based on the compatibility of functional and non-functional attribute values. Results of service invocations are logged at runtime, and are represented by the *Log* entity. Service log records include response time, results of service invocations, and other non-functional attributes collected at runtime and used for analysis of service performance. Responsibility for managing services is assigned to system administrators and represented by the *Administrators* entity.

Service Repository Interface

Service repository APIs are implemented using WCF and provide access to repository information. The following methods have been implemented:

- **Search:** this method is used to query the service repository database and to retrieve services based on the specified values of QoS parameters (e.g. service type, availability, response time, etc.)

- **GetInfo:** this method retrieves information about a specific cloud service, including basic service description and QoS information
- **UpdateQoS:** this method is used to insert and update the QoS information for a specified cloud service

4.2 Service Adaptors

Service adaptors implement generic interfaces for different types of services (e.g. a payment service) that support common service functions (e.g. payment, refund, etc.). This allows runtime substitution of cloud services and can improve the overall reliability of enterprise applications. At design time, cloud services can be replaced by alternative services with improved QoS characteristics, as these become available. The main function of a service adaptor is to transform application requests into the format supported by the current version of the corresponding external service, and to ensure that incoming responses maintain compatibility with internal applications. Generic messages and methods that support common service functions are defined for each service type and mapped into the corresponding messages and methods of specific cloud provider services. So that for example, the Dropbox adaptor transforms the generic *Download* request into the Dropbox *DownloadAsync* request, and the Google Drive Adaptor transforms this request into *GetByteArrayAsync* request. The use of service adaptors across all enterprise applications alleviates the need to modify individual applications when a new version of the cloud service API is released. Another function of service adaptors is to perform runtime logging of performance parameters that are used to calculate QoS attributes.

Service Adaptor Library

The Service Adaptor Library contains generic APIs that include methods for various types of services. For example, a generic payment service interface contains three common methods: *Pay*, *Refund*, and *CheckBalance* that use generic messages (*PaymentRequest*, *PaymentResponse*, *RefundRequest*, *RefundResponse*, etc.). Service adaptors inherit the generic interface and implement the body of the methods. The Service Adaptor Library currently contains adaptors for PayPal, eWay, Stripe payment services, and Dropbox and GoogleDrive storage services. We intent to expand the range of adaptors, but at the same time we recognize that this may not be a workable solution in situations where the functionality of services from different cloud providers is significantly different.

4.3 Workflow Engine

The purpose of the workflow engine is to implement simple workflows using a combination of adaptors and pre-defined sub-workflows (i.e. workflow fragments that implement a specific function, e.g. the Retry Fault Tolerance reliability strategy). The SCF workflow engine it is not intended to replicate a fully-functional orchestration engine

(e.g. BPEL engine). The workflow engine determines the sequence of service invocations for a given application requirement, and is typically used to configure adaptors to

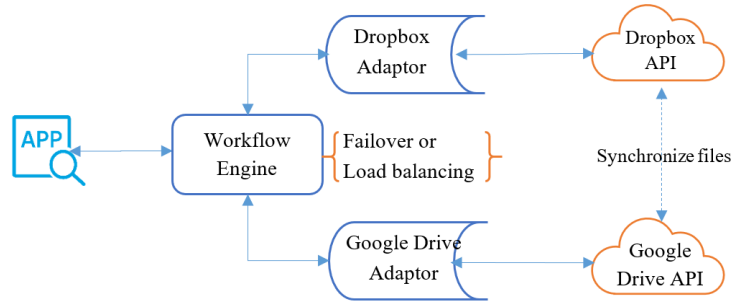


Fig. 3. Example of a fault tolerant cloud storage workflow

provide failover function using various fault tolerance strategies. We have demonstrated, using payment services PayPal and eWay, that relatively simple fault tolerance strategies such as Retry Fault Tolerance, Recovery Block Fault Tolerance or Dynamic Sequential Fault Tolerance strategy can lead to significant improvements in application availability [30]. Figure 3 shows an example of a workflow that uses Dropbox and Google Drive as alternative storage systems. During normal operation, the data is replicated across both storage systems. The workflow engine switches between the storage systems to maintain continuity of operation in the event of a single storage system failure; on recovery, the storage systems are re-synchronized.

Workflow Engine Library

Workflow Engine is a class library developed using the C# programming language. A workflow can contain a sequence of service adaptors and sub-workflows (pre-configured workflows, e.g. Retry Fault Tolerance strategy for payment services). When executing a workflow, adaptors and sub-workflows invoke individual cloud services in a pre-defined sequence.

4.4 Monitoring Center

The function of the monitoring centre is to monitor the runtime performance of cloud services and to calculate QoS values that are used for optimizing applications. The Monitoring Centre provides three basic functions:

- **Recording log data:** This function collects service log data from enterprise applications. The *Log Collector* is invoked by service adaptors or by enterprise applications and records log data in the monitoring database. At the same time, alerts are generated that indicate fault conditions and departures from the expected QoS values.

- **QoS calculation:** This function calculates the response time and availability of cloud services using recorded log data. The resulting QoS values can be used for cloud service selection during the service identification phase.
- **Cloud service monitoring:** The availability and runtime performance of cloud services is compared to the expected QoS values as specified in the RFS.

The monitoring centre consists of *Monitoring Centre Database*, *Log Collector*, *QoS Analysis* and *Service Monitor* modules. The monitoring centre database is implemented using Microsoft SQL server and stores log records generated by service invocations. The log records include the service identifier (ServiceCode) and the application identifier (ApplicationID) of the enterprise application that executed the API call, service execution start (StartTime) and end times (EndTime), result of the call (i.e. success/failure) and error codes generated by the adaptor. Service adaptors record the runtime logs in the monitoring database using the log collector module. Whenever the log collector detects a service failure, the monitoring centre sends a notification to the relevant system administrators. The log data is used to generate hourly, daily and monthly reports of average availability and response time for individual cloud services. The QoS analysis module developed using C# programming language is deployed on a AWS EC2 server and configured to execute as a Window Service. The service monitor module is developed using ASP.Net MVC 5 and is deployed as a client tool for monitoring the cloud services. The service monitor module displays the runtime QoS information for individual cloud services and compares these values with the QoS values defined in the requirements specification.

5 Conclusions

We have argued that traditional systems development methods and tools are not adequate in the context of cloud services, and that a new approach that supports cloud service consumer lifecycle activities is required. In our earlier work, we have proposed a Service Consumer System Development Lifecycle (SC-SDLC) that focuses on the activities of cloud service consumers. In this paper, we describe implementation of the Service Consumer Framework (SCF) that supports design and runtime activities throughout the SC-SDLC phases. SCF is a research prototype intended to evaluate the feasibility of a relatively *light-weight* solution suitable for SMEs (Small and Medium size Enterprises) that are in the process of developing enterprise applications that consume externally provided cloud services. We have evaluated the implementation of several fault tolerant strategies (RFT, RBFT and DSFT) and found that the experimental results obtained using the SCF are consistent with theoretical predictions, indicating significant improvements in service availability when compared to invoking cloud services directly [30]. Both the SC-SDLC and SCF have been evaluated during the development of a Hospital Management application for Family Medical Practice (<https://www.vietnammedicalpractice.com/>), a leading international primary health care provider in Vietnam [26]. We have received positive feedback indicating that the SC-SDLC method guided developers throughout the project and SCF framework provided a suitable tool for recording information about cloud services and the various SC-

SDLC phases, leading to an improvement in overall productivity. Additionally, the cross-provider failover capability implemented using the workflow engine, and monitoring center features were regarded as having potential to significantly reduce outages and improve application availability. Areas of potential future improvement include the definition of guiding principles and documentation of best practices for each SC-SDLC phase.

References

1. Rehman, Z.-u., O.K. Hussain, and F.K. Hussain, *User-side cloud service management: State-of-the-art and future directions*. Journal of Network and Computer Applications, 2015. **55**: p. 108-122.
2. Arun, S., A. Chandrasekaran, and P. Prakash, *CSIS: Cloud Service Identification System*. International Journal of Electrical and Computer Engineering (IJECE), 2017. **7**(1): p. 513-520.
3. Ghamry, A.M., et al. *Towards a Public Cloud Services Registry*. in *International Conference on Web Information Systems Engineering*. 2017. Springer.
4. Hajlaoui, J.E., et al. *QoS Based Framework for Configurable IaaS Cloud Services Discovery*. in *Web Services (ICWS), 2017 IEEE International Conference on*. 2017. IEEE.
5. Rotem, R., A. Zelovich, and G. Friedrich, *Cloud services discovery and monitoring*, 2016, Google Patents.
6. Yang, K., et al., *Model-based service discovery—prototyping experience of an OSS scenario*. BT technology journal, 2006. **24**(2): p. 145-150.
7. Zisman, A., et al., *Proactive and reactive runtime service discovery: A framework and its evaluation*. IEEE Transactions on Software Engineering, 2013. **39**(7): p. 954-974.
8. Ciuffoletti, A., *Application level interface for a cloud monitoring service*. Computer Standards & Interfaces, 2016. **46**: p. 15-22.
9. Qu, L., et al. *Context-aware cloud service selection based on comparison and aggregation of user subjective assessment and objective performance assessment*. in *Web Services (ICWS), 2014 IEEE International Conference on*. 2014. IEEE.
10. Qu, L., Y. Wang, and M.A. Orgun. *Cloud service selection based on the aggregation of user feedback and quantitative performance assessment*. in *Services computing (scc), 2013 IEEE international conference on*. 2013. IEEE.
11. Montes, J., et al., *GMonE: A complete approach to cloud monitoring*. Future Generation Computer Systems, 2013. **29**(8): p. 2026-2040.
12. ProgrammableWeb. *ProgrammableWeb - API Directory*. 2018 Accessed on: 20.07.2018, Available from: <https://www.programmableweb.com/>.
13. Feuerlicht, G. and H.T. Tran. *Service Consumer Framework*. in *Proceedings of the 16th International Conference on Enterprise Information Systems-Volume 2*. 2014. SCITEPRESS-Science and Technology Publications, Lda.

14. Tran, H.T. and G. Feuerlicht, *Service development life cycle for hybrid cloud environments*. Journal of Software, 2016.
15. Joshi, K., et. al. *Integrated lifecycle of IT services in a cloud environment*. in *Proceedings of The Third International Conference on the Virtual Computing Initiative (ICVCI 2009)*, Research Triangle Park, NC. 2009.
16. Joshi, K.P., Y. Yesha, and T. Finin, *Automating Cloud Services Life Cycle through Semantic Technologies*. IEEE Transactions on Services Computing, 2014. **7**(1): p. 109-122.
17. Field, L., et al., *The emi registry: Discovering services in a federated world*. Journal of grid computing, 2014. **12**(1): p. 29-40.
18. Mahmood, Z. and S. Saeed, *Software engineering frameworks for the cloud computing paradigm*. 2013: Springer.
19. Thönes, J., *Microservices*. IEEE Software, 2015. **32**(1): p. 116-116.
20. Rimal, B.P., et al., *Architectural requirements for cloud computing systems: an enterprise cloud approach*. Journal of Grid Computing, 2011. **9**(1): p. 3-26.
21. Brogi, A., et al., *SeaClouds: a European project on seamless management of multi-cloud applications*. ACM SIGSOFT Software Engineering Notes, 2014. **39**(1): p. 1-4.
22. Brogi, A., et al. *SeaClouds: an open reference architecture for multi-cloud governance*. in *European Conference on Software Architecture*. 2016. Springer.
23. Kolb, S. and C. Röck, *Nucleus-Unified Deployment and Management for Platform as a Service*. 2016.
24. Feuerlicht, G. and H. Thai Tran. *Adapting service development life-cycle for cloud*. in *Proceedings of the 17th International Conference on Enterprise Information Systems-Volume 3*. 2015. SCITEPRESS-Science and Technology Publications, Lda.
25. Tran, H.T. and G. Feuerlicht, *Service Development Life Cycle for Hybrid Cloud Environments*. JSW, 2016. **11**(7): p. 704-711.
26. Tran, H.T., *A FRAMEWORK FOR MANAGEMENT OF CLOUD SERVICES*, 2017, University of Technology Sydney.
27. *SQL Server 2017 on Windows and Linux | Microsoft*. Accessed on: 20.07.2018, Available from: <https://www.microsoft.com/en-au/sql-server/sql-server-2017>.
28. Anderson, R. *ASP.NET MVC 5*. 2018 Accessed on: 20.07.2018, Available from: <https://docs.microsoft.com/en-us/aspnet/mvc/mvc5>.
29. *Windows Communication Foundation*. 2018 Accessed on: 20.07.2018, Available from: <https://docs.microsoft.com/en-us/dotnet/framework/wcf/>.
30. Tran, H.T. and G. Feuerlicht. *Improving reliability of cloud-based applications*. in *European Conference on Service-Oriented and Cloud Computing*. 2016. Springer.

