Institute of Architecture of Application Systems

Service Selection for On-demand Provisioned Services

Karolina Vukojevic-Haupt, Florian Haupt, Dimka Karastoyanova, and Frank Leymann

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{vukojevic, haupt, karastoyanova, leymann}@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{INPROC-2014-34
  author    = {Karolina Vukojevic-Haupt and Florian Haupt and
               Dimka Karastoyanova and Frank Leymann},
  title     = {Service Selection for On-demand Provisioned Services},
  booktitle = {Proceedings of the 18th IEEE International EDOC Conference
               EDOC 2014, 01. – 05. September 2014, Ulm, Germany},
  year      = {2014},
  pages     = {120 - 127},
  doi       = {10.1109/EDOC.2014.25},
  publisher = {IEEE}
}
```


© 2014 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Universität Stuttgart
Germany

Service Selection for On-demand Provisioned Services

Karolina Vukojevic-Haupt, Florian Haupt, Dimka Karastoyanova, and Frank Leymann
Institute of Architecture of Application Systems (IAAS)
University of Stuttgart, Stuttgart, Germany
lastname@iaas.uni-stuttgart.de

Abstract—**Service selection is an important concept in service oriented architectures that enables the dynamic binding of services based on functional and non-functional requirements. The introduction of the concept of on-demand provisioned services significantly changes the nature of services and as a consequence the traditional service selection process does not fit anymore. Existing approaches for service selection rely on the always on semantic of services, an assumption that is not valid for on-demand provisioned services. We tackle this problem by adapting the traditional service selection process and by defining an additional step covering the changes introduced by the concept of on-demand provisioning. Our solution comprises an extended architecture for on-demand provisioning, a metamodel for a service registry, and a detailed definition and discussion of the adapted and extended service selection process. The work presented in this paper allows keeping the advantages of dynamic service binding at runtime and combining them with the advantages of Cloud computing exploited through the concept of on-demand provisioning.**

Keywords— on-demand provisioning and deprovisioning; service selection; service package selection; eScience; Cloud; SOC;

I. INTRODUCTION

The main building blocks in service oriented architectures (SOA) are services, loosely coupled components providing functionality over a unified interface. To realize more complex functionalities, different services can be reused and combined. This concept is called service composition. Usually, service compositions are modeled (modeling time), then deployed on a suitable execution environment (deployment time) and finally executed by this environment (run time). In service oriented computing (SOC) [2] the concept of *publish-find-bind* aims to decouple service providers and service consumers. A service provider registers all services it offers in a service registry (publish). A service consumer, which needs a specific service, then searches the service registry for a suitable service (find). Afterwards, he can start sending requests to the selected service (bind). The step of choosing the right service is called *service selection*. The search for a suitable service and the binding to this service can be performed at modeling time, at deployment time, or at run time of a service composition. When the service selection is done at modeling time or at deployment time, this binding strategy is called *static binding*. Nevertheless, if the service selection is done at run time, this binding strategy is called *dynamic binding*. Using the binding strategy dynamic binding, the needed services are at modeling time described with functional and non-functional requirements. At run time, for each service call the corresponding requirements are forwarded to a middleware component, the so-called enterprise service bus (ESB) [3]. This component carries out the service selection, based on the passed functional and non-functional requirements, and finally binds the service call to a suitable service.

A basic assumption in SOC is that services are always on and available. In traditional SOA scenarios from the business domain, services are typically used continuously. From a service provider point of view, it is therefore absolutely appropriate to make the service constantly available. However, there exist domains where services are used rarely and not regularly, e.g. simulation workflows. In such cases, it is not suitable for a service provider to make his services constantly available as this means wasting resources. In our work, we consider the eScience domain, especially scientific experiments modeled as simulation workflows [5][6]. Simulation workflows are typically executed irregular and rarely. When a simulation workflow is executed, the used services however require significant resources. For the time the simulation workflow is not running, the services are not needed, but the corresponding allocated resources are furthermore blocked. Altogether, this leads to a bad utilization of services and the corresponding resources.

In our previous work we addressed this deficit using Cloud technologies. We developed an approach and architecture for the on-demand provisioning and de-provisioning of workflow execution middleware and services for simulation workflows [1]. In this approach, services including their underlying middleware and infrastructure are provisioned not until they are needed, and de-provisioned when they are not needed anymore. As simulation workflows are typically long running, the additional provisioning time is not expected to affect the execution time noticeably. In such an on-demand provisioning scenario, the traditional service selection process from SOC can no longer be applied. There are two main reasons for this. First, in our approach we use two fundamentally different service types, traditional services with always on semantic (*provisioned services*) and services which are provisioned on demand (*not provisioned services*). Second, not provisioned services are provided as *service packages*. A service package contains all artifacts needed to provision a service automatically. Therefore, the service selection process has to be extended with an additional service package selection step determining a suitable service package.

To solve the problem of service selection for on-demand provisioning and de-provisioning of services, in this paper we contribute (1) an extension of our existing architecture to enable a sophisticated selection of not provisioned services with different types of service packages, (2) a metamodel for a service registry supporting the discussed scenario, and (3) the definition of a service and service package selection process for the on-demand provisioning and de-provisioning of services.

The rest of the paper is organized as follows. In section II we present our previous architecture for the on-demand provisioning and de-provisioning of workflow execution middleware and services for simulation workflows. In section III we extend this architecture to enable service selection also for not provisioned services. In section IV we first introduce our metamodel for the service registry and then we define our service and service package selection process. Some aspects of this new selection process are discussed in detail in section V. An overview about related work is given in section VI and we finish the paper with a summary and outlook in section VII.

