Institute of Architecture of Application Systems

Dynamic Service Provisioning for the Cloud

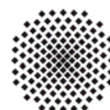
Katharina Görlach and Frank Leymann

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

BIBTEX:

```
@inproceedings{Goerlach12,
  author    = {Katharina Görlach and Frank Leymann},
  title     = {Dynamic Service Provisioning for the Cloud},
  booktitle = {Proceedings of the 9th IEEE International
               Conference on Services Computing, SCC 2012
               24-29 June 2012, Honolulu, Hawaii, USA},
  year      = {2012},
  pages     = {555--561},
  publisher = {IEEE Computer Society}
}
```

© 2009 IEEE Computer Society. 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

Dynamic Service Provisioning for the Cloud

Katharina Görlach
Institute of Architecture of Application Systems
University of Stuttgart
Germany
goerlach@iaas.uni-stuttgart.de

Frank Leymann
Institute of Architecture of Application Systems
University of Stuttgart
Germany
leymann@iaas.uni-stuttgart.de

Abstract—**This paper introduces a method realizing dynamic provisioning of services in a distributed environment. Depending on a particular state of infrastructure the call of a service can lead to a new instance in the infrastructure or to using an existing instance. Hence, the dynamic deployment allows optimized distribution of service instances within a certain infrastructure. The paper introduces a context model for services that are registered in a distributed runtime environment. Furthermore, algorithms are introduced determining the need for instantiation as well as the best location for deployment. Hence, the best location is determined by correlating the context model, the certain state of infrastructure as well as data transfer costs.**

Web Service, Service Composition, Service Provisioning, Cloud Computing, Elasticity

I. INTRODUCTION

Service-based applications are mostly composite and executed in a distributed runtime environment. Moving a composite application to a particular runtime environment requires a distribution of contained services before deployment. Hence, application owners want to optimize the distribution of services in order to save time and money. A dynamic service provisioning, i.e. dynamic resource allocation and dynamic deployment of services in a distributed environment enables high flexibility and optimized usage of the infrastructure. In particular, optimized distributions of service instances can be created by taking (communication) dependencies between services into account. Correlating the service dependencies and a particular state of infrastructure at deployment time leads to a best qualified region in the infrastructure for the deployment.

Assuming a service Y that is called by another service X the approach hand determines whether an existing instance of Y can be used or a new instance need to be created close to the calling instance of service X. Hence, the current state of the infrastructure, i.e. available service instances as well as service dependencies are taken into account in order to reduce costs for data transfer and to avoid long waits on data. The approach at hand focus on the costs for running instances and costs for data transfer as users of a particular infrastructure are mostly interested in these aspects. However, the approach is suitable to cover further aspects.

The proposed method for dynamic provisioning of services has the following assumptions: (1) Instantiation needs to be automatic, especially on demand. A flexible number of instances run in parallel at a specific state of infrastructure but a maximum number of allowed instances is specified by the owner of the service. If an instance processes no request the instance is deleted immediately and the resource is reallocated. Note that a service can have its own instance management that is responsible for the creation of instances within the particular service instance. However, the approach at hand is based on service instances that are created within the infrastructure. (2) Dynamic addresses for services and a management component for dynamic addresses need to be supported by the infrastructure. Dynamic addresses are exclusively visible in the infrastructure as they cover deployed instances of services in a particular infrastructure. Conventional (static) addresses are visible everywhere and cover services ignoring concrete instances. A management component for dynamic addresses needs to provide the matching of static service addresses to a concrete dynamic service address covering a particular instance (cf. AWS Elastic IP-Addresses [1]).

In general, the approach at hand introduces the *closeness* of two service instances that can be measured at runtime. In particular, the closeness of services has static as well as dynamic aspects. Static aspects can be specified in an introduced context model of a service, i.e. the context model of a service X specify how close a partner service Y need to be. Dynamic aspects of the closeness need to be evaluated by the particular runtime environment. For supporting dynamic aspects the approach at hand introduces the betweenness of services covering the data transfer between services, i.e. its dynamic properties and takes the certain (dynamic) state of infrastructure into account.

