An Interoperable and Self-adaptive Approach for SLA-based Service Virtualization in Heterogeneous Cloud Environments

A. Kertesz[a], G. Kecskemeti[a], I. Brandic[b]

[a]*MTA SZTAKI, H-1518 Budapest, P.O. Box 63, Hungary*
[b]*Vienna University of Technology, 1040 Vienna, Argentinierstr. 8/181-1, Austria*

Abstract

Cloud computing is a newly emerged computing infrastructure that builds on the latest achievements of diverse research areas, such as Grid computing, Service-oriented computing, business process management and virtualization. An important characteristic of Cloud-based services is the provision of non-functional guarantees in the form of Service Level Agreements (SLAs), such as guarantees on execution time or price. However, due to system malfunctions, changing workload conditions, hard- and software failures, established SLAs can be violated. In order to avoid costly SLA violations, flexible and adaptive SLA attainment strategies are needed. In this paper we present a self-manageable architecture for SLA-based service virtualization that provides a way to ease interoperable service executions in a diverse, heterogeneous, distributed and virtualized world of services. We demonstrate in the paper that the combination of negotiation, brokering and deployment using SLA-aware extensions and autonomic computing principles are required for achieving reliable and efficient service operation in distributed environments.

Keywords: Cloud computing, Service virtualization, SLA negotiation, Service Brokering, On-demand deployment, Self-management.

1. Introduction

The newly emerging demands of users and researchers call for expanding service models with business-oriented utilization (agreement handling) and support for human-provided and computation-intensive services [6]. Though Grid Computing [21] has succeeded in establishing production Grids serving various user communities, and both Grids and Service Based Applications (SBAs) already provide solutions for executing complex user tasks, they are still lacking non-functional guarantees. Providing guarantees in the form of Service Level Agreements (SLAs) is also highly studied in Grid Computing [30, 4, 9], but they have failed to be commercialized and adapted for the business world.

Cloud Computing [6] is a novel infrastructure that focuses on commercial resource provision and virtualization. These infrastructures are also represented by services that are not only used but also installed, deployed or replicated with the help of virtualization. These services can appear in complex business processes, which further complicates the fulfillment of SLAs. For example, due to changing components, workload and external conditions, hardware-, and software failures, already established SLAs may be violated. Frequent user interactions with the system during SLA negotiation and service executions (which are usually necessary in case of failures), might turn out to be an obstacle for the success of Cloud Computing. Thus, there is demand for the development of SLA-aware Cloud middleware, and application of appropriate strategies for autonomic SLA attainment. Despite business-orientation, the applicability of Service-level agreements in the Cloud field is rarely studied, yet [41]. Most of the existing works address provision of SLA guarantees to the consumer and not necessarily the SLA-based management of loosely coupled Cloud infrastructure. In such systems, it is hard to react to unpredictable changes and localise, where the failures have happen exactly, what is the reason for the failure and which reaction should be taken to solve the problem. Such systems are implemented in a proprietary way, making it almost impossible to exchange the components (e.g. use another version of the broker).

Autonomic Computing is one of the candidate technologies for the implementation of SLA attainment strategies. Autonomic systems require high-level guidance from humans and decide, which steps need to be done to keep the system stable [19]. Such systems constantly adapt themselves to changing environmental conditions. Similar to biological systems (e.g. human body) autonomic systems maintain their state and adjust operations con-

Email addresses: `keratt@sztaki.hu` (A. Kertesz), `kecskemeti@sztaki.hu` (G. Kecskemeti), `ivona@infosys.tuwien.ac.at` (I. Brandic)

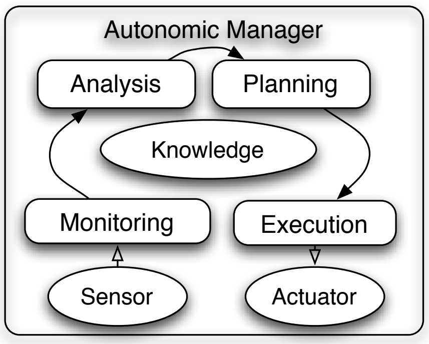

Figure 1: General architecture of an autonomic system.

sidering their changing environment. Usually, autonomic systems comprise one or more managed elements e.g. QoS elements.

An important characteristic of an autonomic system is an intelligent closed loop of control. As shown in Figure 1, the Autonomic Manager (AM) manages the element's state and behavior. It is able to sense state changes of the managed resources and to invoke appropriate set of actions to maintain some desired system state. Typically control loops are implemented as MAPE (monitoring, analysis, planning, and execution) functions [19]. The *monitor* collects state information and prepares it for the *analysis*. If deviations to the desired state are discovered during the analysis, the *planner* elaborates change plans, which are passed to the *executor*. For the successful implementation of the autonomic principles to loosely coupled SLA-based distributed system management, the failure source should be identified based on violated SLAs, and firmly located considering different components of the heterogeneous middleware (virtualization, brokering, negotiation, etc. components). Thus, once the failure is identified, Service Level Objectives (SLOs) can be used as a guideline for the autonomic reactions.