Decentralized Billing and Subcontracting of Application Services for Cloud Environment Providers

Wolf Posdorfer¹, Julian Kalinowski¹, Heiko Bornholdt¹, and Winfried Lamersdorf¹

University of Hamburg, Department of Informatics,
Vogt-Kölln-Straße 30, 22527 Hamburg, Germany

Abstract. This paper proposes a decentralized billing and subcontracting system for regional cloud service providers. Based on blockchain technology, this system allows, on the one hand side, to collectively offer services in a distributed environment in a strict or ad-hoc federation and, on the other, to bill each user of such a services individually without a respective central service. In order to do so, it uses a blockchain-based transaction process which uses specialized tokens in order to enable a fair and secure distribution of requested cloud services. It maintains the ability to achieve consensus by validating the respective blockchain (part). In result, the proposed system is not bound to a specific technology, but rather open to any blockchain that allows arbitrary data or modeling of custom transactions.

Keywords: Blockchain · Cloud Computing · Cloud Environment Provider · Consensus · Decentralized Ledgers

1 Introduction

Enabled by the increasing need to offload work intensive or space hungry applications into cloud environments, a few major players have emerged to dominate the market. This oligarchy is not only dangerous for the end consumer but also greatly hinders a fair and competitive market for smaller regional providers [7]. A study shows that in 2017 four companies dominate the cloud market with a combined share of over 50% [16].

By imposing secret migration hindrances through inflexible APIs these major players are enforcing a vendor lock-in which negatively affects smaller providers, as a complete stack migration to their service becomes either unfeasible or simply impossible. Due to their higher market power they can essentially also dictate the service prices by which smaller providers have to abide to stay somewhat competitive.

The introduction of Bitcoin in 2008 triggered a new movement in decentralization. Blockchains enable consensus-based replication of data in an untrustworthy environment. Every participating node has the same identical view of all the

transactions and their respective order. The underlying database is fully replicated and provides a high reliability. Even though the blockchain was initially created to serve the single purpose of being a "cryptocurrency" the technology can be used for many other applications and business processes.

We propose a scenario in which a multitude of smaller cloud environment providers can form a federation. Allowing them to bundle their resources into one virtual cloud provider. By utilizing the blockchain technology we can achieve a verifiable billing and subcontracting system. This enables all providers to act as equals and provides fair distribution and payment for the requested cloud services. The proposed process is generalizable to other use cases whenever they have a similar system composition and the process can be divided into sub-steps. The following use case will outline the process in the billing and subcontracting of cloud services.

Our approach differs from [14] in that it is not locked into Ethereum-VM compatible blockchains relying solely on smart contracts for application logic. By defining standard transaction types, which can be run on any blockchain, we do not impose technological restrictions. This also mitigates the requirements of Proof-of-Work, allowing for shorter block times and less energy consumption. Also in contrast to [18] our approach does not require a registry of current provider prices. Computing hours are not distributed in an auction style system where cheaper services are always favored, thus no price war is created between smaller providers.

2 Use Case

This section will provide a use case for a decentralized billing platform to illustrate the benefits it can provide. With a decentralized subcontracting and billing platform of application services in cloud environments multiple smaller cloud computing providers can form a federation to act as a single provider, thus allowing them to be more competitive in today's market. Instead of offering their services as single entities they can offer a combined service that is transparent for the end user.

Figure 1 shows an exemplary composition of a customer requesting 90 hours of service and three providers (Provider 1-3), with two subcontractors (Sub3a & Sub3b).

2.1 Cloud Service Billing

One of the necessities of a decentralized platform where multiple providers share an incoming workload is the correct billing of the performed computation hours by each participant. In this scenario we assume that a customer has paid for a service in advance and that the service will run as long as its being paid for. Depending on the configuration the fiat money will be evenly distributed between all the providers connected to the federation. In Figure 1 the customer requests 90h, which is evenly distributed as 30h for each provider.

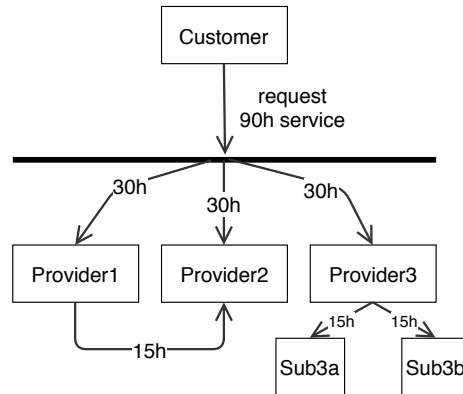


Fig. 1. Service Billing and Subcontracting

2.2 Cloud Service Subcontracting

On the assumption that all connected cloud service providers have limited resources it is quite possible that an even distribution of workload between them will lead to bottlenecks. By allowing providers to offload a complete or partial workload to another provider these bottlenecks can be overcome. It even allows a single provider to act as a proxy to subcontractors, e.g. regional providers. Figure 1 shows how Provider3 subcontracts to two additional providers Sub3a and Sub3b.

3 Blockchain

A blockchain is a decentralized data structure, whose internal consistency is being maintained by reaching consensus over an application state in a network. The data itself is fully replicated and kept in synchrony over every participating node [2].

To change the state of the blockchain a transaction has to be submitted. Each transaction is bundled into a block, which will be chained together by calculating a hash over the transactions and a hash pointing to the previous block. Thus effectively chaining each block to its predecessor and creating a definite order. Figure 2 depicts an exemplary blockchain datastructure showing three blocks, their linking via the predecessors hash (Prev_Hash) and the root hash of a merkle-tree (Tx_Root) containing transactions (Tx0 - Tx3).

3.1 Transaction

The fundamental data structures of the blockchain are transactions and blocks. The transaction is the smallest data unit being processed and capable of changing

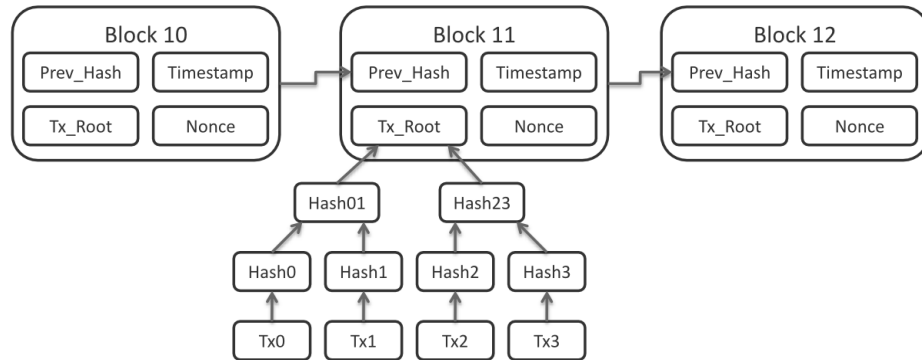


Fig. 2. Simplified Blockchain Datastructure

the overall state of the blockchain. Network participants can create them and propagate them to other nodes through a peer-to-peer network. Depending on the blockchain technology used transactions can contain different arbitrary data. In Bitcoin [13] and other cryptocurrencies they contain info about the sender, receiver and quantity of currency (simplified). Even function calls are possible, by using smart contracts or chain code as used in Ethereum [19]. But blockchains are not limited to the usage of currency or smart contracts. They can also be used for several other applications like supply chain management, voting or ballots, crowd funding or to store data in general.

3.2 Block

After a transaction has been published and propagated through the network it will be bundled into a block. The transactions will be stored and hashed with a suited algorithm and data structure, usually variants of the SHA-algorithm and Merkle-Trees. Every block consist of a header containing the predecessors blocks hash, its own transactions hash (root hash) and depending on the used technology other information like timestamps, version, block height, target, nonce or others.

By using the preceding block hash they are effectively chained together all the way back to the genesis block. The chaining of blocks creates a traceability over all transactions and thus also the overall state and state changes of the blockchain.

3.3 Consensus

The key element behind every blockchain is its consensus algorithm. Through it the blockchain ensures that the majority of nodes has the same valid shared state in the long-term. Depending on the used algorithm the majority of nodes necessary for a valid block typically lies at 51% in chain-based algorithms or $+\frac{2}{3}$ in Byzantine Fault Tolerant (BFT)-based algorithms.

Chain-based / Append-Based Algorithms In Bitcoin and similar technologies the consensus algorithm is referred to as *Proof-of-Work* (PoW) [4, 13, 19]. Its goal is to provide trust through a cryptographic challenge. The challenge consists of finding a *Nonce* so that the resulting hash of Nonce and Root-Hash meets a certain target or difficulty in the form of amount of leading zeros. Every node interested in solving the challenge by brute-forcing nonces is called a *Miner*, while the process itself is referred to as *Mining*. Its rather simple to verify the validity of the produced block by other nodes. The difficulty ensures that very rarely two different blocks are propagated through the network at the same time. Also it ensures that it becomes harder and harder to forge previous blocks by recalculating hashes with different nonces.

To create a block a miner selects a set of transactions calculates the root hash and starts brute-forcing nonces until he finds one that meets the current difficulty. The difficulty is automatically adjusted by the network in order to keep the median time between two blocks in roughly the same timespan. With the increased participation in the cryptographic challenge the amount of computing power also increases as a direct result of the PoW algorithm and self-adjusting difficulty.

In contrast to PoW the *Proof-of-Stake* (PoS) algorithms try to mitigate the waste of resources [6, 9, 10]. In PoS miners can stake their own coins (or other values) in order to create blocks more easily. Owning and staking a higher amount of coins results in a higher likelihood of creating a block. Other algorithms impose additional requirements on the coins, like the coin-age, to limit the usage of massive amount of coins.

BFT-based / Propose-Based Algorithms BFT-based algorithms try to solve the consensus problem by using algorithms that solve the byzantine generals problem [17]. Usually they are loosely based on the PBFT-algorithm [5] and 2-Phase-Commit-algorithms [12]. In BFT-based PoS algorithms there is a certain set of nodes called Validators, who are responsible for consensus. Validators each take turns in proposing new blocks. This ensures that only one block for a given height is valid. Unlike in chain-based algorithms where more than one block can compete to be the next block. This also means that there can be no forks in the chain. While in chain-based algorithms any number of nodes can choose to not partake in the block-finding process, in BFT style algorithms a minimum set of more than $\frac{2}{3}$ (or $+\frac{2}{3}$) of validators need to be online at any given time. If there are less than $+\frac{2}{3}$ the proposed block will not reach consensus.

3.4 Process

Every blockchain independent of its underlying consensus algorithm follows the same sequence of steps until a new block which is accepted by other nodes is appended. Every blockchain participant is running the same client, which either contains the application layer (like Bitcoin) or an API for a custom application (like Hyperledger [1]). Every node is linked to a certain amount of other nodes via a peer-to-peer network which allows the distribution of messages.