A. Use Cases

The approach at hand introduces a *logical management level for provisioning* on top of an infrastructure. The introduced management level allows automatic allocation and reallocation of infrastructure resources as well as automatic calculation of distribution strategies. That means the user of a distributed runtime environment does not have to manually distribute his service instances on the infrastructure any longer. Furthermore advanced strategies

for deployment can be calculated as the current state of the dynamic infrastructure can be taken into account.

The approach is also applicable for deciding about *data vs. function shipping* as it balances data transfer against instance creation. In case the costs for data transfer between existing instances exceeds the costs for data transfer with a new instance the particular service is newly deployed close to the data receiver (cf. function shipping). Otherwise an existing instance can be used and the data need to be transferred to the existing instance (cf. data shipping). In summary, the costs for data transfer as well as locations of existing instances are used to decide about the shipping of data or functions, respectively.

The deployment of services on demand allows an easy mechanism for the *extension of existing applications*. That means, only additional services need to be deployed whereas existing instances of services contained in the application can remain unchanged if no modifications on the service's logic is needed. Finally, a context-aware deployment of services allows the *bundling of related compute-intensive services*. Different instances of such a service bundle can be distributed on different deployment regions in order to properly manage peak loads in the infrastructure.

II. CONTEXT MODEL

The approach at hand extends the specification of a service that is registered in a specific distributed runtime environment by a context model. Considering a service the context model specifies depending services as well as the context dependencies to these partner services including the particular degree of context. Specified *context dependencies* are directed, i.e. services S_1, …, S_n that are utilized by a service X are in context of service X but service X does not have to be in context of the services S_1, …, S_n. Concrete types of context dependencies are user-defined and need to be related to the classes of context provided by a certain infrastructure. In particular, matching introduced types of dependencies with provided classes of context allows the classification of the strength of dependencies between services.

A distributed environment is typically divided into regions, e.g. different server clusters. Based on such regions a distributed environment can provide *classes of context.* The approach at hand introduces the following conceptual classes of context:

0. Context-free, i.e. in context of no other resource
1. Very close a related resource Y, e.g. on the same server
2. Close to a related resource Y, e.g. in the same cluster
3. In the same closed world as a related resource Y, i.e. in the same infrastructure, e.g. cloud environment
4. In context of a related resource Y but resource Y is part of the open world (includes the closed world of class 3)

Example: Cloudy Composition Engine

This section presents a sample application implementing a composition engine that is responsible for service composition. For distributed execution the engine is modularized and each module creates a service. Figure 1 shows the modularization of the basic functionality of a composition engine. Furthermore, each module is related to a context level collecting modules of the same context class.

The navigator of a composition engine including the related composition logic creates the first module. Considering the navigating functionality the module is also responsible for the monitoring. The navigator communicates with a service invocation module that is responsible for requesting other services and the resolution of request parameters. That means the composition engine in Figure 1 exclusively manages data by reference but need to provide concrete data in service requests. However, a reference resolution system including data storage is responsible for the managing of data and data references [2]. The module expression evaluator, e.g. for XPath expressions [3] is essential for the composition engine. For example a navigator decides about control flow alternatives by evaluating an expression. That means the navigator needs to be informed about evaluation results. Hence, the evaluation result must be provided as concrete data and not by reference.

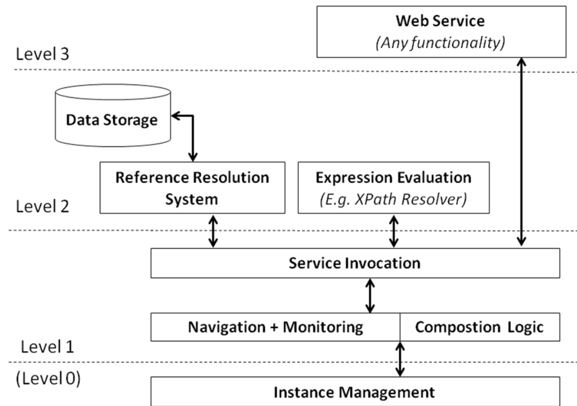


Figure 1 Modularized composition engine

As mentioned before, every module can have its own instance management. That means the instances of engine modules act independent from each other and independent from other modules. However, an instance of a module is probably used by multiple instances of another module. For example the same instance of the service invocation module can be used by multiple instances of the navigator. In summary, the approach at hand allows multiple instances of engine modules managing different instances of compositions. Conventional engines usually support one instance of the engine managing multiple instances of compositions.