In this paper we propose a novel holistic architecture considering resource provision using a virtualization approach combined with business-oriented utilization used for SLA agreement. Thus, we provide an SLA-coupled infrastructure for on demand service provision based on SLAs. First we gather the requirements of a unified service architecture, then present our solution called SLA-based Service Virtualization (SSV) built on agreement negotiation, brokering and service deployment combined with business-oriented utilization. We further examine this architecture and investigate, how previously introduced principles of Autonomic Computing appear in the basic components of the architecture in order to cope with changing user requirements and on demand failure handling. After presenting our proposed solution, we evaluate the performance gains of the architecture through a more computational-intensive biochemical use case.

The main contributions of this paper include: (i) the *presentation* of the novel loosely coupled architecture for the SLA-based Service Virtualization and on-demand resource provision, (ii) the description of the architecture including *meta-negotiation, meta-brokering, brokering and automatic service deployment* with respect to the principles of autonomic computing, and (iii) the *evaluation* of the SSV architecture with a biochemical case study using simulations.

In the following section we summarize related works. Then, in Section 3, we provide the requirement analysis for autonomous behavior in the SSV architecture through two scenarios. Afterwards, in Section 4, we introduce the SSV architecture, while in Section 5 the autonomous operations of the components are detailed. In Section 6 we present the evaluation of the SSV architecture with a biochemical case study in a heterogeneous simulation environment. Finally, Section 7 concludes the paper.

2. Related work

Though Cloud Computing is highly studied, and a large body of work has been done trying to define and envision the boundaries of this new area, the applicability of Service-level agreements in the Cloud and in a unified distributed middleware is rarely studied. The envisioned framework in [15] proposes a solution to extend the web service model by introducing and using semantic web services. The need for SLA handling, brokering and deployment also appears in this vision, but they focus on using ontology and knowledge-based approaches. Most of related works consider virtualization approaches [17, 32, 23] without taking care of agreements or concentrate on SLA management neglecting the appropriate resource virtualizations [35, 9]. Works presented in [31, 29] discuss incorporation of SLA-based resource brokering into existing Grid systems, but they do not deal with virtualization. Venugopal et al. propose a negotiation mechanism for advance resource reservation using the alternate offers protocol [40], however, it is assumed that both partners understand the alternate offers protocol. Lee et al. discusses application of autonomic computing to the adaptive management of Grid workflows [25] with MAPE (Monitoring, Analysis, Planning, Execution) decision making [19], but they also neglect deployment and virtualization. The work by Van et. al. [38] studied the applicability of the autonomic computing to Cloud-like systems, but they almost exclusively focus on virtualization issues like VM packing and placement.

In [7], Buyya et al. suggests a Cloud federation oriented, just in time, opportunistic and scalable application services provisioning environment called InterCloud. They envision utility oriented federated IaaS systems that are able to predict application service behavior for intelligent down and up-scaling infrastructures. Then, they list the research issues of flexible service to resource mapping, user and resource centric QoS optimization, integration with in-house systems of enterprises, scalable monitoring of system components. Though they address self-management and SLA handling, the unified utilization of other distributed systems are not studied. Recent Cloud Computing projects, e.g. Reservoir [33] and OPTIMIS [11], address specific research topics like Cloud interoperability, but they do not consider autonomous SLA management across diverse distributed environments. Comparing the currently available solutions, autonomic principles are not implemented in a adequate way because of they are lacking an SLA-coupled Cloud infrastructure, where failures and malfunctions can be identified using well defined SLA contracts.

Regarding high-level service brokering, LA Grid [34] developers aim at supporting grid applications with resources located and managed in different domains. They define broker instances, each of them collects resource information from its neighbors and save the information in its resource repository. The Koala grid scheduler [16] was redesigned to inter-connect different grid domains. They use a so-called delegated matchmaking (DMM), where Koala instances delegate resource information in a peer-2-peer manner. Gridway introduced a Scheduling Architectures Taxonomy to form a grid federation [39, 24], where Gridway instances can communicate and interact through grid gateways. These instances can access resources belonging to different Grid domains. Comparing the previous approaches, we can see that all of them use high level brokering that delegate resource information among different domains, broker instances or gateways. These solutions are almost exclusively used in Grids, they cannot co-operate with different brokers operating in pure service-based or Cloud infrastructures.

Current service deployment solutions do not leverage their benefits on higher level. For example the Workspace Service (WS) [17] as a Globus incubator project supports wide range of scenarios involving virtual workspaces, virtual clusters and service deployment from installing a large service stack to deploy a single WSRF service if the Virtual Machine (VM) image of the service is available. It is designed to support several virtual machines. The XenoServer open platform [32] is an open distributed architecture based on the XEN virtualization technique aiming at global public computing. The platform provides services for server lookup, registry, distributed storage and a widely available virtualization server. Also the VMPlants [23] project proposes an automated virtual machine configuration and creation service which is heavily dependent on software dependency graphs, but this project stays within cluster boundaries.

3. Use Case and requirements for SLA-coupled autonomic Cloud middleware

Deployment and runtime management in cloud-like environments aim at providing or modifying services in a dynamic way according to temporal, spatial or semantic requirements. Among many other purposes, it has also strong relation with adaptation and self-management. To gather the requirements for an autonomic service virtualization environment we refer to motivating scenarios.