II. Basic Architecture

In our previous work we have developed the architecture of a system supporting our approach for on-demand provisioning and de-provisioning of workflow execution middleware and services needed for the execution of (simulation) workflows [1]. We present the architecture in Fig. 1, where we distinguish between components run locally on the user's machine and the components run on a Cloud. We also show which components are used during which life cycle phases of the involved applications (i.e. simulation workflows, execution middleware, services). The life cycle phases we consider here are the modeling of simulation workflows, the middleware runtime/execution phase and the service runtime phase.

A. Modeling Phase

The architecture components used during the modeling phase are the *modeling and monitoring tool* [8] and the *bootware* running locally on the user's machine, and the *service package repository*, the *service registry* and the *user registry* running in a Cloud environment. These components are active during all life cycle phases. In the modeling phase, the modeling and monitoring tool is used to model workflows. The service registry and the service package repository provide all services that can be used by the workflows. The bootware is utilized by the modeling and monitoring tool to start the next life cycle phase, the middleware runtime phase.

The bootware is the basic piece of software needed to provision the workflow execution middleware (in a Cloud environment). Instead of provisioning the whole workflow execution middleware in one step, we follow a two-step bootstrapping process. In the first step (Fig. 1, step 1) the bootware provisions the provisioning engine and its underlying middleware and infrastructure to a Cloud environment. This reduces the complexity of the bootware component by limiting its capabilities to the provisioning of one special component - the provisioning engine. The provisioning engine itself is a generic component able to provision any kind of service and is a rather complex system [9]. In the second step, the bootware calls the provisioning engine that provisions the workflow execution middleware in a Cloud environment.

The service package repository contains service packages. Services that are available in the service package repository are always registered in the service registry. The service registry is a central data store containing information about all available services and enabling their discovery. The information provided includes functional and nonfunctional properties of a service and a reference to the corresponding service package in the service package repository. The information in the service registry is not only about services stored in the service package repository but also about services that are already available (and provided by a third party).

We distinguish between two kinds of services. The first kind of service is provided by a service provider, who also manages the service. The scientist can use this service, but he has no knowledge about the implementation and the underlying middleware and infrastructure. We call this kind of service a *provisioned service*. For the second kind of service all artifacts needed to provision the service and the underlying middleware and infrastructure can be accessed by the scientist. This kind of service we call a *not provisioned service*. Provisioned services follow the always on semantic, they are running and ready to use. Not provisioned services have to be explicitly provisioned before they can be used. In our previous work we worked out an extended classification for service binding strategies [1]. Typical strategies for static and dynamic service binding rely on provisioned services. To enable the on-demand provisioning and de-provisioning of services including their underlying middleware and infrastructure we defined a new service binding strategy which we call *dynamic binding with software stack provisioning*. This service binding strategy is based on not provisioned services.

A not provisioned service can furthermore be a *dedicated* or a *shared service*. A dedicated service can or may only process one service call at the same time. If several service calls are sent to the same dedicated service, for every service call we have to provision a new instance of the service including its underlying middleware and infrastructure. An example for such a dedicated service can be a simulation service needing a lot of compute resources without having any elasticity capabilities. A shared service can in contrast process several service calls at the same time.