Based on identified context levels for modules a context model can be created for the modularized composition engine. Figure 2 shows the context model for the navigator service in the modularized composition engine specified with TOSCA [4]. A rectangle represents a TOSCA node template including provided operations, i.e. including a specific node type. An arrow between two nodes represents a TOSCA relationship template referring to a particular relationship type. The context model introduces the following context

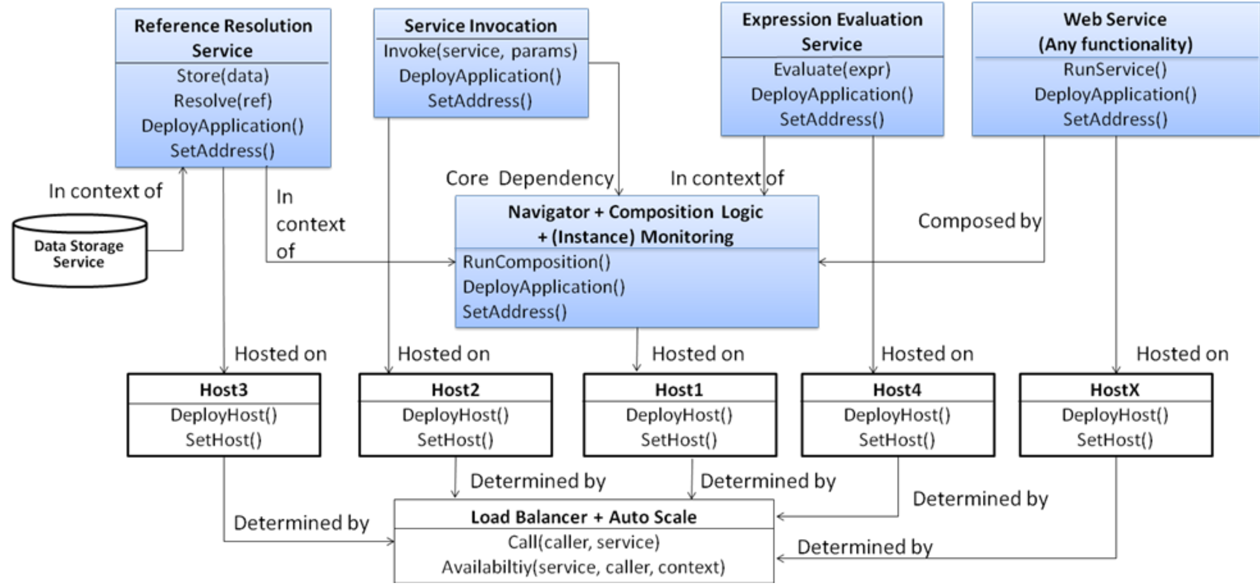


Figure 2 Context model for the modularized composition engine

dependencies as relationship types: core dependency (correlates with context class 1), "in context of" (correlates with context class 2), composition dependency (correlates with context class 3 and 4).

Based on the context model instances of the modules can be deployed as close to each other as required. In case a service composition needs to be instantiated the module "navigation" with the particular composition logic need to be considered at first. In case no existing instance of the navigator is available (e.g. because of work load) a new instance is deployed close to existing instances of depending service instances, e.g. close to existing instances of the modules "service invocation" and "expression evaluation". Furthermore, the classes of context specify how close a service instance needs to be located in order to create a valid distribution.

In summary the introduced modularized composition engine combined with dynamic provisioning allows to meet advanced requirements on composition engines in the cloud. In particular, cloud-aware engines must support multiple instances in the cloud infrastructure for providing elasticity. As described before the modularized composition engine introduced in this section easily allows additional instances of any required modules in case of scaling up.