3.1. Requirement analysis for a transparent, autonomic service virtualization

The aim of this section is to investigate the requirements of the self-management aspects of runtime management of service virtualization. Temporal provision of services requires certain infrastructure features that we classified into three groups. There must be

- a **negotiation phase** where it is specified, which negotiation protocols can be used, which kind of service has to be invoked, what are the non-functional conditions and constraints (temporal availability, reliability, performance, cost, etc.);
- a **brokering phase** which selects available resources that can be allocated for providing the services. These resources can be provided in many ways: Clouds (virtualized resources configured for a certain specification and service level guarantees), clusters or local Grids (distributed computing power with limited service level guarantees) or volunteer computing resources (no service level guarantees at all).
- Finally, during the **deployment phase** services have to be prepared for use on the selected resources in an automatic manner.

In the following we refer to two motivating scenarios to gather requirements from, and exemplify the need for an autonomous service virtualization solution.

Scenario 1. There are certain procedures in various activities that may require specific services and resources in an ad-hoc, temporal way. Generally, there is no reason to provide these services in a static 24/7 manner with performance guarantees, instead, these services should be created and decommissioned in a dynamic, on-demand way for the following reasons:

- These tasks represent fundamentally different computations that cannot be re-used or composed, potentially not even overlapped, e.g. air tunnel simulation, crash test analysis, various optimization procedures, and so on. These services must be provided independently from each other in a well defined and disjoint time frame.
- There is no need to dedicate certain resources to these activities as they occur rarely or at least infrequently. For example, resources used by the optimization services, can be re-used for other purposes if they are not required at the foreseeable future.

Scenario 2. Similarly to the previous case, if we think of using real life products, we also face problems that require an autonomic service management infrastructure. For example, in a car navigation scenario there are services that do not need to or cannot be provided in a static, permanent way. Instead, these services should be created and decommissioned in an adaptive, dynamic or on-demand way for the following reasons:

- There is no need to dedicate certain resources to these activities as they occur rarely or at least infrequently. As an example this happens when an accident causes a traffic jam that causes all the surrounding navigation systems to request services from the GPS unit provider. This situation however does not last longer than the last car moves away from the problematic area with the help of the unit.
- It is possible that certain services are needed upon a certain event e.g., only in case testing phase do not complete successfully specific simulation services have to be invoked.
- As some ad-hoc services may be mobile, they cannot be assumed constant. E.g., navigation systems offer services within the ad-hoc network and the network splits to two disjunct parts, then the availability of the offered services should be ensured in both parts.
- In certain cases, dynamic deployment of a service that is not available locally may be necessary. For example, the GPS unit is used to broadcast live information by the customer (like a radio transmission) towards the navigation system provider that might not have enough bandwidth to distribute the content towards its other users. Therefore the provider deploys a repeating service right next to the source. This service will serve the consumers and even enable re-encoding the transmission for the different needs of the different users, meanwhile demanding small bandwidth and processing power from the broadcaster itself.

4. The autonomic, SLA-based Service Virtualization architecture

According to the requirements introduced in the previous section, here we present a unified service architecture that builds on three main areas: agreement negotiation, brokering and service deployment using virtualization. We suppose that service providers and service consumers meet on demand and usually do not know about the negotiation protocols, document languages or required infrastructure of the potential partners. First we introduce the general architecture naming the novelties and open issues, then we detail the aforementioned three main areas with respect to the shown architecture. Figure 2 shows our proposed, general architecture called SLA-based Service Virtualization (SSV).

4.1. The SSV architecture and the roles of its components

The main components of the architecture and their roles are gathered in Table 1.

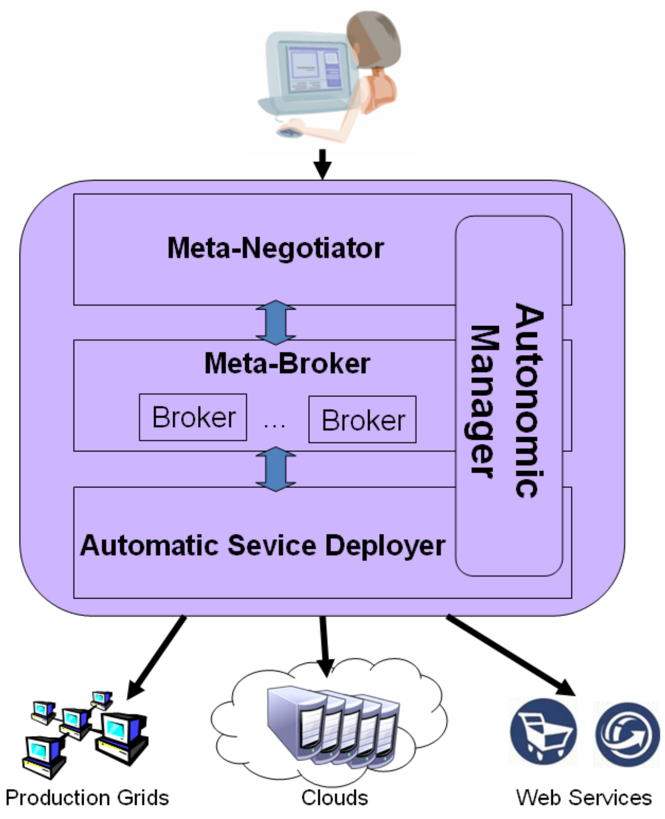

Figure 2: SSV architecture.