Considering the characteristics of the service types mentioned above, the service registry stores specific information for each type of service. Independent of the service type a link to the interface description is available. For provisioned services the endpoint is already known and therefore stored. For not provisioned services the service registry contains a link to the corresponding service package in the service package repository and if the service is dedicated or shared. In addition, the number of currently running instances is also stored.

B. Middleware RuntimePhase

The workflow middleware runtime phase is supported by the components of the simulation workflow execution middleware provisioned at the end of the modeling phase. In

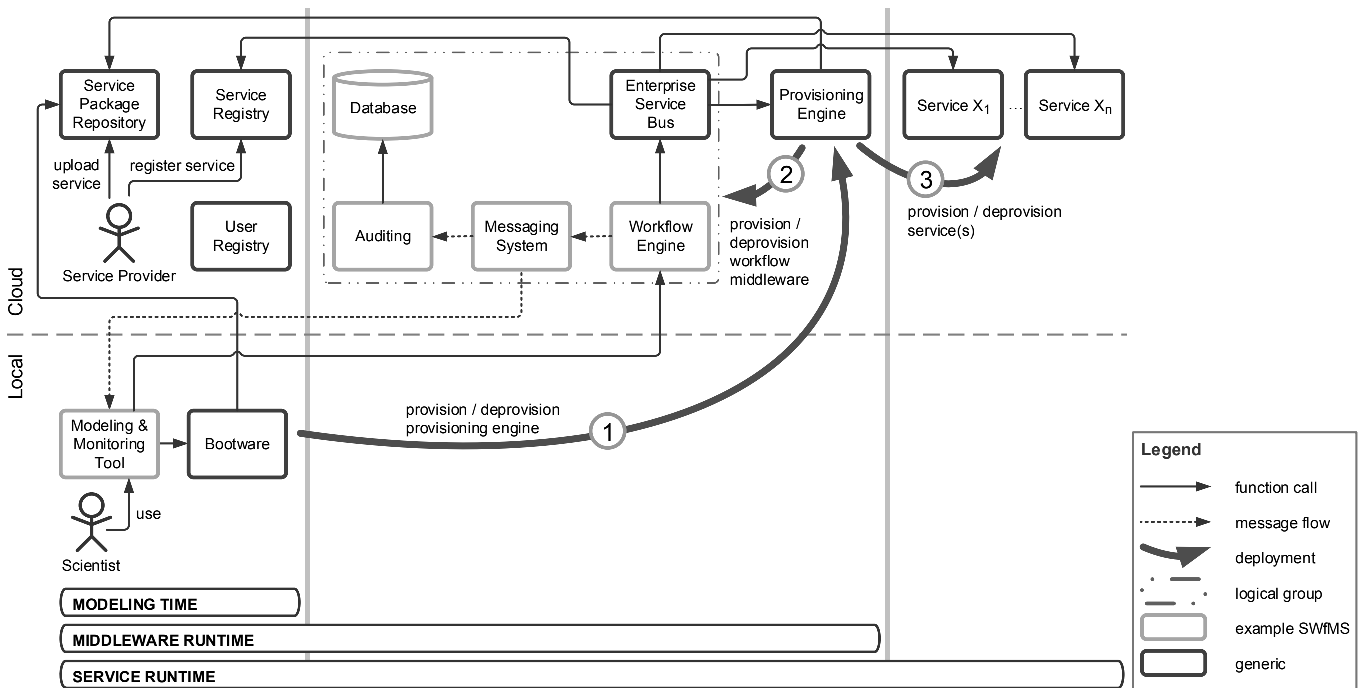

Fig. 1. Basic Architecture for On-demand Provisioning

our example these are the SimTech SWfMS [7], the ESB and the provisioning engine. These components interact with the components used already in the modeling phase. The ESB receives service calls from the workflow engine to invoke services on behalf of workflow activities. For provisioned services the ESB selects the endpoint of a service from the service registry and forwards the service call. For not provisioned services the ESB interacts with the service registry and the provisioning engine. First it gets all information needed to provision the service, like a reference to the service package repository or if the service is dedicated or shared, from the service registry. Then the ESB calls the provisioning engine to provision the service (which starts the runtime phase of the service life cycle). The provisioning engine gets all needed artifacts like the implementation of the service from the service repository and uses these artifacts to provision the service including its underlying middleware and infrastructure (Step 3 in Fig. 1). After the service provisioning is done, the ESB forwards the service call to the newly provisioned service.

C. Service RuntimePhase

During the service runtime phase the services are executing the functionality they are implementing. At the beginning of this phase all components of our architecture shown in Fig. 1 are provisioned and running. As soon as a service has finished its computation the result is returned to the ESB, which in turn sends it back to the workflow engine.

D. Deprovisioning of Services and Middleware

For dedicated services the ESB then calls the provisioning engine to de-provision the service. For shared services the ESB first checks if the service is still processing other service calls. Only if the service is idle it will be de-provisioned.

After the workflow engine has finished the execution of all running workflows, the bootware initializes the de-provisioning of the workflow execution middleware. In the first step the provisioning engine de-provisions all other middleware components. In the next step the bootware de-provisions the provisioning engine.

III. EXTENDED ARCHITECTURE

So that a service can be automatically provisioned in our architecture, all artifacts needed for the provisioning have to be available as a service package. Such an artifact is for example the topology of the service, i.e. a description of which applications, middleware and infrastructure are required to operate a service and how these are connected. Other artifacts are the implementations of each component respectively the references to these implementations. A service package can be available in different established formats such as Chef[1], Puppet[2] or TOSCA [4]. For each of these formats there exist provisioning engines which can handle the corresponding format. For example, a service package in Chef format can be read and automatically provisioned by a Chef Provisioning Engine. However, to provision a service package in TOSCA format, a special provisioning engine for TOSCA is needed. In our previous realization of the architecture described in section II we use TOSCA for the description of the service packages, OpenTOSCA [10] as provisioning engine and Amazon AWS[3] as Cloud environment. We could have also realized our architecture using a different service package format and a different provisioning engine, for example Puppet and a Puppet provisioning engine. Our architecture is designed to be generic,

[1] http://www.getchef.com/
[2] http://puppetlabs.com/puppet/puppet-open-source
[3] http://aws.amazon.com/

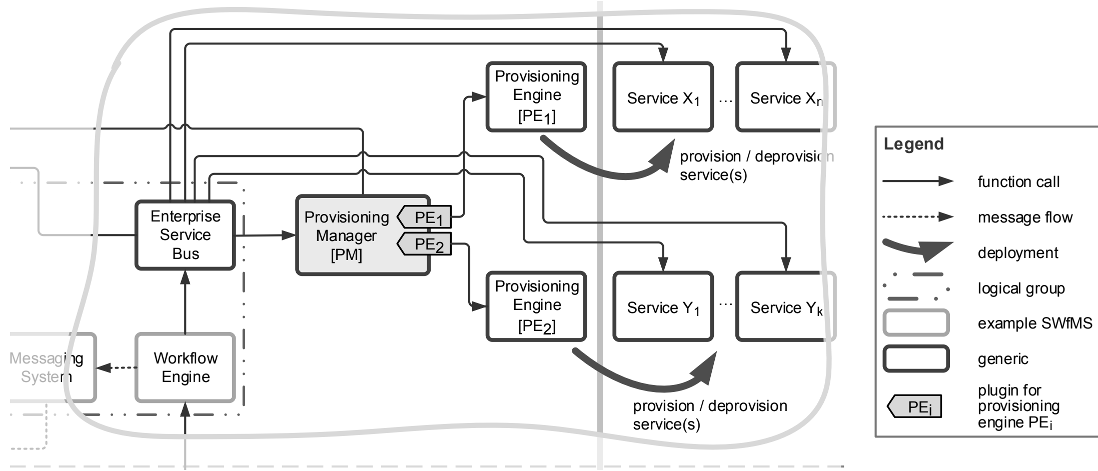

Fig. 2. Extended Architecture for On-demand Provisioning

there is no dictation about a concrete provisioning format and a concrete provisioning engine. Only upon realization the decision for a concrete format has to be made.