Once the application layer has created a transaction, which contains data depending on the use-case/technology, it will be passed to the validation component. The transaction validation depends highly on the use case, e.g. checking if an account balance is sufficient. If it is valid the transaction will be placed into the *mempool* and broadcasted to other peers, who repeat this process. Ideally this ensures that every network participant has the same valid transactions in its mempool.

Once a node has qualified for creating a block (or proposing) it will select a number of transactions from the mempool and bundle them into a block. Depending on certain criteria like transaction age or transaction fees the node can choose which transactions to include. After forming the block the node will distribute it to its peers. Upon receiving a block the node has to perform its own validity checks on the block and transactions within the block, as to not append an incorrect state to its own blockchain.

4 Problem definition

A classic approach for managing cross-company payments, costs and distribution of revenue would be to establish and make use of a trusted third party. This trusted third party keeps track of everything that is relevant to the system, such as commissions, orders, computing hours and billable hours. This implies that the third party will get to know details about business relationships between the companies and, of course, the account balances.

Instead of trusting a third party with this data, it is desirable to keep as much of the data private as possible and to instead distribute the trust amongst all parties.

Since in the sketched scenarios, we will have multiple participants, potentially distrusting each other (dishonesty can lead to personal advantage), a blockchain solution seems appropriate [20]. It provides data integrity with multiple untrusted writers without a trusted third party. Additionally, blockchain transactions can be designed in a way that they support required business processes in a network of equal partners, where nobody is in control, and yet everybody can verify the correctness of a process.

The concept of blockchain was created with transparency in mind, which is why all stored data is available for everyone to validate [13]. The validation in turn provides the necessary security for a distributed database with multiple participants who potentially distrust each other. Each participant is given the opportunity to vote for his own sense of correctness of a given transaction in the blockchain and to do so, he must have access to the data. This is why validation is a critical part of a blockchain and it is tightly coupled with transparency.

In the given scenario however, full transparency might not be a valid option as data are trade secrets and should not be made publicly available. While a blockchain can be private, not offering public access for anyone who is interested, transparency of sensible information remains a problem: At least all authorized participants would be able to read all data, which is bad by itself, especially

when direct competitors are involved in the same system. On the other hand transparency is highly necessary to ensure a working validation and checking for transaction correctness to guarantee reaching consensus.

5 Approach

The general idea is that, instead of transferring fiat money, trading happens on the blockchain using tokens. This enables a fast, reliable and secure way for exchanging a value representation without having to pay fees, enabling participants to reflect every single transfer of value in the blockchain. Additionally, the blockchain will provide a decentralized way of clearing, such that the tokens can be exchanged for fiat money after a given period. This period may be inspired by the underlying business process and common for all participants. It may also depend on individual preferences and should not be restricted, however.

5.1 Billing & Subcontracting Scenario

In this scenario, there are two groups of participants, service *providers* and *customers*. Each provider may maintain business relationships with other providers, although there will not always be a direct connection between any two of them in this graph of relationships. Instead, multiple smaller strongly connected sub-graphs corresponding to individual groups of co-operation are possible.

Figure 3 shows a sample graph of this scenario, where C_i are customers and P_i are providers. The edges are labeled with the amount of tokens that are being sent. Dotted edges represent distribution of a previous token, performed by a provider. Marked in *red* for each provider is the sum of tokens after all the transactions are performed, e.g. the net sum for each provider.

The customers (or a central proxy) are the only entities that may issue new *tokens*, just like a mint would do with fiat money. Tokens (representing money or computing hours) are then given to one or more providers, who can in turn split them and pass them on to other providers.

In this scenario, tokens, once issued, can be distributed and passed from provider to provider. However, they can never vanish and a provider can only pass on a token he owns and hasn't already spent otherwise.

Now, after all tokens have been transferred according to the underlying business process, each provider knows his token balance at any time, which corresponds e.g., to computing hours.

In order to exchange his tokens for fiat money, he must be able to generate a proof showing anyone with access to the blockchain data that he is the rightful owner of his tokens.

5.2 Transaction Types

Based on the previously introduced use case at least the following three transaction types are necessary to model the business processes. Every Transaction can

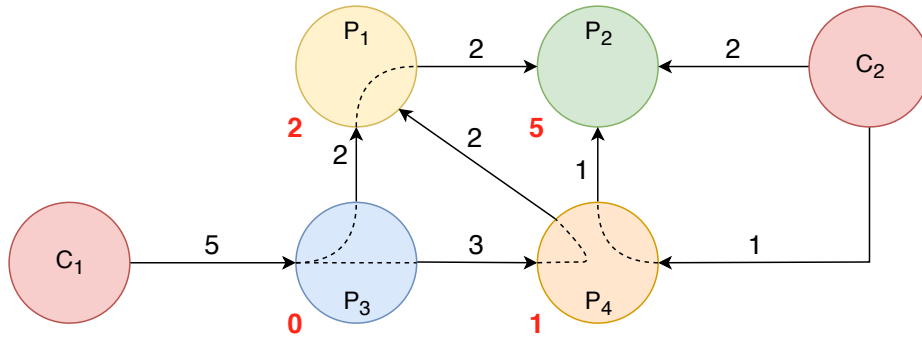


Fig. 3. Billing scenario

contain multiple input tokens and multiple output tokens. Input tokens must be owned by the same participant, while outputs can be assigned to different parties. The following transaction types are not unique to a special blockchain, but can be implemented on any technology that allows custom transactions, like: Corda [3], Ethereum [19], Hyperledger [1], Tendermint [11] and others. Bitcoin and its descendants are unsuited because of their strict transaction formats and limited transaction payload size.

Initialization (INIT) is used to publish newly created tokens into the system. Only customers can create new tokens, backing them against fiat currency. This transaction type does not require any input tokens as the customer is actively *minting* them and also contains only one output.

Distribution (DIST) allows a single party to transfer or split their tokens as required by the business process. It can be used to sell parts of their computing hours to others.

Payout (POUT) transaction is used to exchange tokens for fiat money with a customer. It contains multiple input tokens from the same owner and a single output token towards the customer.

Figure 4 shows an exemplary transaction flow. The customer deposits fiat money and converts it to 100 Tokens (T). The $100T$ are then issued to Provider1 using the *INIT* transaction type. After this transactions has been validated and finalized in a block, every participant can now confirm that Provider1 owns $100T$. When Provider1 is unable to provide the $100T$ worth of computing hours he can offload it to another provider. Provider1 distributes his tokens using the *DIST* transaction to split his balance between himself and Provider2. Again after validation and finalization in a block, everyone can confirm that Provider1 and Provider2 both own $50T$. Once Provider2 wants to convert his tokens back into fiat money he issues a *POUT* transaction, reassigning his $50T$ back to the customer. Upon receiving the tokens the customer will issue the respective amount of fiat money to Provider2 off-chain.

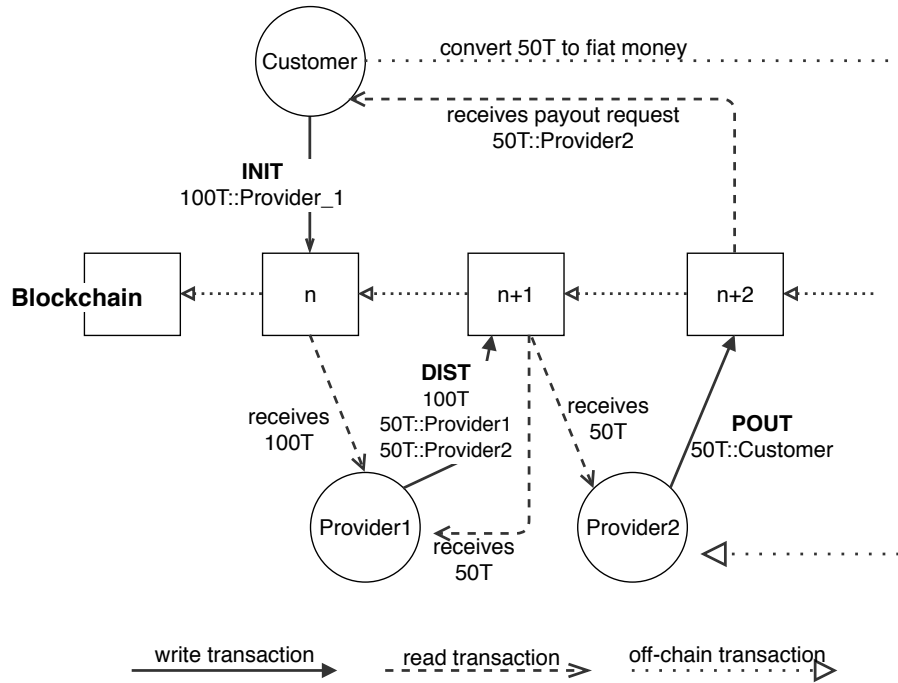


Fig. 4. Example transaction flow

A minor impracticality in this scenario is the huge reliance on trust. In order to mitigate wrongdoings by customers a proxy-service will have to be placed in between to deposit the fiat money and issue the corresponding tokens respectively. This ensures that the customer has actually deposited fiat money and the providers can later retrieve it.

5.3 Validation and Transparency

As stated in Section 4, validation of transactions is performed by any active participant. This requires full access to the transaction data, which, in our scenario, means every participant can see any tokens being sent between customers (or proxy) and service providers. But as previously stated full transparency is not desired for at least the following three aspects:

- Amount of tokens in possession by a single provider should not be revealed.
- Offloading relations should be hidden from other providers not participating in the corresponding customer’s job.
- General anonymity is also not ensured as every token assignment must be directed to a specific provider.

A reasonable validation rule, executed by every participant of the system, would be: *"For any tokens that are distributed and payed-out, is there a valid incoming token transaction for this provider?"* In a fully transparent blockchain system, this would be trivial to check as everyone has access to the balances and can take a look at past transactions. However, in a system with private balances and transaction data, there may be no validation possible at all (thus rendering the blockchain approach pointless). This implies that designing a privacy-protective solution may come with a trade-off between privacy and validation.

The natural approach in favor of privacy would be full encryption of the transaction data. All data would be private and visible for the sender and recipient only. Obviously, nobody may validate this data except for the sender and receiver, respectively.

6 Future Work and Conclusion

The proposed blockchain based solution shows a generalizable method to provide a decentralized billing and subcontracting process for cloud environment providers. It can be adapted to a multitude of other business processes that share the same characteristics. Whenever a process is started from a single entity and can be divided into measurable subparts the proposed solution is a suitable candidate.

Another exemplary use case can be the execution of distributed workflows. Where an orchestrator (customer) distributes actions of the workflow to services (provider), which in turn can divide the actions into subactions and redistribute them to other services all in a traceable and verifiable manner on the blockchain.

As transaction transparency and transaction validation are closely reliant on each other new methods for privacy protection must be established. As full data encryption is not feasible, because it breaks validation, other steps must be taken when specific use cases highly require data protection.

One of the measures to take to ensure validation is the usage of homomorphic encryption [8]. Homomorphic encryption allows for token values to be encrypted while still maintaining the ability to construct sums and verify input and output values.

When introducing encryption the encrypted values or token sums must also remain unforgeable. Thus a binding value must be chosen in the form of a verifiable secret. A commitment solves these issues as it firstly hides the input value and secondly is also binding [15].