Table 1: Roles in SSV architecture

Abbreviation	Role	Description
U	User	A person, who wants to use a service
MN	Meta-Negotiator	A component that manages Service-level agreements. It mediates between the user and the Meta-Broker, selects appropriate protocols for agreements; negotiates SLA creation, handles fulfillment and violation.
MB	Meta-Broker	Its role is to select a broker that is capable of deploying a service with the specified user requirements.
B	Broker	It interacts with virtual or physical resources, and in case the required service needs to be deployed it interacts directly with the ASD.
ASD	Automatic Service Deployment	It installs the required service on the selected resource on demand
S	Service	The service that users want to deploy and/or execute
R	Resource	Physical machines, on which virtual machines can be deployed/installed

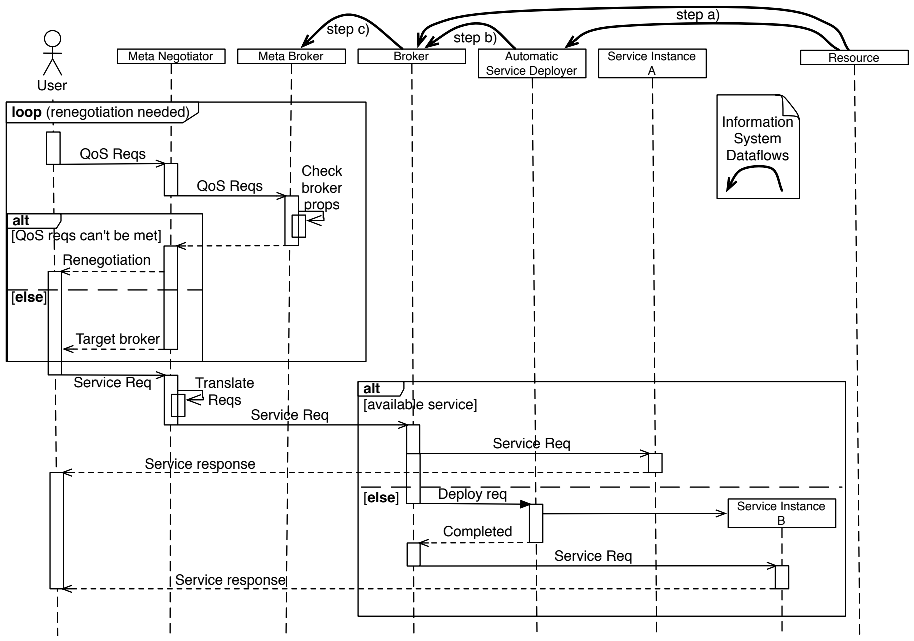

Figure 3: Component interactions in the SSV architecture.

The sequence diagram in Figure 3 reveals the interactions of the components and the utilization steps of the architecture, which are detailed in the following:

- User starts a negotiation for executing a service with certain QoS requirements (specified in a Service Description (SD) with an SLA)
- MN asks MB, if it could execute the service with the specified requirements
- MB matches the requirements to the properties of the available brokers and replies with an acceptance or a different offer for renegotiation
- MN replies with the answer of MB. Steps 1-4 may continue for renegotiations until both sides agree on the terms (to be written to an SLA document)
- User calls the service with the SD and SLA
- MN passes SD and the possibly transformed SLA (to the protocol the selected broker understands) to the MB
- MB calls the selected Broker with SLA and a possibly translated SD (to the language of the Broker)
- The Broker executes the service with respect to the terms of the SLA (if needed deploys the service before execution)
- In the final step the result of the execution is reported to the Broker, the MB, the MN, finally to the User (or workflow engine)

Note that this utilization discussion does not reflect cases when failures occur in the operational processes of the components, when local adaptation is required. This sequence represents the ideal execution flow of the SSV architecture. In this case there is no autonomic behaviour is needed, however in the next section we discuss and head towards the autonomous components and their effects on this ideal sequence (see tables 2, 3 and 4 for details).

While serving requests the architecture also processes background tasks that are not in the ideal execution path, these are also presented on Figure 3. The following background tasks are information collecting procedures that provide accurate information about the current state of the infrastructure up to the meta-brokering level:

- ASD monitors the states of the virtual resources and deployed services (step a)
- ASD reports service availability and properties to its Broker (step b)
- All Brokers report available service properties to the MB (step c)

4.2. Autonomically managed service virtualization

The previously presented SSV architecture and the detailed utilization steps show that agreement negotiation, brokering and service deployment are closely related and each of them requires extended capabilities in order to interoperate smoothly. Nevertheless each part represents an open issue, since agreement negotiation, SLA-based service brokering and on-demand adaptive service deployment are not supported by current solutions in diverse distributed environments. In this subsection we focus on illustrating how autonomic operations appear in the components of the SSV architecture. Figure 4 shows the autonomic management interfaces and connections of the three main components: agreement negotiation, brokering and service deployment.