As discussed, our previous architecture does not support multiple provisioning formats. However, when for example a scientist models a workflow he may use services provided by several other scientists. It is quite possible, that these scientists use different formats for their service packages. In Fig. 3 we explain this issue in more detail. On the left part of the figure a workflow including two communication activities C and D is shown. Activity C calls a service implementing the interface x and activity D calls a service implementing the interface y. On the right part of the figure a service package repository is depicted. This repository contains a service package in TOSCA format which contains a service implementing the interface x. Moreover, it contains a service package in Puppet format as well as a service package in Chef format, both implementing the interface y. Since activity C of the depicted workflow calls a service which is only available as a TOSCA service package and activity D calls a service which is not available as TOSCA service package, to provision these two services two different provisioning engines are needed.

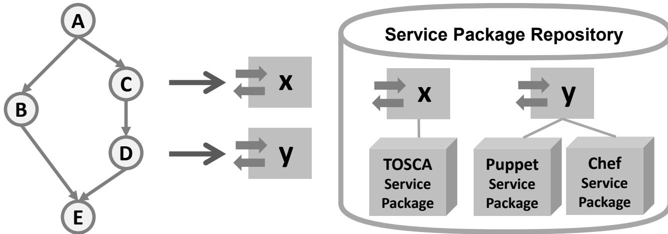

Fig. 3. Different Service Package Types

As a consequence, we extend our architecture to support the execution of workflows that call services that are available in different service package formats and therefore have to be provisioned by different provisioning engines. In Fig. 2 we show the excerpt of our architecture in which we realized this extension. In our previous architecture there was a direct information flow between the ESB and the provisioning engine. In our extended architecture we introduce a new component, the so-called *provisioning manager* (depicted in the center of Fig. 2, initially described in [14]). This component decouples the ESB from the provisioning logic. The ESB receives service calls and is responsible to forward them to suitable services. The provisioning manager in contrast handles all tasks related to the provisioning of services. When receiving a reference to a service package from the ESB, the provisioning manager retrieves the corresponding service package and its meta data from the service package repository. Depending on the format of the service package, the provisioning manager decides which provisioning engine is able to process this service package and finally forwards it to the selected provision engine.

The architecture of the provisioning manager is modular, as the provisioning manager can be extended by plugins. A plugin connects a provisioning engine to the provisioning manager. The plugin declares to the provisioning manager which service package format and which target Cloud environment is supported by the corresponding provisioning engine.

IV. Service and Service Package Selection

Using the binding strategy “dynamic binding with software stack provisioning” changes the service selection process. On the one hand the selection process has to consider both, provisioned services as well as not provisioned services. On the other hand for not provisioned services an additional service package selection is needed. Before introducing the service and service package selection process, we will present the metamodel for the service registry used in our approach. The ESB interacts with the service registry based on this metamodel.

A. Metamodel for Service Registry

The metamodel of the service registry is depicted in Fig. 4 as entity relationship diagram (in Chen notation). In this section we will only present the parts of the metamodel that are relevant in context of this paper. The service registry provides a set of *service configurations*. A service configuration describes the combination of a *service interface*, i.e. the functional properties of a service, and a set of nonfunctional properties, the so-called *quality of services (QoS)*. QoS are modeled as simple name-value pairs. Although we consider

service selection as an important step for dynamic binding, the details of how requirements and properties are matched is not in the focus of our work and there already exist several sophisticated approaches for this [15][16]. The metamodel allows multiple service configurations with the same interface but different QoS. A service call, which generally consists of functional and nonfunctional requirements, can be mapped to at most one service configuration. The part of the metamodel described so far represents a service on an abstract level. In addition the service registry also provides information on how to access specific service instances.