III. Dynamic Provisioning Strategies

The method introduced in this section decides about the need for instantiation and determines the best location for deployment by balancing costs for data traffic against costs for running new instances. Assuming a service X that is called the method determines the best provisioning strategy for X based on the context model of X and the current state of the infrastructure. At first the method searches for an existing instance of X close to the caller. If such an instance does not exist a new instance is created close to the caller and close to other services that depend on service X.

The costs for data transfers are estimated based on previous instance runs considering the amount of data that need to be transferred as well as the probability of the occurrence of concrete data transfers. The costs for instances are abstracted, i.e. the service owner specifies a maximum number of instances that are allowed to run in parallel. For autonomously deciding about the creation of instances the infrastructure need to be aware of the maximum number of instance that are allowed to run in parallel. That means the infrastructure is allowed to create a new instance in case there is a need for instantiation and further instances can be deployed. Otherwise existing instances need to be sufficient. However, pay-per-use is still applicable and the owner of a service only has to pay for running instances but not necessarily for the maximum number of instances.

A. Betweenness

For each service X that sends data to a service Y the infrastructure provides the betweenness(X,Y). Analogous to context dependencies the betweenness is directed, i.e. the infrastructure does not necessarily provide the betweenness(Y,X). The betweenness of two services X and Y contains a size value and a probability value: The size of the betweenness specifies the rate of the dependency, i.e. the amount of data that is transferred from service X to service Y at runtime. The probability value of the betweenness specifies how many instances of service X effectively depend on service Y, i.e. how many data transfers effectively occur.

The infrastructure component that provides the betweenness needs to dynamically update the betweenness values. In particular the component needs to calculate the average of the amount of data over all numbers of instances. In case a new instance is running an infrastructure-internal monitoring component needs to pass relevant information about the new instance to the betweenness component.

However, the dynamism of the introduced betweenness (i.e. dynamic metrics) enables dynamic provisioning strategies.

Example: Distribution on Multiple Clouds

Figure 3 shows two deployment alternatives for an instance of service X that was called by a service A. On top of Figure 3 the context model of service X as well as the interesting part of the context model of service A is represented. Service X requests three other services S1, S2, and S3 that are located in the open world (cf. context class 4). Furthermore, service X is in context of service A but can also be located in the open world, i.e. no restrictions exist on the closeness of service A and X as well as on the closeness of service X and service S1 (or S2, S3). However, the betweenness decides about the location for the deployment of a new instance of service X.

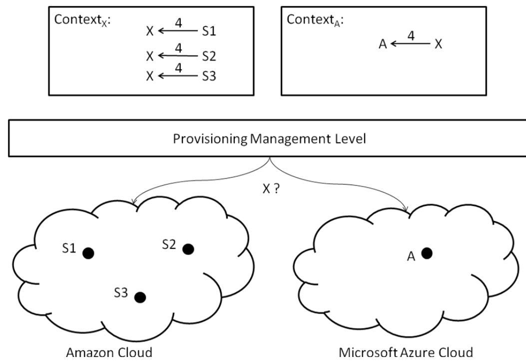


Figure 3 Deployment alternatives for service X that is called by service A

Let's assume a service X alternatively requesting service S1 or S2 and the following betweenness rates:

betweenness(A,X) = (5GB, 100%)
betweenness(X, S1) = (1MB, 48%)
betweenness(X, S2) = (5GB, 52%)

That means service A sends 5GB data to service X in 100% of the instances of A. Furthermore, service X sends 1MB data to service S1 in 48% of the instances of X, and service X sends 5GB data to service S2 in 52% of the instances of X. For determining the best location for X the particular amount of data transfer needs to be compared. At first absolute values for the betweenness rates are calculated by multiplying the size of the betweenness with its probability value. Afterwards the absolute values of services that are deployed in the same region are summed up. Finally, the maximum sum determines the region for deploying X. For the betweenness rates above the following comparison determines a location close to service A for the deployment of service X:

$$5000 * 1 > 1 * 0.48 + 5000 * 0.52$$

That means, the amount of data that need to be transferred from service A to service X is mostly probable higher than the amount of data that need to be transferred from service X to service S1 or S2. Therefore a location close to A is certainly the best location.