We distinguish three types of interfaces in our architecture: the *job management interface*, the *negotiation inter-*

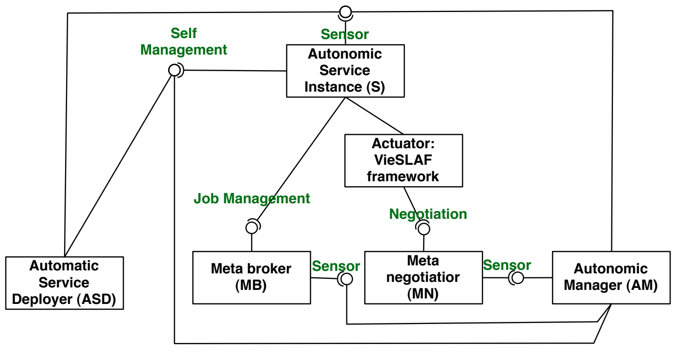

Figure 4: Autonomic components in the SSV architecture.

face and the *self-management interface*. Negotiation interfaces are typically used by the monitoring processes of brokers and meta-brokers during the negotiation phases of the service deployment process. Self-management is needed to re-negotiate established SLAs during service execution. The negotiation interface implements negotiation protocols, SLA specification languages, and security standards as stated in the meta-negotiation document (see Figure 5).

Job management interfaces are necessary for the manipulation of services during execution, for example for the upload of input data, or for the download of output data, and for starting or cancelling job executions. Job management interfaces are provided by the service infrastructure and are automatically utilized during the service deployment and execution processes.

In the next section we will focus on the self-management interface. The *Autonomic manager* in the SSV architecture is an abstract component, that specifies how self-management is carried out. (Later on use cases in Figure 9 will reflect, how the abstract Autonomic Manager is realized and used in the three components of the architecture.) All components of the architecture is notified about the system malfunction through appropriate sensors (see Figure 4). This interface specifies operations for sensing changes of the desired state and for reacting to that changes. Sensors can be activated using some notification approach (e.g. implemented by the WS-Notification standard). Sensors may subscribe for specific topic, e.g. violation of the execution time. Based on the measured values as demonstrated in [5] notifications are obtained, if execution time is violated or seems to be violated very soon. After the activation of the control loop, i.e. propagation of the sensed changes to the appropriate component, the service actuator reacts and invokes proper operations, e.g. migration of resources. An example software actuator used for the meta-negotiations is the VieSLAF framework [5], which bridges between the incompatible SLA templates by executing the predefined SLA mappings. Based on various malfunction cases, the autonomic manager propagates the reactions to the *Meta negotiatiator*, *Meta-broker* or *Automatic Service Deployer*. We discuss the closed loop of control using the the example with meta-negotiations in the next section.

5. Required components to realize the autonomic SSV architecture

In this section we detail three main categories, where the basic requirements of SLA-based service virtualization arise. We place these areas in the SSV architecture shown in Figure 2, and detail the related parts of the proposed solution. We also emphasize the interactions among these components in order to build one coherent system.

In our proposed approach, users describe the requirements for an SLA negotiation on a high level using the concept of meta-negotiations. During the meta-negotiation only those services are selected, which understand specific SLA document language and negotiation strategy or provide a specific security infrastructure. After the meta-negotiation process, a meta-broker selects a broker that is capable of deploying a service with the specified user requirements. Thereafter, the selected broker negotiates with virtual or physical resources using the requested SLA document language and using the specified negotiation strategy. Once the SLA negotiation is concluded, service can be deployed on the selected resource using the virtualization approach.

In subsection 5.1 we discuss the meta-negotiation component and detail the autonomic negotiation bootstrapping and service mediation scenario. In subsection 5.2 we discuss the brokering functionalities of SSV and present a self-management scenario for dealing with broker failures. Then, in subsection 5.3 we discuss autonomic service deployment and virtualization for handling resource and service failures. Finally, in subsection 5.4 we discuss the autonomic capabilities of each SSV layer through representative use cases.

5.1. Agreement negotiation

Prior Cloud infrastructures, users, who did not have access to supercomputers or did not have local clusters big enough, had to rely on Grid systems to execute services requiring high performance computations. In such environments, users had to commit themselves to dedicated Grid portals to find appropriate resources for their services, and on-demand execution was very much dependent on the actual load of the appropriate Grid. Service providers and consumers had to communicate using proprietary negotiation formats supported by the particular portal limiting the number of services a consumer may negotiate with. Nowadays, with the emergence of Clouds, service providers and consumers need to meet each other dynamically and on demand. Novel negotiation strategies and formats are necessary supporting the communication dynamics of the present day service invocations.

Before committing themselves to an SLA, the user and the provider may enter into negotiations that determine the definition and measurement of user QoS parameters, and the rewards and penalties for meeting and violating them respectively. The term negotiation strategy represents the logic used by a partner to decide which provider or consumer satisfies his needs best. A negotiation protocol represents the exchange of messages during the negotiation process. Recently, many researchers have proposed different protocols and strategies for SLA negotiation in Grids [30]. However, these not only assume that the parties to the negotiation understand a common protocol but also assume that they share a common perception about the goods or services under negotiation. In reality however, a participant may prefer to negotiate using certain protocols for which it has developed better strategies, over others. Thus, the parties to a negotiation may not share the same understanding that is assumed by the earlier publications in this field.

In order to bridge the gap between different negotiation protocols and scenarios, we propose a so-called meta-negotiation architecture [4]. *Meta-negotiation* is needed by means of a meta-negotiation document where participating parties may express: the pre-requisites to be satisfied for a negotiation, for example a specific authentication method required or terms they want to negotiate on (e.g. time, price, reliability); the negotiation protocols and document languages for the specification of SLAs that they support; and conditions for the establishment of an agreement, for example, a required third-party arbitrator. These documents are published into a searchable registry through which participants can discover suitable partners for conducting negotiations. In our approach, the participating parties publish only the protocols and terms while keeping negotiation strategies hidden from potential partners.