References

1. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference. p. 30. ACM (2018)

2. Antonopoulos, A.M.: Mastering Bitcoin: unlocking digital cryptocurrencies. O'Reilly Media, Inc. (2014)
3. Brown, R.G., Carlyle, J., Grigg, I., Hearn, M.: Corda: An introduction. R3 CEV, August (2016)
4. Buterin, V., et al.: A next-generation smart contract and decentralized application platform. white paper (2014)
5. Castro, M., Liskov, B., et al.: Practical byzantine fault tolerance. In: OSDI. vol. 99, pp. 173–186 (1999)
6. David, B.M., Gazi, P., Kiayias, A., Russell, A.: Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake protocol. IACR Cryptology ePrint Archive **2017**, 573 (2017)
7. Feng, Y., Li, B., Li, B.: Price competition in an oligopoly market with multiple iaas cloud providers. IEEE Transactions on Computers **63**(1), 59–73 (2014)
8. Gentry, C., Boneh, D.: A fully homomorphic encryption scheme, vol. 20. Stanford University Stanford (2009)
9. Jain, A., Arora, S., Shukla, Y., Patil, T., Sawant-Patil, S.: Proof of stake with casper the friendly finality gadget protocol for fair validation consensus in ethereum (2018)
10. King, S., Nadal, S.: Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. self-published paper, August **19** (2012)
11. Kwon, J.: Tendermint: Consensus without mining. Draft v. 0.6, fall (2014)
12. Lampson, B., Sturgis, H.E.: Crash recovery in a distributed data storage system (1979)
13. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
14. Neidhardt, N., Köhler, C., Nüttgens, M.: Cloud service billing and service level agreement monitoring based on blockchain. In: EMISA. pp. 65–69 (2018)
15. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Crypto. vol. 91, pp. 129–140. Springer (1991)
16. Synergy Research Group: The leading cloud providers continue to run away with the market (2017), <https://www.srgresearch.com/articles/leading-cloud-providers-continue-run-away-market>
17. Vukolić, M.: The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In: International Workshop on Open Problems in Network Security. pp. 112–125. Springer (2015)
18. Wang, H., Shi, P., Zhang, Y.: Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization. In: Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on. pp. 1846–1855. IEEE (2017)
19. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper **151**, 1–39 (2018)
20. Wst, K., Gervais, A.: Do you need a blockchain? Cryptology ePrint Archive, Report 2017/375 (2017), <https://eprint.iacr.org/2017/375>

May Contain Nuts: The Case for API Labels

Cesare Pautasso¹ and Erik Wilde²

¹ Software Institute, Faculty of Informatics, USI, Lugano, Switzerland

² CA Technologies, Zürich, Switzerland

Abstract. As APIs proliferate, managing the constantly growing and evolving API landscapes inside and across organizations becomes a challenge. Part of the management challenge is for APIs to be able to describe themselves, so that users and tooling can use descriptions for finding and filtering APIs. A standardized labeling scheme can help to cover some of the cases where API self-description allows API landscapes to become more usable and scalable. In this paper we present the vision for standardized API labels, which summarize and represent critical aspects of APIs. These aspect allow consumers to more easily become aware of the kind of dependency they are going to establish with the service provider when choosing to use them. API labels not only summarize critical coupling factors, but also can include claims that require to be validated by trusted third parties.

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

APIs are the only visible parts of services in API-based service landscapes. The technical interface aspect of APIs has been widely discussed with description languages such as WSDL, RAML, and Swagger/OpenAPI. The non-functional aspects are harder to formalize (e.g., see the survey by García et al. [8]) but can also benefit from a framework in which information can be represented and used.

The idea of “API Labels” is equivalent to that of standardized labeling systems in other product spaces, for example for food, for device energy consumption, or for movie/games audience ratings. In these scenarios, labels enable consumers to understand a few key (and often safety-critical) aspects of the product. This framework is not intended to be a complete and exhaustive description of the product. Instead, it focuses on areas that are important and helpful to make an initial product selection. The assumption is that the information found on the label can be trusted, so that consumers can make decisions based on labels which are correct and do not contain fraudulent information.

In the API space, numerous standards and best practices have evolved how APIs can be formally described for machine processing and/or documented for human consumption [14] (e.g., WSDL [4], WADL [9], RESTdesc[24], hRESTS [12], RADL [19], RAML, Swagger/OpenAPI [22], SLA★ [10], RSLA [21], SLAC [23]

just to mention a few). However, there still is some uncertainty how to best combine and summarize these, and how to use them so that API description, documentation, and labeling can be combined. This paper proposes the API Labels Framework (AFL) to introduce API labels as a synthesis of existing API descriptions combined with additional metadata which can help customers assess several practical qualities of APIs and their providers and thus be useful to reduce the effort required to determine whether an API can be worthy of consideration.

The main motivation for labeling APIs is probably not so much about a way to enable providers to put marketing labels on their APIs nor is it a way to summarize information that is already present in existing formal API descriptions. Instead, it is about providing assurances for API consumers about crucial characteristics of the service behind the API that may not be visible on its surface.

The rest of this paper is structured as follows. In Sec. 2 we present general background on labeling and related work which has inspired the current paper. In Sec. 3 we apply the concept of labeling to APIs and discuss how to use OpenAPI Link Objects and Home Documents to make API labels easy to find. We discuss the issue of how to establish trust for API labels in Sec. 4 and then introduce different label types in Sec. 5. The following Sec. 6 provides a non-exhaustive set of label type examples. The problem of discovering labels and ensuring that they can evolve over time are identified in Sec. 7.2. Finally we draw some conclusions in Sec. 8 and outline possible directions for future work in Sec. 9.

2 Background and Related Work

Labeling helps to identify, describe, assess and promote products [13]. Branding and labeling contribute to differentiate competing products by assuring the consumer of a guaranteed level of quality or by restoring consumer’s confidence after some negative publicity leading to a loss of reputation. More specifically, food labeling has also been used to educate consumers on diet and health issues [5]. Labeling can thus be used as a marketing tool [1] by providers or as a provider selection tool by consumers [2].

This work is inspired by previous work on designing simplified privacy labels of Web sites [11] based on the now discontinued P3P standard [7]. It shares similar goals to provide a combined overview over a number of “API Facts”. However, one important difference is that P3P was a single-purpose specification intended to standardize everything required for embedding privacy labels. It thus had fixed methods to locate privacy policies (four variations of discovering the policy resource), fixed ways how those were represented (using an XML-based vocabulary), and a fixed set of acceptable values (also encoded into the XML vocabulary) to be used in these policies.

The work presented in this paper is bigger in scope, and on the framework level. As such, we do not authoritatively prescribe any of the aspects that P3P was defining. Instead, we are assuming that with organizations and user groups

using API labels, certain patterns will emerge, and will be used inside these communities. We can easily envision a future where our framework is used as a foundation to define a more concrete set of requirements, but this is out of scope for this paper, and most likely would benefit substantially from initial usage and feedback of the API label framework presented here.

3 Labeling APIs

The idea of API labels is that they apply not just to individual resources, but to a *complete API*. Many APIs will provide access to a large set of resources. It depends on the API style how APIs and individual resources relate [18]. In the most popular styles for APIs today, which are HTTP-based, the API is established as a set of resources with distinct URI identities, meaning that the API is a set of (potentially many) resources. One exception to this are RPC-oriented API styles (such as the ones using SOAP, grpc or GraphQL) which “tunnel” all API interactions through a single “API endpoint”. In that latter case, there is no such thing as a “set of HTTP-oriented resources establishing the API”, but since we are mostly concerned with today’s popular HTTP-based styles, the question of the scope of API labels remains relevant.

Applications consuming APIs are coupled to them, and the choice of API to be consumed introduces critical dependencies for consumers [17]. Consumers need to be made aware about non-functional aspects, concerning the short-term availability and long-term evolution of API resources [15]. Likewise, when a resource is made available by a different API, different terms of service may apply to its usage.

From the consumer point of view, the concept of an “API boundary” can seem arbitrary or irrelevant, or both. API consumers most importantly want to implement applications. To do so, they need to discover, select and invoke one or more APIs. However, even when from the strict application logic point of view the “boundary” between APIs may not matter (applications will simply traverse resources either driven by application logic or by hypermedia links), it still may be relevant for non-functional aspects, such as when each API resource is made available by a different provider and therefore different terms of service apply to its usage.

Generally speaking, the Web model is that applications use various resources to accomplish their goals, and these resources often will be provided by more than one API. In this case the question is how it is possible to get the API labels for every resource, if applications want to do so. What is the scope of API labels, and how is it possible, starting from *any* resource of an API, to find its API labels? And how can an application know when traversing resources that it traverses an “API boundary”? The Web (and HTTP-based URIs) has no built-in notion to indicate “API boundaries”, so the question is how to establish such a model.

It seems wasteful to always include all API label information in all resources, given that in many cases, applications will not need this information and thus it

would make API responses unnecessarily large. However, there are approaches how this can be done in more efficient ways, and currently there are two solutions available (OpenAPI Link Objects and Home Documents). It is important to keep in mind that it is up to an API designer to decide if and how they will use these techniques to make labels easy to find.

3.1 OpenAPI Link Objects

The API description language *OpenAPI* (formerly known as *Swagger*) has added the concept of a *link object* with its first major release under the new name, version 3.0. Essentially, link objects are links that are defined in the OpenAPI description, and then can be considered to be applicable to specific resources of the API. In essence, this creates a shortcut mechanism where these links are factored out from actual API responses, and instead become part of the API description.

It is important to keep in mind that because of this design, the actual links in the OpenAPI link object never show up in the API itself; instead they are only part of the OpenAPI description. This design allows OpenAPI consumers to use these links without producing any runtime overhead, but it makes these links “invisible” for anybody not using the OpenAPI description and interpreting its link objects.

This design of OpenAPI thus can be seen as effective optimization, because it creates no runtime overhead. On the other hand, it limits self-descriptiveness and introduces substantial coupling by making the links in link objects exclusively visible to clients knowing and using the OpenAPI description.

For this reason, we believe that in environments where this coupling has been introduced already, OpenAPI link objects may be a good solution. This can be any environment where the assumption is that API consumers always know the OpenAPI descriptions of the APIs they are consuming. This may be a decision that is made in certain organizations or communities, but cannot be considered a design that is used in unconstrained API landscape.

In unconstrained API landscapes, it seems that the coupling introduced by making the knowledge and usage of all OpenAPI descriptions mandatory is substantial, and may be counterproductive to the self-describing and loosely coupled consumption of APIs. If the design goal is to focus on self-description and loose coupling, then OpenAPI link objects probably are not the best choice, and instead the approach of *home documents* may be the better one.

3.2 Home Documents

An alternative model to that of OpenAPI is established by the mechanism of *home documents* [16]. The idea of home documents is that there is a “general starting point” for an API. This starting point can provide a variety of information about the API, including information about its API labels. The home document then can be linked to from API resources, and there is a specific **home** link relation that is established as part of the home document model.

Using this model, all resources of an API can provide *one* additional link, which is to the API home document. The home document then becomes the starting point for accessing any information about the API, including an API's labels. This model means that there is an overhead of one link per resource. However, given modern mechanisms such as HTTP/2.0 header compression, it seems that this overhead is acceptable in the majority of cases, even if that link is not so much a functional part of the API itself, but instead provides access to metadata about the API.

One of the advantages of the idea of home documents and providing home links for resources is that this makes the API (or rather its resources) truly self-describing: Consumers do not need any additional information to find and use the information about an API's home document.

One downside to this model is that home documents are not yet a stable standard used across many APIs. The draft has been around for a while and has evolved over time, but it is not guaranteed that it will become a stable standard. One other hand, since this work is rooted in general Web architecture, even without the specification being a stable standard already using it is acceptable, and in fact this is how many IETF standards are conceived: drafts are proposed, already adopted by some, and the eventual standard then is informed by gathering feedback from those who already have gained experience with it.

4 Trusting API Descriptions and Documentations

API labels provide a human-readable format to summarize API descriptions including hyperlinks to relevant documentation and specifications. API labels are also meant to be machine processable to provide the basis for automated support for API landscape visualization and filtering capabilities.