If service X additionally invokes a service S3 immediately after the execution of S2 (but not after the execution of S1) the following betweenness rates would determine another location for the deployment of service X:

betweenness(X, S1) = (1MB, 34%)
betweenness(X, S2) = (5GB, 66%)
betweenness(X, S3) = (5GB, 66%)

In particular more data need to be transferred from service X to the service S1, S2, and S3. Therefore the deployment close to the services S1, S2, and S3 is better than a deployment close to service A:

$$5000 * 1 < 1 * 0.34 + 5000 * 0.66 + 5000 * 0.66$$

Note that a context dependency related to class 3 between the services X and S1, X and S2 or X and S3 would lead to a deployment of X close to S1, S2, or S3 independent from betweenness rates. That means at first the context dependencies and its related context classes determine valid alternatives for the deployment. Afterwards, the comparison of betweenness rates determines the best alternative.

B. Automatic Scaling

Usually a load balancer distributes incoming calls of a service to available instances. Furthermore, a component supporting automatic scaling is possibly responsible for the creation of new service instances. The approach at hand combines the functionality of a load balancer with the functionality of an automatic scaling component in order to allow dynamic provisioning strategies. Note that the combined load balancing functionality does not equally distributes incoming calls to the existing instances but maximizes the closeness of the calling instance and the called instance. Hence, minimal costs for data transfer are ensured.

Figure 4 shows a pseudo code algorithm for the combined functionality of a load balancer and an automatic scaling component. Assuming a call of service X by a service A the algorithm searches for existing instances of X close to the calling instance of service A at first. That means the algorithm checks the location of the caller A. If the calling instance of service A is not part of the particular closed world the algorithm searches for an existing instance of X that can be located everywhere in the particular closed world, i.e. context class 4 is used for searching existing instances of X. If the caller is part of the particular closed world the algorithm searches for existing instances of X as close as possible to the calling instance of A, i.e. in the lowest possible context class. For each existing instance x_i that is found a user-defined condition for scaling up is checked, e.g. $cpu(x_i) < 70\%$. The condition is specified by the owner of service X. In case the condition evaluates to true the call is passed to the existing instance x_i. Otherwise the search for a suitable instance of X continues in the next context class as long as the highest context class, which is allowed by the context model of service A, is reached.

```
call( A, X)
1.    Determine the location locA of the caller A
   1.1  If locA is part of the closed world then context=1 and maxContext=4
   1.2  Else context=4 and maxContext=4
2.    Start with context, repeat until maxContext
   2.1  Search for instances xi of service X with context(xi,(A, context))=true
      2.1.1     For each instance xi that is found
         2.1.1.1  If cpu(xi) < 70%
                       Send request to the current instance xi and terminate
      2.1.2     If context= maxContext
         2.1.2.1  If scaleup
                       Inst=createInstance(X, A, locA) and send request to Inst
         2.1.2.2  Else
                       Send request to xi with cpu>=70% as close as possible to the calling instance of A
      2.1.3     Else
                   context++
```

Figure 4 Load balancing and automatic scaling

In case no suitable instance was found within the given context class a new instance is created if the actual number of instances of X does not exceeds the maximum number of allowed instances. In Figure 4 the flag *scaleup* indicates whether further instances are allowed. In case no further instances are allowed the existing instances need to be used even though the condition for scaling up evaluates to true.

C. Dynamic Distribution

If a new instance may to be created the best location for the new instance should be identified. Figure 5 shows a pseudo code algorithm determining the best location for deploying a new instance of service X that was called by a service A. At first a set of orientation points is collected. In case the caller A is part of the particular closed world the calling instance of A is part of the set of orientation points in order to allow the new instance of X to be as close as possible to the caller. However, if the caller is part of the open world and not part of the closed world the location of the caller cannot be taken into account. Instead, the set of orientation points exclusively contains existing instances of services S_i that are in context of the service X. That means the algorithm searches for existing instances of S_i at the certain state of infrastructure and selects the instances with the lowest possible context class to be an orientation point. Note that for each region only one existing instance is part of the set of orientation points.