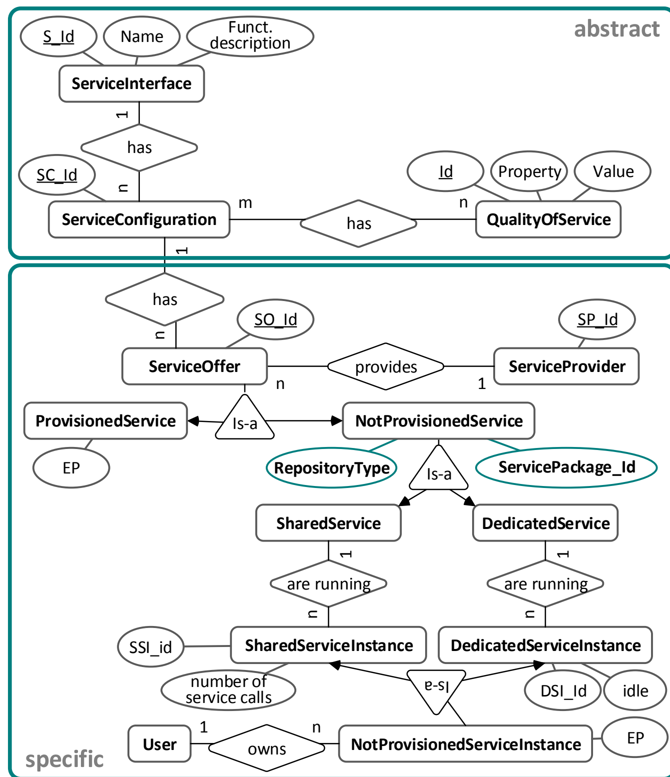

Fig. 4. Metamodel for Service Registry

For a service configuration there can exist multiple *service offers*. A service offer is offered by exactly one *service provider*. A service configuration therefore can be offered by multiple service providers and a service provider can offer multiple service configurations. We distinguish two types of service offers. A *provisioned service* represents a traditional service as known from SOC, i.e. it is always on and available. For such a service an endpoint is provided in the service registry. A provisioned service is a functionality provided at an endpoint with certain nonfunctional properties, everything else is transparent. In contrast, a *not provisioned service* at first has to be explicitly provisioned before it can process service calls.

Consequently for a not provisioned service instead of an endpoint a *service package reference* is provided which points to a service package repository. In the service package repository all data and metadata needed to provision a not provisioned service is stored. Our metamodel allows that a service configuration can be provided by multiple not provisioned services, i.e. for one service configuration there can exist multiple service packages. As one service package can be provisioned multiple times, for a not provisioned service there can exist multiple *not provisioned service instances* which are also managed in the service registry. In addition for not provisioned services we distinguish between *shared services* and *dedicated services* and consequently between *shared service instances* and *dedicated service instances*. For shared service instances the service registry stores the number of currently processed service calls. This information is necessary to determine if a shared service instance is still needed or if it can be safely de-provisioned. Each instance of a not provisioned service is assigned to a user. This user initiated the provisioning of the service and only this user is allowed to call this service. Every instance of a not provisioned service is again available over an endpoint.

B. Service and Service Package Selection Process

In the following we will show how the service and service package selection process is realized in our architecture. In Fig. 5 we present the part of our architecture realizing the service binding. The *workflow engine* is responsible for the execution of the workflows. The *enterprise service bus* coordinates the processing of the service calls. The *service registry* is a global directory containing information about all services. It offers information about functional and nonfunctional properties of a service. For each not provisioned service the *service package repository* contains the corresponding service package together with provisioning metadata. The *provisioning manager* is capable to provision service packages using a suitable *provisioning engine*.

Service calls are initiated by the workflow engine (Fig. 5, step1). A service call contains the actual payload as well as different metadata (step 2). The *functional requirements* (FR) describe the required interface, the *nonfunctional requirements* (NFR) describe requirements concerning the quality of a service, for example cost or security. Whereas the functional and nonfunctional requirements correspond to traditional SOC concepts, the *provisioning requirements* (PR) are specific for our on-demand provisioning approach. They describe requirements specific for the provisioning process, for example allowed cloud providers or the region where resources have to be provisioned.

When receiving a service call, the ESB executes a *service discovery* (step 3). In this step all service configurations which are compliant with the functional requirements of the service call are determined by the service registry (step 4). Afterwards a *service selection* is carried out (step 5). In this step all service offers fulfilling also the nonfunctional requirements are determined (step 6). If the result set contains at least one provisioned service, the service selection component returns exactly one endpoint (of a provisioned service). In this case the ESB forwards the service call to the selected endpoint (step 7a). If the result set contains no provisioned services but at least one running shared service, the service selection component returns exactly one endpoint (of a running shared service) and the ESB forwards the service call to this endpoint (step 7a). If the result set however contains no provisioned services and no running shared services, the service selection component returns a service package reference for each service