The participants publishing into the registry follow a common document structure that makes it easy to discover matching documents (as shown in Figure 5). This document structure consists of the following main sections: Each document is enclosed within the $\langle meta{-}negotiation \rangle$ ... $\langle /meta - negotiation \rangle$ tags. The document contains an $\langle entity \rangle$ element defining contact information, organization and a unique ID of the participant. Each meta-negotiation comprises three distinguishing parts, namely *pre-requisites, negotiation and agreement* as described in the following paragraph.

```
<meta-negotiation ...>
  ...
 <pre-requisite>
  <security>
   <authentication value="GSI" location="uri"/>
  </security>
  <negotiation-terms>
   <negotiation-term name="beginTime"/>
   <negotiation-term name="endTime"/>
    ...
  </negotiation-terms>
</pre-requisite>
<negotiation>
 <document name="WSLA" value="uri" .../>
 <protocol name="alternateOffers"
   schema="uri" location="uri" .../>
</negotiation>
<agreement>
 <confirmation name="arbitrationService" value="uri"/>
</agreement>
</meta-negotiation>
```

Figure 5: Example Meta Negotiation Document

As shown in Figure 5, prerequisites define the role a participating party takes in a negotiation, the security credentials and the negotiation terms. For example, the security element specifies the authentication and authorization mechanisms that the party wants to apply before starting the negotiation process. For example, the consumer requires that the other party should be authenticated through the Grid Security Infrastructure (GSI) [12]. The negotiation terms specify QoS attributes that a party is willing to negotiate and are specified in the $\langle negotiation - term \rangle$ element. As an example, the negotiation terms of the consumer are beginTime, endTime, and price. Details about the negotiation process are defined within the $\langle negotiation \rangle$ element. Each document language is specified within $\langle document \rangle$ element. Once the negotiation has concluded and if both parties agree to the terms, then they have to sign an agreement. This agreement may be verified by a third party organization or may be lodged with another institution who will also arbitrate in case of a dispute. Figure 6 emphasizes a meta-negotiation infrastructure embedded into the SSV architecture as proposed in Figure 2. In the following we explain the Meta-Negotiation infrastructure.

The registry is a searchable repository for meta-negotiation documents that are created by the participants. The meta-negotiation middleware facilitates the publishing of the meta-negotiation documents into the registry and the integration of the meta-negotiation framework into the existing client and/or service infrastructure, including, for example, negotiation or security clients. Besides being as a client for publishing and querying meta-negotiation documents (steps 1 and 2 in Figure 6), the middleware delivers necessary information for the existing negotiation clients, i.e. information for the establishment of the negotiation sessions (step 4, Figure 6) and information necessary to start a negotiation (step 5 in Figure 6).

5.2. Service brokering

To deal with heterogeneity and the growing number of services, special purpose brokers (for human-provided or computation-intensive services) or distributed broker instances should be managed together in an interoperable way with a higher-level mediator, which is able to distribute the load of user requests among diverse infrastructures. This high-level mediator (which we call a meta-broker) component in a unified service infrastructure needs to monitor the states of the services and the performances of these brokers, since brokers may have various properties, which should be expressed with a general description language known by service consumers. These properties may include static ones, some of which are specialized for

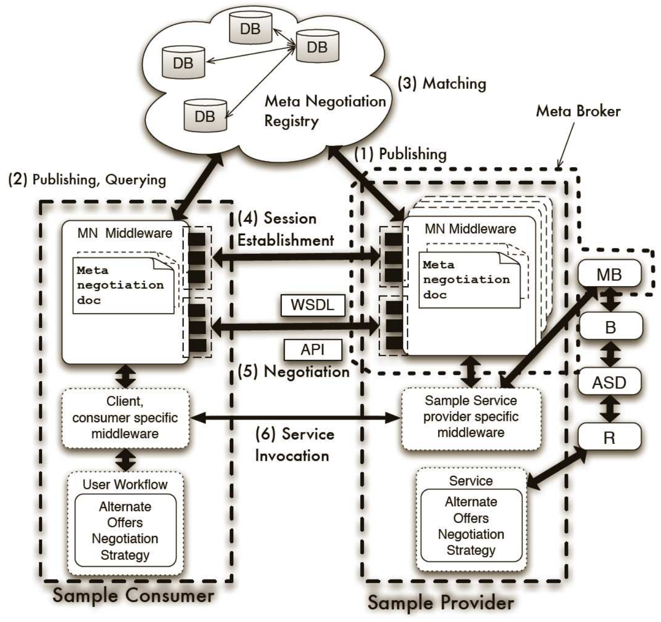

Figure 6: Meta-negotiation in the SSV architecture.

managing human-provided services, others for computing-intensive or data-intensive services; and dynamic ones, e.g. if two brokers are managing the same type of services, some of these may have longer response times, less secure or more reliable. Information on the available services (e.g. type, costs, amount) also belongs to this category, since it changes over time. The broker properties should be stored and updated in a registry accessible by higher-level managers. The update intervals of broker state changes and the amount of data transferred should also be set automatically, with respect to the number of available services and utilized brokers. Too frequent updates could lead to an unhealthy state and rare updates cause higher uncertainty and inefficient management. Therefore this registry should be more like a local database that makes this higher level brokering able to decide, which broker could provide the fittest service (according to the consumer requirements and SLA terms).