Afterwards, the algorithm searches for services S_k that are in context of service X. In particular, for each orientation point the algorithm searches for instances s_k that are located as close as specified in the context model of X to the particular orientation point. Considering one orientation point and different classes of context leads a deployment alternative stored in a set Alt. Such a deployment alternative contains an orientation point as well as available instances of services S_k that are in the particular context of service X. Finally, all deployment alternatives are stored in a set Distr.

```
createInstance(X, A, locA)
1.  OP = {}, C = 1, repeat until C = 4 or OP ≠ {}
  1.1   For each service Si with context(X, Si)=C
    1.1.1 Search for an existing instance si
    1.1.2 if OP contains no sj in the same (sub-)region
          like si then OP = OP ∪ { si }
    1.2   C++
2 If locA is part of the closed world then OP = OP∪{A}
3 Distr = {}, For each op in OP
  3.1 Alt = { op }
  3.2 C = 1, repeat until C = 4
    3.2.1 For each service Sk with context(X, Sk)=C
          specified in ContextX
       3.2.1.1   Search for an instance sk with
                 (sk , (op, C))=true
       3.2.1.2   Alt = Alt ∪ { sk },
    3.2.2 C++
  3.3 Distr = Distr ∪ Alt
4 Max= 0, For each Alt ∈ Distr
  4.3 For each element { s1j , … , snj } in Alt and its
      related Services S1, …, Sn
    4.3.1 B = ∑^n_i=1 betweenness(X, Si).probability *
                  betweenness(X, Si).size
    4.3.2 If Max < B
       4.3.2.1   Max = B
       4.3.2.2   MaxLocation = location(sk) with sk ∈
                 Alt ∪ OP
5 Res=getResource(X, Maxlocation)
6 Deploy X on Res creating a new instance xi
7 Return the new instance xi
```

Figure 5 Creating an instance in the most qualified region

Afterwards the algorithm calculates the sum of the betweenness rates for each alternative and its contained services. The maximum sum indicates the alternative that is most qualified for inserting a new instance of X based on the current infrastructure state. Therefore, the location of the orientation point of the most qualified alternative indicates the best location for the new instance of X.

Example: Cloudy Composition Engine

Let's assume the state of an infrastructure as shown in Figure 6 providing two regions including sub-regions. In detail the infrastructure contains instances of services realizing a composition engine as well as composed services. Region 1 holds two instances of the navigator (including the composition logic) as well as other related engine services. The context model for the composition engine (cf. section II) specifies the need for deploying an instance of the service invocation component very closed to the navigator instances, i.e. in the same sub-region (cf. context class 1). Instances of the expression evaluator and the reference resolution system must be close to the navigator instance, i.e. in the same region (cf. context class 2). Composed services S1, S2, and S3 do not have to be located in the same region, i.e. composed services are part of the open world (cf. context class 4). However, Figure 6 shows composed services that are part of the current infrastructure, precisely part of region 2. These instances of S1, S2, and S3 are be used by navigator instances located in region 2 as well as by navigator instances located in region 1 as the context model allows such a distance.

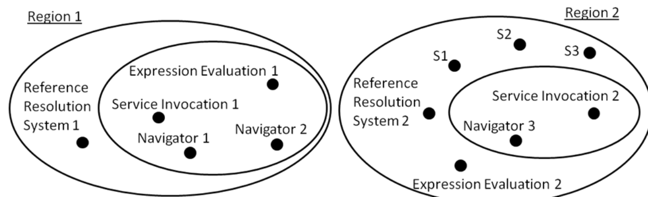


Figure 6 Distributed composition engine and composed services S1, S2, and S3

In case all existing instances of the navigator are busy at the current infrastructure state a new instance needs to be created. Assuming the location of the caller A outside of the closed world the algorithm in Figure 5 would detect two alternatives for the deployment of the new instance:

Alt_1 = {Service Invocation 1, Expression Evaluation 1, Reference Resolution System 1}

Alt_2 = {Service Invocation 2, Expression Evaluation 2, Reference Resolution System 2, S1, S2, S3}

The first alternative would locate the instance in the sub-region of region 1 whereas the second alternative would locate the instance in the sub-region of region 2. However, the second alternative would be selected for deployment as the services S1, S2, and S3 are part of the region 2 and therefore part of the sum of betweenness rates. That means a new instance of X would be deployed in region 2 as the related sum of the betweenness rates is expected to be higher than the sum of betweenness rates related to the deployment alternative in region 1. Obviously the deployment of service X in region 2 maximizes the expected internal data transfer and minimizes expected external data transfer to other regions.