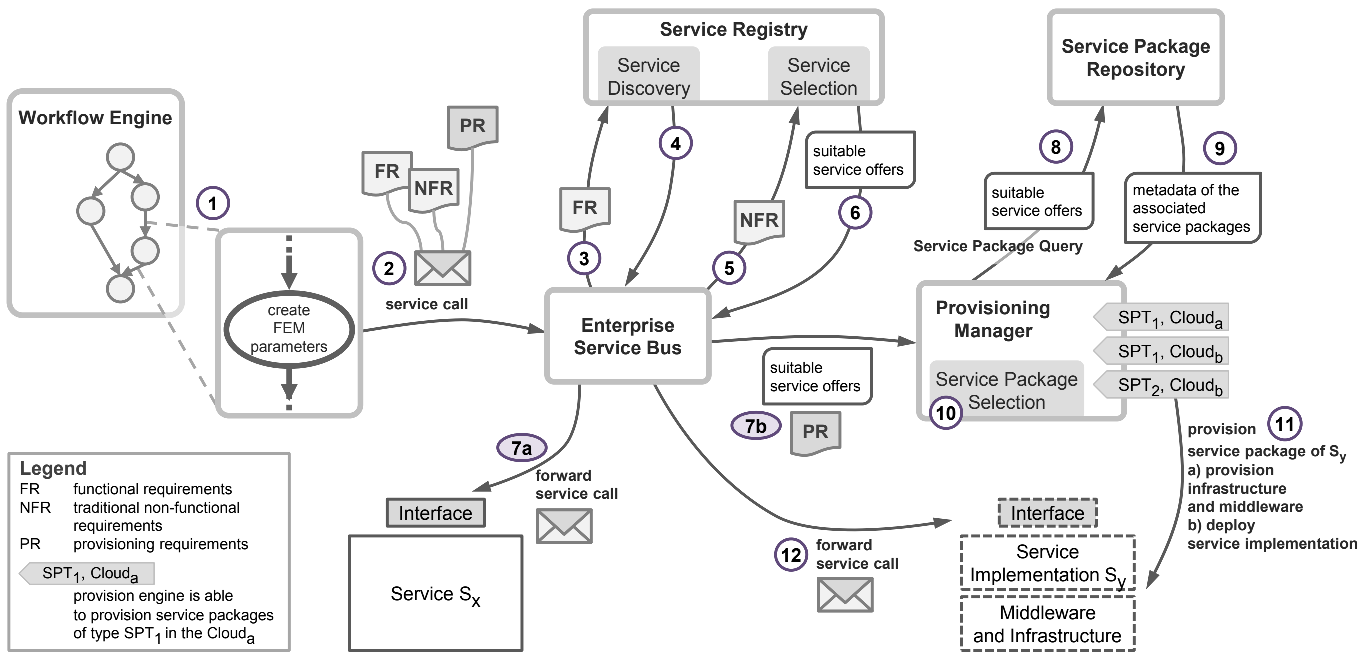

Fig. 5. Architecture for Service Selection and Service Package Selection

offer in the result set. The ESB forwards these service package references together with the provisioning requirements to the provisioning manager (step 7b). Afterwards the provisioning manager dissolves the references by querying the service package repository for the metadata of the referenced service packages (step 8, 9). Then the provisioning manager carries out a service package selection. He selects exactly one service package which on the one hand fulfills the provisioning requirements of the service request and which on the other hand can be processed by one of the available provisioning engines (step 10). After that the selected service package is provisioned by a suitable provisioning engine (step 11). In the last step the ESB forwards the service request to the service provisioned before (step 12).

V. Discussion

A. Service Selection

In traditional SOC usually the service selection step returns exactly one service offer i.e. exactly one service endpoint [13][3]. In our approach this only applies when the set of suitable service offers contains at least one provisioned service. However if the set of suitable service offers contains only not provisioned services, the service selection step returns service package references for all suitable service offers. The ESB then forwards these service package references to the provisioning manager. The provisioning manager encapsulates all provisioning related functionality and therefore has all information available to decide which service package he is actually able to provision. In addition we also delegate the evaluation of the provisioning requirements to the provisioning manager. As a result our architecture shows a clear separation of traditional service selection and routing capabilities (ESB and service registry) on the one hand and provisioning related components (provisioning manager and service package repository) on the other hand.

In Fig. 6 we show an example that further illustrates the service selection for not provisioned services. As a starting point a set of all provided service offers is depicted on the left. This set contains 12 service offers with different functional and nonfunctional properties. The functional properties are depicted by the shape of the service offer icon, the nonfunctional properties are depicted by the hatching of the service offer icon. After a service call arrives, in the first step a service discovery is performed i.e. all service offers providing a certain interface are selected. In our example the wanted interface is symbolized by a square. The service offers S1, S2, S4, S5, S6, and S8 provide the wanted interface and are therefore candidates for the service request. In the second step a service selection is performed on this candidate set i.e. all service offers fulfilling the nonfunctional requirements are selected. In our example the nonfunctional requirements are symbolized by a diagonal hatching. The service offers S1, S4, and S8 fulfill these nonfunctional requirements and are therefore still candidates for the service request. In the third step the service package discovery is performed, i.e. for each service offer the corresponding service package is determined. The following service package selection consists of two steps. First the provisioning manager matches the provisioning requirements with the provisioning capabilities of the service packages (step 4). In our example the provisioning requirements states that the service has to be provisioned in the Amazon Cloud infrastructure (AWS). These requirements are fulfilled by service offer S4 and S8. Second the provisioning manager matches the formats of the service packages with capabilities of the available provisioning engine plugins. In our example the service package of S4 has the format “Chef” and the service package of S8 has the format “TOSCA”. The provisioning manager has two plugins available both supporting Chef but for different Cloud infrastructures. As a result the service package of S4 is selected (step 5). In our example this service offer fulfills all requirements – functional, nonfunctional and

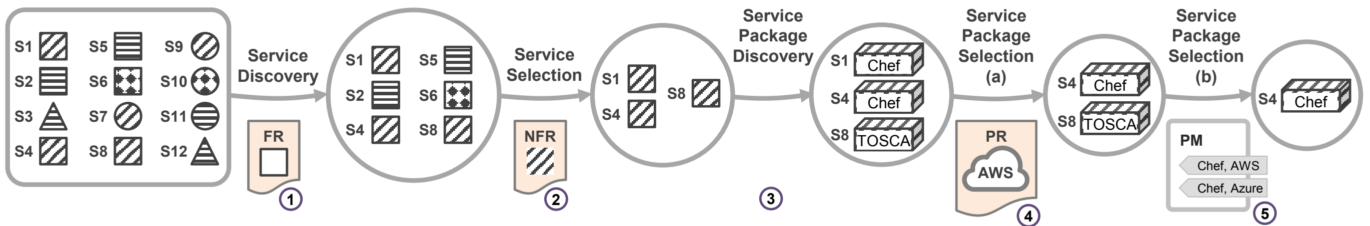

Fig. 6. Service Discovery, Service Selection, Service Package Discovery & Service Package Selection

provisioning requirements - and it can be provisioned by one of the available plugins of the provisioning manager.