One example for this are the link relation types for Web services [26]. These could be readily used as API labels (if they are made discoverable through the general API label mechanism). Some of the resources are likely just human-readable (for example API documentation provided as PDF), while other resources might be machine-readable and to some extent even machine-understandable (for example API description provided as OpenAPI which can be used by testing and documentation generation tools).

API labels are not meant to provide a complete specification of APIs and replace existing languages and service discovery tools. Instead, they are designed to include information that is currently not found in API descriptions as written by service providers, because this information may include claims that need to be verified by trusted third parties. Additionally, the summary described in the label can lead to more detailed original sources that can be used to confirm the validity of the summarized information.

While it is in a provider's best interest to provide a correct representation of its APIs functional characteristics (operation structure, data representation formats, suggested interaction conversations) so that clients may easily consume the API appropriately, questionable providers may be tempted to misrepresent

some of the Quality of Service levels they may be capable of guaranteeing. Hence labeling APIs could provide the necessary means to certify and validate the provided API metadata information complementing other means to establish and assess the reputation of the API provider [3]. This is a rather challenging task that would require to deal with a number of non-trivial issues.

For example, how would consumers establish trust with a given API label certification authority? Is one centralized authority enough or should there be multiple ones taking into advantage the decentralized nature of the Web [6]? If multiple parties can certify the same API, how should consumers deal with conflicting labels? How to ensure labels can be certified in an economically sustainable way (are consumers willing to pay to get verified labels?) without leading to corruption (providers are willing to pay to get positive labels)? How would the authority actually verify the QoS claims of the provider? How to avoid that a provider obtains good results when undergoing a certification benchmark but poor performance during normal operations when servicing ordinary customer requests? How to ensure API labels are not tampered with? Should labels be signed by reference or by value?

While it is out of scope of this paper to deal with all of these issues, we believe some form of delegation where APIs reference labels via links to label resources hosted by third parties will be one of the key mechanisms to enable trust into certified API labels. This way, even if the label value itself is not provided by the API, but by using the delegation mechanism, we could still make it discoverable through the API.

5 Label Types

In order to be understandable, labels must follow a framework of well-defined types that can be “read” as API labels. Some of these may already exist as evolving or existing standards. The link relations for Web services discussed in the previous section can be considered potential API labels that are defined in an evolving standard. An example for an existing standard is the `license` link relation defined in RFC 4946 [20], which is meant to convey the license attached to resources made available through a service.

A label type identifies the kind of label information that is represented by attaching a label of this type. In principle, there are three different ways of how label types can communicate label information to consumers:

- By Value: If the label is simply an identifier, then the meaning of the label is communicated by the label value itself. The question then is what the permissible value space is (i.e., which values can be used to safely communicate a well-defined meaning between label creators and label readers). The value space can be fixed and defined by enumerating the values associated with the label type, or it can be defined in a way so that it can evolve. This second style of managing an evolving value space often is implemented through registries [25], which effectively decouples the definition of the label type and the definition of its value space.

- **By Format:** If the label is intended to communicate its meaning by reference, then it will link to a resource that represents the label’s meaning. It is possible for label types to require that the format is always the same, and must be used when using that label. This is what P3P (the example mentioned earlier) did, by defining and requiring that P3P policies always must be represented by the defined format. This approach allows to build automation that can validate and interpret labels, by depending on the fact that there is one format that must be used for a given label type.

- **By Link:** It is also possible to not require the format being used. This is the most webby and open-ended approach, where a label links to a resource representing the label’s value, but the link does not pre-determine the format of the linked resource. This approach has the advantage that label value representations can evolve and new ones can be added when required, but it has the disadvantage that there is no a priori interoperability of label producers and label consumers.

Returning to the examples given above, it becomes obvious that the existing mechanisms discussed so far that could be considered to be used as API labels already use different approaches from this spectrum. The link relation for licenses [20] is based on the assumption that a license is identified by value, thus requiring licenses to be identified by shared URI identifiers. P3P [7] defines its own format that has to be used for representing P3P labels. The link relations for Web services [26] identify information by link, and do not constrain the format that has to be used with those link relations.

6 API Label Examples

In this section we collect a preliminary list of API label types and values, characterizing several technical and non-technical concepts [27] which are meant to assist consumers during their API selection process. We have compiled this list based on the relevant literature, our experience, including feedback from our industry contacts.

- *Invocation Style:* This label defines on a technical level which kind style is required for clients to invoke the API. We distinguish between Synchronous RPC, Synchronous Callbacks, Asynchronous Events/Messages, REST, and Streaming.

- *Protocol Interoperability:* Which are the interaction protocols supported by the API? Which versions of the protocols? Examples values: SOAP, HTTP, GraphQL

- *Privacy:* Where is the data managed by the API stored? While clients do not care whether their data is stored in SQL or XML, they do worry whether their data is located in a different country and thus subject to different regulations.

- *Service Level Agreement:* Does an SLA explicitly exist? If it does: how is it enforced? are there penalties for violations? can it be negotiated? This helps to roughly distinguish between APIs without SLAs from APIs having an explicitly (formally or informally) defined SLA, which can be further annotated to highlight whether service providers make serious efforts to stand behind their

promises and whether they are willing to adapt to client needs by negotiating the terms of the agreement with them as opposed to offering a number of predefined usage plans.

- *Pricing*: Also related to SLA, clients want to know: whether there a free price plan? Can the API paid price plans be considered as cheap, reasonable, or expensive? This label needs to be computed based on the client expectations or by comparing with similar APIs.

- *Availability Track Record*: Does the API provider explicitly promises high availability? How well does the promise (e.g., “five nines” or 99.999%) matches the reality? Is the API provider’s availability improving or getting worse? Additionally, clients need to know how to set their timeouts before giving up and determining that the API is no longer available. The Availability Track Record should label APIs for which such information is explicitly found in the corresponding SLA.

- *Maturity/Stability*: The Maturity label should provide a metric to determine whether the API has reached flying altitude and can be considered as mature enough, i.e., it is likely to be feature complete and stable during the entire lifecycle of clients consuming it. This can be inferred from versioning metadata, or some kind of metric summarizing the API version history (e.g., the number of changes over time, or how many alternative versions of the same API are supported in parallel by the provider). Conversely, if APIs are not yet mature and unstable, clients would benefit from knowing how much time they have to react to breaking API changes. Different providers may allow different amounts of time between announcing changes and carrying them out. In a similar way, as APIs eventually disappear, does the provider support some notion of *sunset* metadata? Are API features first deprecated and eventually retired, or does the API provider simply remove features without any warning?

- *Popularity*: How many clients are using the API? Is this the mostly used API within the ecosystem/architecture? is it in the top 10 APIs based on daily traffic? or only very few clients rarely invoke it?

- *Alternative Providers*: Are there alternative and competing providers for the API? or there exists only one monopolistic provider? How easy is it to replace the service provider of the API? How easy is it to find a replacement API within minimal differences from the current one?

Additional label types describing energy consumption, sustainability, quality management (e.g. ISO 9001 compliance) or trust certificates are possible.

7 A Recipe for API Labels

As mentioned already, the exact way of how to implement labels is not yet standardized. In this paper, we discuss the parts that need to be in place to use API labels, but we do not prescribe one single correct way. In order to summarize these parts, and to give organizations looking at using API labels a useful starting point, we are summarizing the required parts in an “API label landscape”. We also recommend specific ways of solving these individual issues. In particular,

Section 7.1 provides methods to make labels findable, and Section 7.2 provides methods to manage the types and the values of those findable labels so that the set of labels used in an API landscape can organically grow over time.

7.1 Findable Labels

In order for API labels to be usable and useful, they must be findable. One possibility is to manage them separate from APIs themselves, but this approach is likely to let APIs and their labels go out of sync easily. A more robust approach is to make API labels parts of APIs themselves, which allows labels to be managed and updated by the APIs themselves, and also allows labels to be found and accessed by those that have access to these APIs.

Using such an approach, making API labels findable amounts to allowing them to be accessed through the API. For this to be consistent across APIs, there need to be conventions that are used across APIs to find and access labels. What these conventions look like, depends on the style and technology of APIs. For HTTP APIs that are based on the resource-oriented or the hypermedia style of APIs this amount to providing resources that represent label information.

In terms of currently available practices, using home documents as described in Section 3.2 works well, if it is acceptable as a general API guideline to require APIs to provide home documents. If it is, labels still need to be made discoverable from that home document. We are suggesting to represent labels in a way that represents a set of labels, and that has the ability to “delegate” label representation to third parties, so that that scenarios like the ones discussed in Section 4 can be implemented.

7.2 Extensible Label Sets

Once there is a defined way how labels can be found for APIs and, as suggested above, through the APIs themselves, then the next question is what types of labels can be found (Section 6 suggests a starting set of label types). It is likely that the set of label types is going to evolve over time, so the question is not only which types of labels to support, but also how to manage the continuous evolution of that set of types.

A flexible way to manage label sets is to use registries [25], as mentioned in Section 5. Once the necessary registry infrastructure is in place, registries need to be combined with policies so that values in the registry have a well-defined way how they evolve. For API label types and their corresponding values, a rather standard set of policies for registry management would most likely work well:

- Initial Set Any API label landscape will start with a set of initial label types. This set should be the “minimal viable product”, meaning that it is more important to get API label use off the ground, than to have the perfectly curated set of label types. Likewise, the initial values of each label type will be chosen among values with a fixed and well-understood meaning.

- Additions after community review and consensus: The label landscape will continually grow, with new label types and values being added as required. Additional label types should have some motivation documented, and that motivation should be the starting point for a community review. If there is sufficient consensus to add the type, it is added to the set of existing label types. In a similar way, new values should undergo some review so that they broadly follow the general idea of the label type, and ideally do not create overlaps or conflicts with existing entries.

- Semantics of registered label types and values do not change: API labels should always mean the same, so the meaning of an API label type should never be changed. Once it has been registered, users will start using it and will depend on its registered meaning, so changing its meaning would be a breaking change for all uses of the API label. One exception to this rule is that it is possible to clarify and correct the meaning of a registered label value, but this should be used very carefully because any change being made to a label value's meaning should retroactively invalidate or change the way how a label value has been used before.

- Registered label types and values cannot be removed, but can be retired: Label types should never change meaning, but their usage may not be supported or required anymore. If that is the case, there should be a mechanism how a label type or value can be marked as *deprecated* in the registry, so that it becomes clear that this label may appear, but that it should not be actively used anymore. As opposed to removing it from the registry, the semantics of the deprecated value remain registered and available, allowing everybody to still look up what an assigned label type or value means. However, the status also makes it clear that this value should not be used for new labels.

While this recipe for managing label types and values is not the only possible way, it ensures that label management can evolve, and does not suffer from breaking changes along the way. This is thanks to the combination of stable semantics, and the policies on how to evolve them. Because this is a general pattern how to achieve robust extensibility, a very similar recipe can be used to manage the evolution of the value space of individual labels.

8 Conclusion

In this position paper we have made the case for API Labels. Labeling APIs is driven by the real world needs of consumers to quickly assess the main quality attributes of an API and its provider, which are likely to affect the consumer application built using the API in the long term. We have proposed the *API Label Framework (ALF)*: a framework based on the “API the APIs” principle to make API self-descriptive by attaching API labels as metadata to API resources. We also included an initial proposal for a number of possible label types. Some of these can be automatically derived by summarizing information found in API descriptions written by the providers. Other require some external input by a third-party authority. For API Labels to become a trusted mechanism

for API annotation, comparison and selection, there needs to be a verification and validation process which guarantees that consumers can trust the “facts” mentioned in the label.

9 Future Work

As part of future work we plan to make labels self-describing by creating identifiers for each label type you want to support and make label values self-describing by clearly defining the value space for each label. Tooling will be required to automatically extract labels and validate the consistency of labels with the corresponding detailed API descriptions so that API owners can easily test their labels and see how they are working. Once a number of machine-readable API labels become available, tooling to crawl labels will make it easier for developers to explore the “label graph” of the labels that one or more API providers define.