IV. Conclusion

The approach at hand introduces a context model for services specifying the particular service, its partner services and the context dependencies between them. Furthermore, context dependencies are classified by relating the dependencies to context classes, i.e. regions provided by the infrastructure. The context model of a service is used to realize a dynamic provisioning of services based on the current state of infrastructure. In particular, the need for an instantiation as well as the best deployment region is identified by introduced algorithms.

For specifying the context model we propose TOSCA [4]. Considering an application covering multiple components TOSCA allows the specification of the application's structure and additional plans specifying how to build the application. The approach at hand uses the TOSCA structure model for the context model whereas TOSCA plans can be generated by interpreting the context model. For example, the classification of context dependencies can be related to an essential or probable deployment: That means depending services in context class 1 and 2 must be deployed together with a certain service. Depending services in context class 3 and 4 can have decrements of probable deployments as they must be close enough or provided by other applications for example. In the generated TOSCA plan the particular service is deployed at first, i.e. all services of the context model without a dependency to another service are deployed at first. Afterwards the depending services are deployed successively in context class 1 and 2. Other depending services can be deployed under certain conditions.

At the other hand generic plans can be used to realize an exposed context dependency in the context model. For example a BPMN process [5] implementing the dependency "hosted by" is shown in Figure 7. The plan is valid for any module Y contained in the context model of the modularized composition engine that is called at a specific time, i.e. a specific infrastructure state. At first existing instances of Y close to the caller are identified. If such an instance exists and scaling up is not necessary (i.e. the instance is not busy) the existing instance of Y is used to process the request. Otherwise, the best qualified infrastructure region for a new instance of Y is determined and a new instance is created in that region. Note that the plan in Figure 7 is generic, i.e. usable for determining the hosts of any service or module (cf. Figure 4). However, the task "Search best qualified region R" need to be aware about context dependencies in order to determine the best qualification (cf. Figure 5).

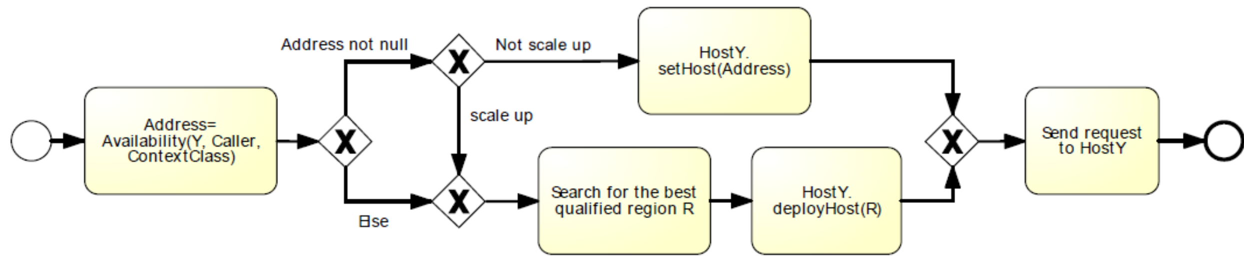


Figure 7 BPMN build plan for any module Y of the modularized composition engine

The introduced betweenness allows further information that can be related to instances of partner services. The betweenness reflects how much data in which probability need to be transferred between two instances. The probability value is dynamic information and provided by a monitoring component in the infrastructure. However, the betweenness can be extended to reflect other dynamic relevant information next to data transfer. The approach at hand uses the betweenness in order to choose the best qualified region among a set of alternative regions. In detail, the region with the highest sum of betweenness rates of contained instances is qualified as the best alternative.