Self-adaptability also appears at this level: generally a bit higher uncertainty exists in broker selection compared to service selection, since the high dynamicity of the broker (and forwarded, filtered service) properties and availability cause volatility in the information available at this level. To cope with this issue, several policies could be defined by sophisticated predicting algorithms, machine learning techniques or random generator functions. Load balancing among the utilized brokers should also be taken into account during broker selection. Finally basic fault tolerant operations, such as re-selection on broker failure or malfunctioning also need to be handled.

In our proposed SSV architecture Brokers (B) are the basic components that are responsible for finding the required services deployed on a specific infrastructure with the help of the ASD. This task requires various activities, such as service discovery, matchmaking and interactions with information systems, service registries, repositories. There are several brokering solutions both in Grid [22] and SOAs[26], but agreement support is still an open issue. In our architecture brokers need to interact with ASDs and use adaptive mechanisms in order to fulfill the agreement.

A higher-level component is also responsible for brokering in our architecture: the Meta-Broker (MB) [20]. *Meta-brokering* means a higher level resource management that utilizes existing resource or service brokers to access various resources. In a more generalized way, it acts as a mediator between users or higher level tools (e.g. negotiators or workflow managers) and environment-specific resource managers. The main tasks of this component are: to *gather* static and dynamic broker properties (availability, performance, provided and deployable services, resources, and dynamic QoS properties related to service execution), to *interact* with MN to create agreements for service calls, and to *schedule* these service calls to lower level brokers, i.e. match service descriptions (SD) to broker properties (which includes broker provided services). Finally the service call needs to be *forwarded* to the selected broker.

Figure 7 details the Meta-Broker (MB) architecture showing the required components to fulfill the above mentioned tasks. Different brokers use different service or resource specification descriptions for understanding the user request. These documents need to be written by the users to specify all kinds of service-related requirements. In case of resource utilization in Grids, OGF [28] has developed a resource specification language standard called JSDL [1]. As the JSDL is general enough to describe jobs and services of different grids and brokers, this is the default description format of MB. The *Translator* component of the Meta-Broker is responsible for translating the resource specification defined by the user to the language of the appropriate resource broker that MB selects to use for a given call. These brokers have various features for supporting different user needs, therefore an extendable Broker Property Description Language (BPDL) [20] is needed to express metadata about brokers and their offered services. The *Information Collector* (IC) component of MB stores the data of the reachable brokers and historical data of the previous submissions. This information shows whether the chosen broker is available, or how reliable its services are. During broker utilization the successful submissions and failures are tracked, and regarding these events a rank is modified for each special attribute in the BPDL of the appropriate broker (these attributes were listed above). In this way, the BPDL documents represent and store the dynamic states of the brokers. In order to support load balancing, there is an *IS Agent* (IS refers to Information System) reporting to IC, which regularly checks the load of the underlying resources of each connected broker, and store this data. It also communicates with the ASDs, and receives up-to-date data about

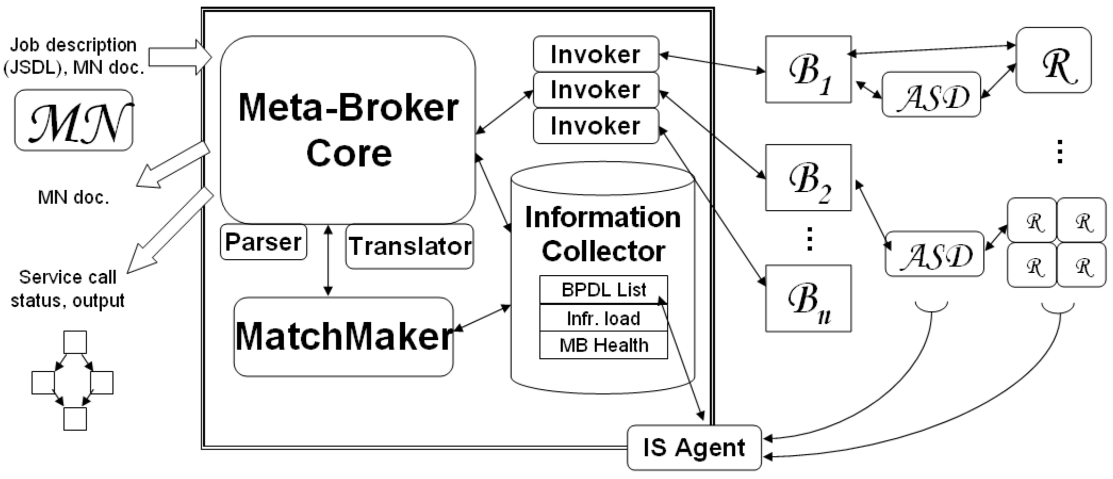

Figure 7: Meta-Broker in the SSV architecture.

the available services and predicted invocation times (that are used in the negotiations). The matchmaking process consists of the following steps: The *MatchMaker* (MM) compares the received descriptions to the BPDL of the registered brokers. This selection determines a group of brokers that can provide the required service. Otherwise the request is rejected. In the second phase the MM counts a rank for each of the remaining brokers. This rank is calculated from the broker properties that the IS Agent updates regularly, and from the service completion rate that is updated in the BPDL for each broker. When all the ranks are counted, the list of the brokers is ordered by these ranks. Finally the first broker of the priority list is selected, and the *Invoker* component forwards the call to the broker.