One important aspect in the just described process is that for not provisioned services the service selection component returns all compliant service offers. Afterwards the provisioning manager can select a service offer containing a service package which is suitable for provisioning. If the service selection component would return always only one service offer, like known from traditional SOC, this can lead to situations where a service request cannot be processed although a suitable service offer exists. Considering the example of Fig. 6 discussed before, if the service selection component would for example return only service offer S1 (in step 2), then the service package selection (step 4) will result in an empty set and the service request cannot be processed. The service package of service offer S1 does not fulfill the provisioning requirements and the service packages of service offers S4 and S8 have been already discarded in the previous service selection step (step 2).

B. Configuration opportunities for the user

When the service selection component determines the set of compliant service offers, i.e. all service offers fulfilling the functional requirements (i.e. the interface) and the non-functional requirements (i.e. QoS) of the service call, this result set can consist of provisioned services as well as not provisioned services. In this case our system per default always returns the endpoint of a provisioned service. However, this behavior is configurable. The user can define his preferred service type: he can choose between provisioned service and not provisioned service. An advantage of provisioned services is that they are always available, whereas not provisioned services have to be initially provisioned before they can process a service call. On the other hand, it is very possible that the scientists prefer not provisioned services. They may rather trust a not provisioned service available as service package than a provisioned service that simply provides an endpoint. A service package contains details about the implementation and the structure of the service, an often very important aspect in the context of traceability, reproducibility and linked experiments [12][11].

By means of the service selection decision tree depicted in Fig. 7 it is described, which results the service selection component returns, depending on the user's configuration. In the left subtree the configuration option "prefer provisioned service" is shown. This configuration corresponds to the default configuration of our system. When the set of compliant service offers contains at least one provisioned service, the service selection component returns an endpoint of exactly one provisioned service. Afterwards, the ESB forwards the initial service call to this endpoint (see also Fig. 5, step 7a). However, if the set of service offers does not contain a provisioned service but at least one running shared service, the service selection component returns an endpoint of exactly one running shared service. Then the ESB forwards the service call to this selected endpoint (Fig. 5, step 7a). If the set of service offers does contain neither a provisioned service nor a running shared service, the service selection component returns for each service offer a service package reference. Afterwards the ESB forwards these service package references and the provisioning requirements to the provisioning manager (Fig. 5, step 7b).

In the right subtree of the service selection decision tree depicted in Fig. 7 is shown, which results are returned by the service selection component for the configuration option "prefer not provisioned service". If the set of compliant service offers contains at least one running shared service, the service selection component returns the endpoint of exactly one running shared service. Then the ESB forwards the initial service call to this endpoint (see also Fig. 5, step 7a). However, if the set of compliant service offers contains at least one not provisioned service but no running shared service, the service selection component returns a service package reference for

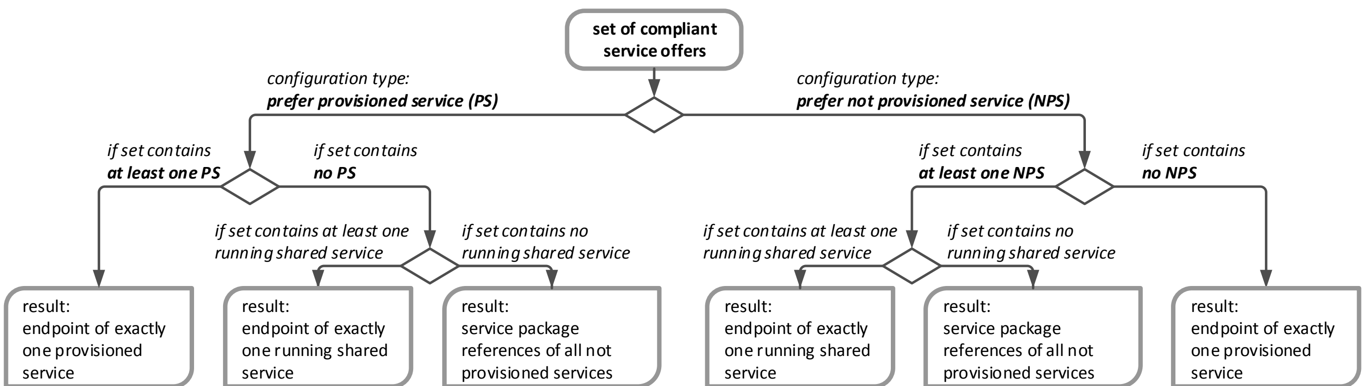

Fig. 7. Service Selection Decision Tree

each service offer of the type not provisioned service. Afterwards the ESB forwards these service package references together with the provisioning requirements to the provisioning manager (Fig. 5, step 7b). If the set of compliant service offers does not contain any not provisioned service, the service selection component returns an endpoint of exactly one provisioned service. The ESB then forwards the initial service call to this endpoint (Fig. 5, step 7a).

VI. Related Work

There exist several approaches for the on-demand provisioning of services [17][18][19]. However, all these approaches do not tackle possible implications on service selection imposed by the concept of on-demand provisioning.

In [19] it is assumed, that all services available for on-demand provisioning provide the same interface. In this work, the dynamicity is in the composition of the middleware and infrastructure a service is hosted on. These parts of the service topology are dynamically selected at runtime, based on the non-functional requirements of the corresponding service call.