A. Related Work

The optimization of the distribution of jobs across different resources was already an important problem in the grid [6], [7]. For example, [8] proposes a basic scheduling mechanism but assumes that jobs act independent from each other. [9] introduces an optimized resource allocation in distributed environments but is restricted on the consumed power. Furthermore, [10] introduces an algorithm for resource allocation that finds an optimal resource minimizing the execution time of the job and a corresponding schedule. That means, the resource allocation is optimized considering a specific schedule and its overall execution time. In contrast, the approach at hand does not use an underlying schedule but optimizes the resource allocation with focus on the closeness and expected costs for data transfer, i.e. the cost model does not exclusively focus on the overall execution time.

[11] introduces an approach optimizing resource allocation for distributed applications in a single cloud. In particular, the approach allows applications to adapt the resource allocation to the given workload and resource availability with respect to performance objectives. Hence, the approach investigates the properties adaptability and stability whereas the approach at hand investigates the property closeness. Furthermore, the approach at hand is not restricted to a single cloud but can be used considering multiple cloud environments. [12] generates possible distributions of a composite application deployment on different clouds and selects the best for deployment. The information of multiple models is correlated for calculating possible distributions and existing algorithms for optimization are reused. In particular, they focus on calculating distribution whereas the approach at hand focuses on the optimization. That means, possible distributions are generated based on information about the certain state of infrastructure and with respect to the required closeness. Finally, the best alternative is indicated by the highest betweenness.

The Amazon Cloud [1] provides many Web Services related to the approach at hand. For example the services auto scaling and load balancing support the automatic management of EC2 instances. The AWS CloudFoundation allows the specification and execution of deployment plans, i.e. a set of components that need to be deployed together. The automatic execution of deployment plans enables an automatic deployment. Finally, the Amazon Cloud provides regions and availability zones in the infrastructure suitable for defining classes of context.

References

[1] Amazon Web Services: http://aws.amazon.com/de/

[2] M. Wieland, K. Görlach, D. Schumm, and F. Leymann: Towards Reference Passing in Web Service and Workflow-Based Applications. EDOC 2009: 109-118

[3] J. Clark (Edt.), and S. DeRose (Edt.): XML Path Language (XPath) 1.0., W3C Standard, November 1999, http://www.w3.org/TR/1999/REC-xpath-19991116

[4] P. Lipton (Edt.), and S. Moser (Edt.): Topology and Orchestration Specification for Cloud Applications (TOSCA), Oasis Specification, October 2011, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca

[5] Business Process Model and Notation (BPMN) Version 2.0, OMG Standard, January 2011, http://www.omg.org/spec/BPMN/2.0/

[6] I. Foster, and C. Kesselman: The grid: bluepront for a new computing infrastructure. Morgan Kaufmann, 2004, ISBN 1558604758

[7] M. Caramia, and S. Giordani: Resource allocation in grid computing: an economic model. Transactions on Computer Research, 3(1), 2008

[8] R. P. Doyle: Model-based resource provisioning in a web service utility. In Proc. of USENIX Symposium on Internet Technologies and Systems, March 2003.

[9] S. U. Khan, and C. Ardil. Energy efficient resource allocation in distributed computing systems. In Proc. of WASET Int'l Conference on Distributed, High-Performance and Grid Computing, August 2009.

[10] K. Li: Job scheduling and processor allocation for grid computing on metacomputers. Journal of Parallel and Distributed Computing 65(11), 2005, pp. 1406-1418

[11] C. Lee, J. Suzuki, A. Vasilakos, Y. Yamamoto, and K. Oba: An evolutionary game theoretic approach to adaptive and stable application deployment in clouds. In Proc. Of the 2nd Workshop on Bio-inspired Algorithms for Distributed Systems, 2012, pp. 29-38

[12] F. Leymann, C. Fehling, R. Mietzner, A. Nowak, and S. Dustdar: Moving Applications to the Cloud: An Approach based on Application Model Enrichment. In: International Journal of Cooperative Information Systems (IJCIS). Vol. 20(3), World Scientific, 2011