Also policies around label changes will need to be established so that it is well-defined when and how to expect label updates and how these are communicated by tracking the history of a given API. Given that label types and values themselves will likely evolve, it will be important to determine how the set of possible known values is defined and where can the identified label types can be reused from. Registries [25] for API labels and possibly their value spaces are like to play a key role for addressing this challenge.

References

1. Atkinson, L., Rosenthal, S.: Signaling the green sell: the influence of eco-label source, argument specificity, and product involvement on consumer trust. *Journal of Advertising* **43**(1), 33–45 (2014)
2. Becker, T.: To what extent are consumer requirements met by public quality policy? In: *Quality policy and consumer behaviour in the European Union.*, pp. 247–266. Wissenschaftsverlag Vauk Kiel KG (2000)
3. Bidgoly, A.J., Ladani, B.T.: Benchmarking reputation systems: A quantitative verification approach. *Computers in Human Behavior* **57**, 274 – 291 (2016). <https://doi.org/https://doi.org/10.1016/j.chb.2015.12.024>
4. Booth, D., Liu, C.K.: *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*. World Wide Web Consortium, Recommendation REC-wsdl20-primer-20070626 (June 2007)
5. Caswell, J.A., Mojduszka, E.M.: Using informational labeling to influence the market for quality in food products. *American Journal of Agricultural Economics* **78**(5), 1248–1253 (1996)
6. Chu, Y.H., Feigenbaum, J., LaMacchia, B., Resnick, P., Strauss, M.: REFEREE: Trust management for Web applications. *Computer Networks and ISDN systems* **29**(8-13), 953–964 (1997)
7. Cranor, L.F.: *Web Privacy with P3P*. O’Reilly & Associates, Sebastopol, California (September 2002)
8. García, J.M., Fernandez, P., Pedrinaci, C., Resinas, M., Cardoso, J.S., Cortés, A.R.: Modeling Service Level Agreements with Linked USDL Agreement. *IEEE Trans. Services Computing* **10**(1), 52–65 (2017). <https://doi.org/10.1109/TSC.2016.2593925>

9. Hadley, M.: Web Application Description Language (WADL). Tech. Rep. TR-2006-153, Sun Microsystems (April 2006)
10. Kearney, K.T., Torelli, F., Kotsokalis, C.: SLA★: An abstract syntax for Service Level Agreements. In: Proc. of the 11th IEEE/ACM International Conference on Grid Computing (GRID). pp. 217–224 (2010)
11. Kelley, P.G., Bresee, J., Cranor, L.F., Reeder, R.W.: A nutrition label for privacy. In: Proceedings of the 5th Symposium on Usable Privacy and Security. p. 4. ACM (2009)
12. Kopecký, J., Gomadam, K., Vitvar, T.: hRESTS: An HTML Microformat for Describing RESTful Web Services. In: 2008 IEEE/WIC/ACM International Conference on Web Intelligence. pp. 619–625. Sydney, Australia (December 2008). <https://doi.org/10.1109/WIIAT.2008.469>
13. Kotler, P.: Marketing management: analysis, planning, implementation and control. Prentice Hall (1997)
14. Lethbridge, T.C., Singer, J., Forward, A.: How software engineers use documentation: the state of the practice. *IEEE Software* **20**(6), 35–39 (Nov 2003). <https://doi.org/10.1109/MS.2003.1241364>
15. Li, J., Xiong, Y., Liu, X., Zhang, L.: How Does Web Service API Evolution Affect Clients? In: 2013 IEEE 20th International Conference on Web Services(ICWS). pp. 300–307 (June 2013)
16. Nottingham, M.: Home Documents for HTTP APIs. Internet Draft draft-nottingham-json-home-06 (August 2017)
17. Pautasso, C., Wilde, E.: Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design. In: Quemada, J., León, G., Maarek, Y.S., Nejd, W. (eds.) 18th International World Wide Web Conference. pp. 911–920. ACM Press, Madrid, Spain (April 2009)
18. Pautasso, C., Zimmermann, O.: The Web as a Software Connector: Integration Resting on Linked Resources. *IEEE Software* **35**(1), 93–98 (2018)
19. Robie, J., Sinnema, R., Wilde, E.: RADL: RESTful API Description Language. In: Kosek, J. (ed.) XML Prague 2014. pp. 181–209. Prague, Czech Republic (February 2014)
20. Snell, J.M.: Atom License Extension. Internet RFC 4946 (July 2007)
21. Tata, S., Mohamed, M., Sakairi, T., Mandagere, N., Anya, O., Ludwig, H.: RSLA: A service level agreement language for cloud services. In: Proc. of the 9th International Conference on Cloud Computing (CLOUD2016). pp. 415–422. IEEE (2016). <https://doi.org/10.1109/CLOUD.2016.60>
22. The Open API Initiative: OAI. <https://openapis.org> (2016), <https://openapis.org/>
23. Uriarte, R.B., Tiezzi, F., De Nicola, R.: SLAC: A formal service-level-agreement language for cloud computing. In: UCC. pp. 419–426. IEEE (December 2014)
24. Verborgh, R., Steiner, T., Deursen, D.V., Coppens, S., Vallés, J.G., de Walle, R.V.: Functional Descriptions as the Bridge between Hypermedia APIs and the Semantic Web. In: Alarcón, R., Pautasso, C., Wilde, E. (eds.) Third International Workshop on RESTful Design (WS-REST 2012). pp. 33–40. Lyon, France (April 2012). <https://doi.org/10.1145/2307819.2307828>
25. Wilde, E.: The Use of Registries. Internet Draft draft-wilde-registries-01 (February 2016)
26. Wilde, E.: Link Relation Types for Web Services. Internet Draft draft-wilde-service-link-rel-06 (August 2018)
27. Wilde, E.: Surfing the API Web: Web Concepts. In: 27th International World Wide Web Conference. ACM Press, Lyon, France (April 2018)

On Limitations of Abstraction-Based Deadlock-Analysis of Service-Oriented Systems

Mandy Weißbach and Wolf Zimmermann

Martin Luther University Halle-Wittenberg, Institute of Computer Science,
Von-Seckendorff-Platz 1, 06120 Halle, Germany
{mandy.weissbach,wolf.zimmermann}@informatik.uni-halle.de

Abstract. Deadlock-analysis of concurrent service-oriented systems is often done by P/T-net-based approaches. We show that there is a concurrent service-oriented system with synchronous (stack behavior) and asynchronous procedure (concurrent behavior) calls with a deadlock that is not discovered by classical P/T-net-based approaches. Hence, P/T-net-based approaches lead to false statements on absence of deadlocks. We propose an approach based on Mayr's Process Rewrite Systems to model both, concurrent and stack behavior while the deadlock problem remains decidable.

Keywords: Deadlock-Analysis; Concurrency; Petri Net Abstraction; Service-Oriented System

1 Introduction

Van der Aalst's workflow nets is a P/T(place/transition)-net-based approach for checking soundness properties, i.e., the absence of deadlocks or livelocks of business process workflows and their (de)composition [10]. This approach is refinement-based, i.e., the workflow nets are refined to an implementation. The approach might be well-suited for an initial implementation but it is well-known that maintaining the consistency of the model and the corresponding implementation requires disciplined programmers. Hence, it is not uncommon that the model for a service and its implementation becomes more and more inconsistent. Furthermore, there exists certainly many services that are not implemented as a refinement of workflow nets. This does not mean that the approach using workflow nets as a tool for checking soundness property is superfluous, if it is used in the other direction: abstract an implementation to P/T-net and check the abstracted P/T-net for absence of deadlocks.

Since P/T-nets are unable to model stack behavior, any P/T-net-based abstraction of an implementation including stack behavior (recursive procedure calls) can not capture this behavior. In [13] it was shown that finite-state approaches for protocol conformance checking may lead to false positives if recursion is allowed, i.e., the approach reports the absence of protocol conformance violations while the real behavior produces one. In [5] we have shown that using Mayr's Process Rewrite Systems (PRSs), the concurrent and recursive behavior

can be modeled adequately, i.e., false positives can not occur. [1] shows that this PRS-based abstraction can also be made compositional and is therefore as appropriate for service compositions as P/T-nets. In this paper we answer the question whether a similar situation occurs for deadlock analysis using workflow nets (and composing them to P/T-nets).

It turns out that we have a similar phenomenon as for protocol conformance checking:

There is a service-oriented system S with a deadlock where the abstraction of its services to workflow nets and their composition leads to a deadlock-free P/T-net.

Thus, if van der Aalst's workflow nets are used to model the behavior of services, it may lead to false statements on the absence of deadlocks.

Section 2 introduces P/T-nets, the Abstraction and Composition Process, and the Programming Model of our service-oriented System. In Section 3 we explain the main results on limitations of deadlock analysis with the help of an example presented in Section 2. Related Work is discussed in Section 4. Section 5 concludes our work.

2 Foundations

P/T-Nets A *place/transition net* (short P/T-net) is a tuple $\Pi \triangleq (P, T, E, \lambda, \mu_0)$ where

- P is a finite set of *places*
- T is a finite set of *transitions*, $P \cup T = \emptyset$.
- $E \subseteq P \times T \cup T \times P$
- $\lambda : E \rightarrow \mathbb{N}$ is a labelling function
- $\mu_0 : P \rightarrow \mathbb{N}$ is the initial marking

A state in Π is a function $\mu : P \rightarrow \mathbb{N}$. Informally, $\mu(p)$ is the number of *tokens* in place p .

Note that $(P \cup T, E)$ is a bipartite directed graph. The set of *pre-places* of a transition t is defined as $Pre(t) \triangleq \{p : (p, t) \in E\}$. Analogously, the set of *post-places* of t is defined as $Post(t) \triangleq \{p : (t, p) \in E\}$.

A transition t of Π is *enabled* in state μ if $\mu(p) \geq \lambda((p, t))$ for all $p \in Pre(t)$, i.e., p contains at least as many tokens as the edge label of (p, t) .

If an enabled transition t *fires* in state μ , then next state μ' is computed as follows:

$$\mu'(p) \triangleq \begin{cases} \mu(p) + \lambda(t, p) & \text{if } p \in Post(t) \setminus Pre(t) \\ \mu(p) - \lambda(p, t) & \text{if } p \in Pre(t) \setminus Post(t) \\ \mu(p) - \lambda(p, t) + \lambda(t, p) & \text{if } p \in Pre(t) \cap Post(t) \\ \mu(p) & \text{otherwise} \end{cases}$$

In this paper, a P/T-net Π may also have a *final state* μ_f . A state δ ($\neq \mu_f$) is called a *deadlock* if no transition is enabled in δ . The absence of deadlocks is

decidable for P/T-nets. It is furthermore decidable if the final state μ_f is always reachable from the initial state μ_0 .

Fig. 3 shows an example. As usual, places are depicted as circles, transitions are depicted as squares, and tokens are depicted as bullets in places. Here $\mu_0(q_0) = 1$, and $\mu_0(q) = 0$ for all $q \in P \setminus \{q_0\}$. There is no label at the edges. By default, this means $\lambda(e) = 1$ for all $e \in E$. For example, transition t_1 is enabled in μ_0 . If t_1 fires, then for the next state it holds $\mu_1(q_1) = 1$, $\mu_1(i_b) = 1$, and $\mu_1(q) = 0$ for each $q \in P \setminus \{q_1, i_b\}$.

A workflow net is a triple $WF \triangleq (\Pi, I, O)$ where $\Pi = (P, T, E, \lambda, \mu_0)$ is a P/T-net, $I \subseteq P$ is a set of *input places*, $O \subseteq P$ is a set of *output places*, and $I \cap O = \emptyset$. Fig 2 shows four workflow nets. The input and output places are the places on the border of the box.