In [18] the proposed system at first tries to satisfy a service call using running services. The paper presents, how the current load of available services can be determined so that a request is, if possible, forwarded to a service with low load. If there is no service available, the system starts the on-demand provisioning process. However, in this step there is no selection of an appropriate virtual machine image performed. Instead, the mapping of a service call to a matching image is already contained in the process model. To summarize, this approach supports service selection but no service package selection.

In [20] an on-demand provisioning approach for grid environments is proposed. Similarly to [19] the focus of this work is the selection of an appropriate grid node a requested service will be provisioned on. It is assumed, that the service requestor explicitly asks for the provisioning of a certain service package, i.e. in contrast to other approaches the provisioning is not handled transparently.

VII. Summary and Outlook

In our previous work we introduced and realized the concept of on-demand provisioning and de-provisioning of workflow execution middleware and services for simulation workflows. Besides its advantages like optimized resource allocation and a user friendly way of managing complex systems, this approach has some implications on the traditional service selection known from SOC. In this paper we developed a solution approach for this challenge. We introduced an extended architecture for on-demand provisioning supporting service selection as well as service package selection. As part of this architecture we also provided a metamodel for the service registry as foundation for the selection process. Finally we gave a detailed description and discussion of the service and service package selection process. As a result our extended architecture is able to transparently handle service selection for provisioned as well as not provisioned services.

Besides the ongoing realization of the whole system we plan to extensively evaluate our system using a real world use case from the domain of simulation workflows. Although we already achieved some promising results regarding some single aspects of our approach [1][21], an evaluation of an overall end to end scenario is still missing.

Acknowledgement

K. Vukojevic-Haupt and D. Karastoyanova would like to thank the German Research Foundation (DFG) for financial support of the project within the Cluster of Excellence in Simulation Technology (EXC310/1) at the University of Stuttgart. This work was partially funded by the BMWi project Migrate! (01ME11055).

References

[1] Vukojevic-Haupt, K.; Karastoyanova, D.; Leymann, F.: *On-demand Provisioning of Infrastructure, Middleware and Services for Simulation Workflows*. In: Proceedings of SOCA 2013.

[2] Papazoglou, M.P.:Service-oriented computing: concepts, characteristics and directions. In: Proceedings of WISE 2003

[3] Chappell, D.: *Enterprise Service Bus: Theory in Practice*. 2004.

[4] *Topology and Orchestration Specification for Cloud Applications* Version 1.0. OASIS Committee Specification 01. http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html.

[5] Sonntag, M. et al.: *Using Services and Service Compositions to Enable the Distributed Execution of Legacy Simulation Applications*. In: Proceedings of ServiceWave 2011.

[6] Görlach, K. et al.: *Conventional Workflow Technology for Scientific Simulation*. In: Guide to e-Science, Springer-Verlag, 2011.

[7] Sonntag, M.; Karastoyanova, D.: *Ad hoc Iteration and Re-execution of Activities in Workflows*. In: International Journal On Advances in Software. Vol. 5 (1 & 2), Xpert Publishing Services, 2012.

[8] Sonntag, M.; Karastoyanova, D.: *Next Generation Interactive Scientific Experimenting Based On The Workflow Technology*. In MS 2010.

[9] Lipton, P.: *Escaping Vendor Lock-in with TOSCA, an Emerging Cloud Standard for Portability*. In: CA Technology Exchange 4, 1, 2013.

[10] Binz, T.; Breitenbücher, U.; Haupt, F.; Kopp, O.; Leymann, F.; Nowak, A.; Wagner, S.: *OpenTOSCA - A Runtime for TOSCA-based Cloud Applications*. In: Proceedings of ICSOC 2013.

[11] Leymann, Frank: *Linked Compute Units and Linked Experiments: Using Topology and Orchestration Technology for Flexible Support of Scientific Applications*. In: Software Service and Application Engineering, 2012.

[12] Giles, J.: *The trouble with replication*. In: Nature, 442(7101), 2006.

[13] Leymann, F: *The (Service) Bus: Services Penetrate Everyday Life*. In: Service-Oriented Computing - ICSOC 2005.

[14] Schneider, V.: *Dynamic Provisioning of Web Services for Simulation Workflows*. Diploma Thesis 3473, IAAS, University of Stuttgart, 2013.

[15] Vu, L.-H. et al.: *QoS-Based Service Selection and Ranking with Trust and Reputation Management*. Proceedings of CoopIS 2005

[16] Raghuram, M. et al.: *Agent-based service selection*. In: Journal of Web Semantics, Volume 1, Issue 3, April 2004.

[17] Chrysoulas, C. et al.: *Applying a Web-Service-Based Model to Dynamic Service-Deployment*. In Proceedings of CIMCA 2005.

[18] Dornemann, T. et al.: *On-Demand Resource Provisioning for BPEL Workflows Using Amazon's Elastic Compute Cloud*. In: CCGRID 2009.

[19] Retter, R. et al: *Combining Horizontal and Vertical Composition of Services*. In: Service Oriented Computing and Applications, 2012.

[20] Kecskemeti, G. et al.: *Automatic Service Deployment Using Virtualisation*. In: Proceedings of PDP 2008.

[21] Strauch, S.; Andrikopoulos, V.; Karastoyanova, D.; Vukojevic-Haupt, K.: *Migrating eScience Applications to the Cloud: Methodology and Evaluation*. In: Cloud Computing with E-science Applications, 2014.

All links were last followed on 18.06.2014.