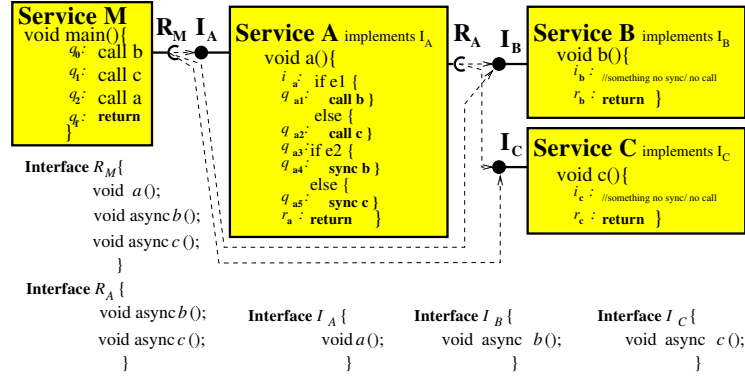


Fig. 1: A service-oriented system with services M , A , B and C . Service M acts as a client. Procedure a is a synchronous procedure while procedures b and c are asynchronous procedures.

Programming Model and Abstraction Process Fig. 1 shows a service-oriented system with a client service M and services A , B and C . Furthermore, Fig. 1 defines the interfaces R_M , R_A , I_A , I_B , and I_C . An *interface* is a finite set of procedure signatures (denoted in C-style).

A service X may provide an interface I_X (*provided interface*), i.e., the procedures in this interface I_X have to be implemented by X . This implementation may call procedures of other services. The set of signatures of these called procedures is the *required interface* R_X of X .

In Fig. 1, service M has no provided interface and services A , B , and C have the provided interfaces I_A , I_B , and I_C , respectively. Furthermore service M has the required interface R_M and service A has the required interface R_A . Services B and C have no required interface.

Control Structure	P/T-Net	Control Structure	P/T-Net
$q : \text{assignment};$ $q' : \dots$		Synchronous procedure p $q : \text{call } p$ $q' : \dots$	
$q_1 : \text{if } e \{$ $q_2 : \dots$ $q_3 : \text{last program point} \}$ $\text{else} \{$ $q_4 : \dots$ $q_5 : \text{last program point} \}$ $q_6 : \dots$		$i_p : \dots$ $r_p : \text{return} \}$	
Synchronization $q : \text{sync } p;$ $q' : \dots$		asynchronous procedure p $a \{ \dots$ $q : \text{call } p$ $q' : \dots$ $q'' : \text{return} \}$	
$i_p : \dots$ $r_p : \text{return} \}$		$i_p : \dots$ $r_p : \text{return} \}$	

Table 1: Control-flow abstractions to P/T-nets

Each procedure p in a required interface R must be connected to a procedure p in a provided interface I .

For example, R_M contains the signature `void a()` and is connected to the provided interface I_A containing the same signature `void a()`. The service-oriented system in Fig. 1 starts its execution by executing `main` in the client M .

Procedures can be synchronous or asynchronous. If a synchronous procedure is being called, the caller waits until the callee has been completed. Therefore synchronous procedure calls behave like classical procedures. In case of recursion, their semantics behaves as stacks. If an asynchronous procedure is being called, the caller and the callee are concurrently be executed. For example, in Fig. 1 procedure a is synchronous and procedures b and c are asynchronous, indicated by the keyword `async`. It is not possible to connect synchronous procedures to asynchronous procedures and vice versa.

There are two possibilities of synchronization for asynchronous procedure calls: First, the caller reaches a `sync`-statement. In this case, the caller waits until the callee returns. Second, the caller reaches a `return`-statement. Then, the caller waits until the callee returns. For example, the statement q_{a4} waits until the call of the asynchronous procedure b in q_{a1} has been completed. The other control structures are the classical ones with the classical semantics.

Table 1 shows different control structures and their abstraction to P/T-nets. The main principle is that each program point corresponds to a place. Each procedure p has a unique entry place i_p and a unique return place r_p .

A token in a place means that the control is at the corresponding program point in the state of program execution. Important control structures are atomic statements, e.g., assignments, conditionals, synchronous procedure calls and re-

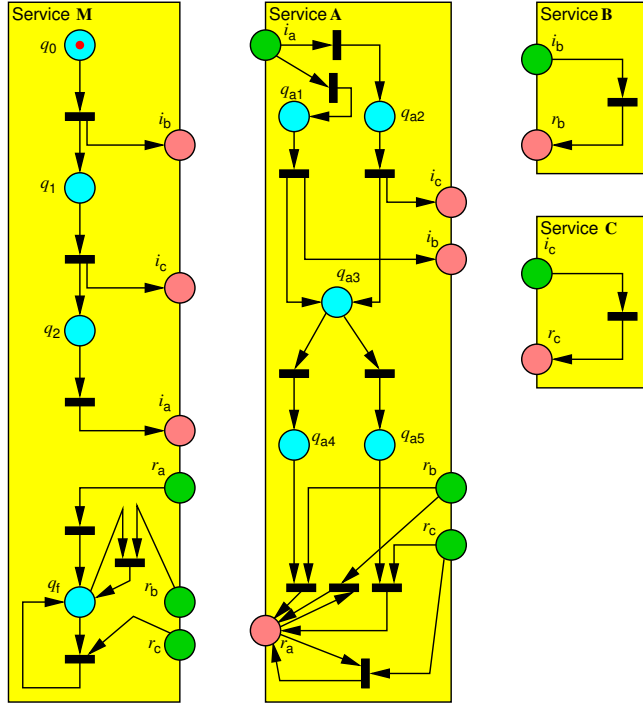


Fig. 2: Workflow net abstraction of Fig. 1

turns, asynchronous procedure calls and returns, and synchronizations. Loops and case statements are abstracted similarly to conditionals.

Note that for each procedure p in a provided interface I_X of a service X , i_p is an input place and r_p is an output place of the workflow net WF_X for X . Similarly, for each procedure q of a required interface R_X of a service X , i_q is an output place and r_q an input place of WF_X . We further assume that a service containing only required interfaces is a client and has an initial marking μ_0 such that $\mu_0(q_0) = 1$ if q_0 is the first program point of *main* and $\mu_0(q_0) = 0$ otherwise. For all non-client services, there is no token in the initial marking.

Fig. 2 shows the workflow nets of the abstractions obtained from the service-oriented system in Fig. 1.

Composition A service-oriented system is implemented by connecting the required interfaces of a service (external call to another service) to a corresponding provided interface of another service. Following the ideas of [7], the *composition of workflow nets* WF_1, \dots, WF_n is a P/T-net

$$P_c \triangleq (P_1 \cup \dots \cup P_n, T_1 \cup \dots \cup T_n, E_1 \cup \dots \cup E_n, \lambda_1 \cup \dots \lambda_n, \mu_0^{(1)} \cup \dots \cup \mu_0^{(n)})$$

under the assumption that all places in the workflow nets $WF_i = (\Pi_i, I_i, O_i)$

with $\Pi_i = (P_i, T_i, E_i, \lambda_i, \mu_0^{(i)})$, $i = 1, \dots, n$ are pairwise disjoint except for input and output places.

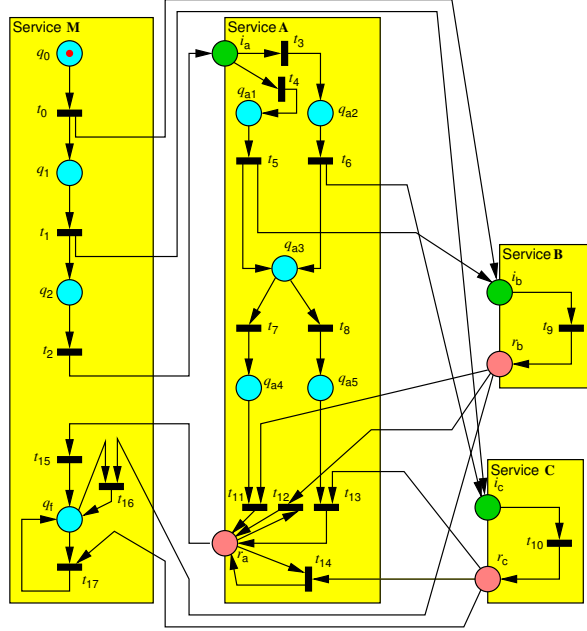


Fig. 3: Composition of the workflow nets in Fig. 2

Fig. 3 shows the composition of the workflow nets in Fig. 2.

Remark 1. Suppose service X calls a procedure p of a service Y . Then output place i_p of the workflow net WF_X is identified with input place i_p of the workflow net WF_Y . Similarly, the output place r_p of WF_Y is identified with the input place r_p of WF_X . Note that the output places i_b of services M and A are both identified with the input place i_b of service B . Similarly, output place r_b of service B is identified with the input places r_b of services M and A , respectively. In certain sense, the treatment of procedures is similar to the treatment in context-insensitive interprocedural program analysis. We therefore call this kind of composition *context-insensitive composition*. In contrast, in *context-sensitive compositions* the workflow net for a procedure p is copied for each call. However, this is impossible if recursion is allowed.

Remark 2. Our approach is similar to [7]. There, the workflow nets are called modules and in addition, each module has a unique starting place α and a unique final place ω . Hence, the abstractions to workflow nets as discussed in our work are modules in the sense of [7]. Our notion of composition corresponds to the notion of syntactic composition of modules.

Remark 3. For Mayr's process rewrite systems (PRS), P/T-nets are equivalent to the class of (P,P)-PRS [8]. The abstraction mechanism leads to a set of PRS rules for each service [5]. The composition is called *combined abstraction* [1]. For the special class of (P,P)-PRS, this corresponds to the composition of workflow nets as described above.

3 Limitations of Deadlock Analysis

Claim 1 *The P/T-net abstraction (cf. Fig. 3) of the service-oriented system in Fig. 1 is deadlock-free.*

Proof. It must be shown that the final state μ_f (i.e. $\mu_f(q_f) = 1$ and $\mu_f(q) = 0$ for all places $q \neq q_f$) is always reached from the initial state μ_0 . For simplicity, for all places q not mentioned in the definition of a state μ , we assume $\mu(q) = 0$.

Step 1: Each state $\mu_1 \in M_1 \triangleq \{\mu : \mu(r_a) = 1, \mu(i_b) + \mu(r_b) + \mu(i_c) + \mu(r_c) = 2\}$ always reaches μ_f

Step 2: Each state $\mu_2 \in M_2 \triangleq \{\mu : \mu(q_{a3}) = 1, \mu(i_b) + \mu(i_b) \geq 1, \mu(i_c) + \mu(i_c) \geq 1, \mu(i_b) + \mu(i_b) + \mu(i_c) + \mu(i_c) = 3\}$ always reaches a state $\mu_1 \in M_1$

Step 3: μ_0 always reaches a state $\mu_2 \in M_2$.

If we have proven this, then the initial state q_0 always reaches q_f .

Remark 4. M_1 contains all states where r_a has one token and services B and C together have two tokens. M_2 describes all states where q_{a3} has one token, services B and C have at least one token and both service, B and C have together three tokens.

Step 1: It is sufficient to consider only situations where service B and C has tokens in r_b and r_c since tokens in i_b and i_c mean that transitions t_9 are t_{10} enabled and i_b and i_c do not have other successors. We consider the following two cases:

- (i) $r_a, r_b,$ and r_c have one token, i.e. only transitions t_{12}, t_{14} and t_{15} are enabled.
- (ii) r_a has one token and r_b has two tokens, i.e., only transitions t_{12} and t_{15} are enabled.

The case where r_a has one token and r_c has two tokens is analogous to (ii). The following tables show all possible firing sequences of (i) and (ii). Each of this firing sequences end in the final state μ_f :

(i) : t_{12}, t_{14}, t_{15} t_{12}, t_{15}, t_{17} t_{14}, t_{12}, t_{15} t_{14}, t_{15}, t_{16}	(ii) : t_{12}, t_{12}, t_{15} t_{12}, t_{15}, t_{16} t_{15}, t_{16}, t_{16}
--	---

Step 2: Analogously to Step 1, it is sufficient to consider only situations where service B and C have their tokens in r_b and r_c , respectively. We consider the case where r_b has two tokens and r_c has one token. The other case (r_b has one token and r_c has two tokens) is proven analogously. In this state, t_7 and t_8 are

the only two transitions being enabled. The following firing sequence all lead to a state $\mu_1 \in M_1$:

t_7, t_{11} reaches state $\mu(r_a) = \mu(r_b) = \mu(r_c) = 1$

t_8, t_{13} reaches state $\mu(r_a) = 1, \mu(r_b) = 2, \mu(r_c) = 0$

Step 3: According to the discussions of Steps 1 and 2, it is sufficient to consider only situations where i_b and i_c have at least one token, respectively, and $\mu(i_b) + \mu(i_c) = 3$. We show that a state $\mu_2 \in M_2$ is always reached from the initial state. Under the above circumstances, the simulation of the P/T-net always starts with the firing sequence t_0, t_1, t_2 reaches a state where i_a, i_b and i_c contain one token, respectively. Now, the transitions t_3 and t_4 are the only enabled transitions (except the inner transitions t_9 and t_{10}). If t_3 fires, then only t_6 is enabled, leading to one token in q_{a3} , one token in i_b , and two tokens in i_c . If t_4 fires, then only t_5 is enabled leading to one token in q_{a3} , two tokens in i_b , and one token in i_c . Both states are in M_2 .

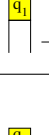
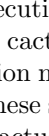
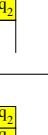
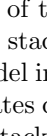
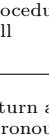
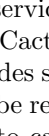
Control Structure	Cactus Stack	Control Structure	Cactus Stack
assignment		asynchronous procedure call	
synchronous procedure call		return asynchronous procedure call	
return synchronous procedure call		synchronization	

Table 2: Execution Semantics with Cactus Stacks (program points)

We look now at the execution of the service-oriented system in Fig. 1. The runtime system is based on cactus stacks. Cactus stacks were introduced as tree of stacks by [3]. Our execution model includes states of unbounded recursion and unbounded concurrency. These states can be represented by cactus stacks. Thus, the execution transforms cactus stacks into cactus stacks. Table 2 shows these transitions. If a synchronous procedure is called, there is transition to the next program point and a stack frame with the initial state of the called procedure is pushed onto a stack. If an asynchronous procedure is called, a new stack frame is created that forks from the caller. The top stack frame of the caller and the bottom element of the new stack are linked together (like a saguaro cactus). Thus, synchronization is only possible with two elements that are forked from a top-of-stack frame.

Claim 2 *The service-oriented system in Figure 1 may end in a deadlock.*

Proof. Table 3 shows an execution trace of the service-oriented system in Fig. 1. In the first step q_0 forks to q_1 and i_b . Then, the control moves from i_b to r_b which waits for synchronization or the return from *main*. Hence, the only possible step is the asynchronous call of c . The next step move the control to r_c . Now, the only possibility is the (synchronous) call of a . This means that the next state q_f and the initial state i_a are pushed on the stack. After this call it is not possible to synchronize with r_b and r_c forked from q_f since q_f is not at top of a stack. The final cactus stack is a deadlock since q_{a5} waits for synchronization with r_c but there is no r_c for synchronization.

Remark 5. [5] discusses the abstraction to general PRS and shows a 1 – 1 correspondence between cactus stacks (of program points) and process-algebraic expressions. Hence, the deadlock can be found by using general PRS.

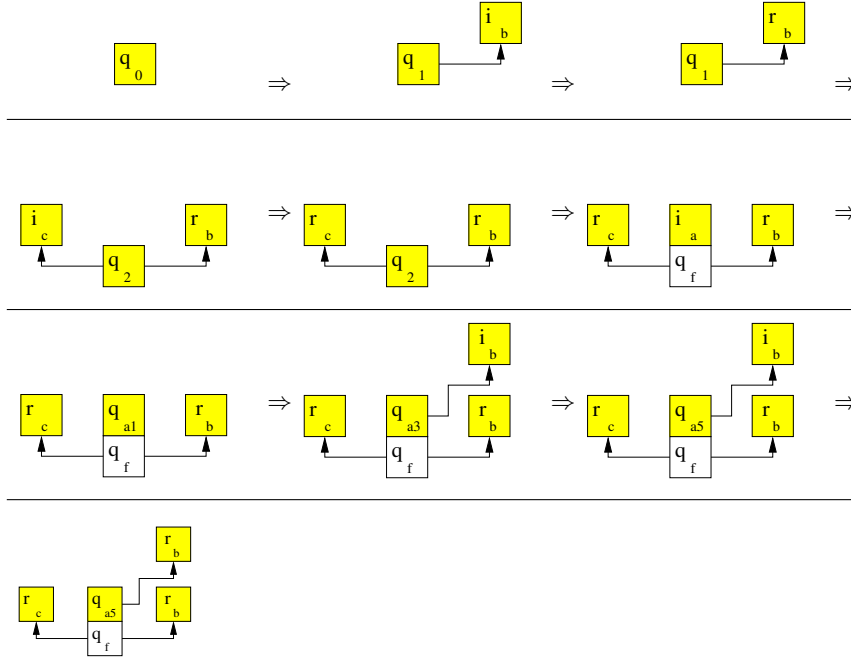


Table 3: Derivation from the initial state to a deadlock

Remark 6. The deadlock in Table 3 means that the control reached r_c , q_{a5} and twice r_b . This means that the P/T-net in Fig. 3 reaches a state that contains

two tokens in r_b , one token in r_c , and one token in q_{a5} . Thus t_{13} is enabled and it is the only transition being enabled. Hence, the deadlock in the execution of the service-oriented system does not correspond to a deadlock in the P/T-net abstraction (based on workflow nets).

4 Related Work

Woflan [11] is a Petri Net-based analysis tool which verifies parallel business process workflows. Recursion of processes are not considered. It is not clear whether their composition is context-sensitive or context-insensitive.

In [9] recursive Petri Nets (rPNs) are used to model the planning of autonomous agents which transport goods from location A to location B and their coordinating problem. The model of rPNs is used to model dynamic processes (e.g., agent's request). Deadlocks can only arise when interactions between agents (e.g., shared attributes) invalidates preconditions. For that reason a coordinating algorithm is introduced to prevent these interactions between agents.

A refinement based approach is described in [6]. Hicheur models healthcare processes based on algebraic and recursive Petri Nets [4], a high level algebraic Petri Net. Hicheur et al. use recursive Petri Net to model subprocesses that are called by a process (e.g., the main process), i.e. a context-sensitive composition. However, to the best of our knowledge, we are not aware of any work on deadlock analysis for recursive Petri Nets.

Bouajjani et al. [2] propose an abstraction-based approach to model control structures of recursively parallel programs (e.g. Cilk, X10, Multilisp). Their approach is based on recursive vector addition systems. They explore the decidability and complexity of state-reachability. It seems that their model is slightly more general than ours as there are situations where the reachability problem becomes undecidable.

Our approach is similar to [7], cf. Remark 2. However, it seems that exactly one call to a module is being considered. Hence, context-sensitivity does not play a role in the notion of composition.

We are not aware of any work stating out the drawback of deadlock analysis of systems with synchronous and asynchronous procedure calls and also synchronization concepts.

5 Conclusion

We presented an example of a service-oriented system with synchronous procedures, asynchronous procedures and a barrier-based synchronization mechanism. We have discussed a straightforward abstraction mechanism to workflow nets and their context-insensitive composition to P/T-nets. Furthermore, we have also shown a runtime based on cactus stacks (which was already be defined as a runtime system of Simula67 [3]). Our main result is an example that the workflow net approach doesn't satisfy its goals for deadlock analysis: the resulting

P/T-net is free of deadlocks (Claim 1) while the execution of the service-oriented system leads to a deadlock (Claim 1). Note, that our example is not a spurious counterexample. A spurious counterexample would be a deadlock in the P/T-net while the service-oriented system is deadlock-free. In our previous work [12], we showed another phenomenon using workflow nets abstractions: the approach only has spurious counterexamples while the real one is not discovered.

Our result shows that in general, deadlock checking based on straightforward P/T-net abstraction with context-insensitive composition should not be used to prove deadlock-freeness. In contrast, PRS-abstractions are able to model the stack behavior of synchronous procedure calls as well. On the other hand, it might be that context-sensitive composition might solve the problem for bound recursion depth. Unbound recursion would require an infinite expansion of the procedure calls.

In future work it remains to investigate the concurrent and recursive concept and also the synchronization concept of other languages (e.g. Java Threads, Simula and so on). Another open issue is the occurrence of deadlocks in a certain recursion depth. We conjecture that if a deadlock associated with recursive behavior (recursion or recursive callbacks) in a service-oriented system occurs, in its PRS-abstraction it always occurs in recursion depth one. If the conjecture would be true, it should be possible to use a P/T-net-based abstraction for deadlock checking (possibly with a special class of context-sensitive compositions of workflow nets).

References

1. Both, A., Zimmermann, W.: Automatic protocol conformance checking of recursive and parallel component-based systems. In: *Component-Based Software Engineering, 11th International Symposium (CBSE 2008)*. pp. 163–179 (October 2008)
2. Bouajjani, A., Emmi, M.: Analysis of recursively parallel programs. In: *ACM SIGPLAN Notices*. vol. 47, pp. 203–214. ACM (2012)
3. Dahl, O.J., Nygaard, K.: Simula: an algol-based simulation language. *Communications of the ACM* 9, 671–678 (1966)
4. Haddad, S., Poitrenaud, D.: Modelling and analyzing systems with recursive petri nets. In: *Discrete Event Systems*, pp. 449–458. Springer (2000)
5. Heike, C., Zimmermann, W., Both, A.: On expanding protocol conformance checking to exception handling. *Service Oriented Computing and Applications* 8(4), 299–322 (2014)
6. Hicheur, A., Dhieb, A.B., Barkaoui, K.: Modelling and analysis of flexible healthcare processes based on algebraic and recursive petri nets. In: *Foundations of Health Information Engineering and Systems*, pp. 1–18. Springer (2012)
7. Martens, A.: Analyzing web service based business processes. In: *International Conference on Fundamental Approaches to Software Engineering*. pp. 19–33. Springer (2005)
8. Mayr, R.: Process rewrite systems. *Information and Computation* 156(1-2), 264–286 (2000)
9. Seghrouchni, A.E.F., Haddad, S.: A recursive model for distributed planning. In: *Proceedings of the 2nd International Conference on Multi-Agent Systems (ICMAS'96)*. pp. 307–314 (1996)

10. Van Der Aalst, W.M.: Workflow verification: Finding control-flow errors using petri-net-based techniques. In: Business Process Management, pp. 161–183. Springer (2000)
11. Verbeek, E., Van Der Aalst, W.M.: Woflan 2.0 a petri-net-based workflow diagnosis tool. In: Application and Theory of Petri Nets 2000, pp. 475–484. Springer (2000)
12. Weißbach, M., Zimmermann, W.: On abstraction based deadlock analysis in service-oriented systems with recursion. In: Proceedings of the European Conference on Service-Oriented and Cloud Computing (ESOCC 2017) (2017), to appear
13. Zimmermann, W., Schaarschmidt, M.: Automatic checking of component protocols in component-based systems. In: Löwe, W., Südholt, M. (eds.) Software Composition. LNCS, vol. 4089, pp. 1–17. Springer (2006)