As previously mentioned, three main tasks need to be done by MB. The first, namely the information gathering, is done by the IS Agent, the second one is negotiation handling and the third one is service selection. They need the following steps: During the negotiation process the MB interacts with MN: it receives a service request with the service description (in JSDL) and SLA terms (in MN document) and looks for a deployed service reachable by some broker that is able to fulfill the specified terms. If a service is found, the SLA will be accepted and the and MN notified, otherwise the SLA will be rejected. If the service requirements are matched and only the terms cannot be fulfilled, it could continue the negotiation by modifying the terms and wait for user approval or further modifications.

In the following we demonstrate, how principles of autonomic computing are applied to this level of the SSV. Brokers are the basic components that are responsible for finding the required services with the help of ASD. This task requires various activities, such as service discovery, matchmaking and interactions with information systems and service registries. In our architecture brokers need to interact with ASD and use adaptive mechanisms in order to fulfill agreements.

5.3. Service deployment and virtualization

Automatic service deployment (ASD) builds on basic service management functionalities. It provides the dynamics to service based applications (SBAs) – e.g. during the SBA‘s lifecycle services can appear and disappear (because of infrastructure fragmentation, failures, etc.) without the disruption of the SBA's overall behavior. Service deployment process is composed of 8 steps: (*i*) *Selection* of the node where the further steps will take place; (*ii*) *Installation* of the service code; (*iii*) *Configuration* of the service instance; (*iv*) *Activation* of the service to make it public; (*v*) *Adaptation* of the service configuration while it is active; (*vi*) *Deactivation* of the service in to support its offline modifications; (*vii*) Offline *update* of the service code to be up-to-date, after update the new code optionally gets reconfigured; and, (*viii*) *Decommission* of the offline service when it is not needed on the given host any more. Automation of deployment is automation of all these steps.

In this article, even though there could be other services that are built on top of service deployment, we focus and provide an overview on the various requirements and tasks to be fulfilled by the ASD to support the meta-negotiation and meta-brokering components of the SSV architecture.

Figure 8 shows the SSV components related to ASD and their interconnections. The ASD is built on a repository where all deployable services are stored as virtual appliances (VA). In this context, virtual appliance means everything what is needed in order to deploy a service on a selected site. The virtual appliance could be either defined by an external entity or by the ASD's appliance extraction functionality [18].

To interface with the broker the ASD publishes in the repository the deployable and the already offered services. Thus the repository helps to define a schedule to execute a service request taking into consideration those sites where the service has been deployed and where it could be ex-

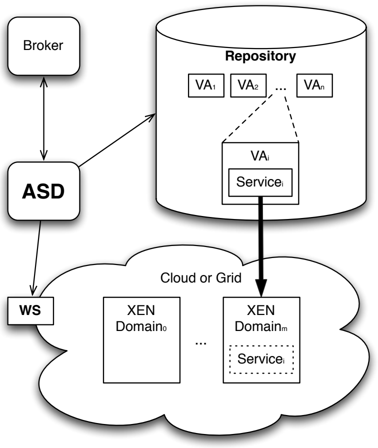

Figure 8: Service deployment in SSV.

ecuted but has not yet been installed. If the requested services are not available, the broker checks whether any of the resources can deliver the service taking into account the possible deployments.

According to the OGSA-Execution Planning Services (EPS) [13] scenarios, a typical service broker has two main connections with the outside world: the *Candidate Set Generators* (CSG), and the Information Services. The task of the CSG is to offer a list of sites, which can perform the requested service according to the SLA and other requirements. Whilst the information services should offer general overview about the state of the SBA, Grid or Cloud. In most cases the candidate set generator is an integral part of the broker. Thus instead of the candidate set adjustments, the *broker* queries the candidate site list as it would do without ASD. Then the broker would evaluate the list and as a possible result to the evaluation it would initiate the deployment of a given service. As a result the service call will be executed as a composed service instead of a regular call. The composition will contain the deployment task as its starting point and the actual service call as its dependent task. Since both the CSG and the brokers heavily rely on the *information system*, the ASD can influence their decision through publishing dynamic data. This data could state service presence on sites where the service is not even deployed.

The selected *placement* of the ASD depends on the site policies on which the brokering takes place. In case the site policy requires a strict scheduling solution then either the CSG or the information system can be our target. If there is no restriction then the current broker can be replaced with an ASD extended one. In case the *candidate set generator* is altered then it should be a distributed, ontology-based adaptive classifier to define a set of resources on which the service call can be executed [36]. The CSG can build its classification rules using the specific attributes of the local information systems. Each CSG could have a feedback about the performance of the schedule made upon its candidates. The ASD extended CSG should have three interfaces to interoperate with other CSGs and the broker. First of all, the CSGs could form a P2P network, which requires two interfaces. The first manages the ontology of the different information systems by sharing the classifier rules and the common ontology patterns distributed as an OWL schema. The second interface supports decision-making among the peers. It enables the forwarding of the candidate request from the broker. The third interface lies between the broker and the CSGs to support passing the feedback for the supervised learning technique applied by the CSGs. This interface makes it possible for the broker to send back a metric packet about the success rate of the candidates.

The *brokers* should be adapted to ASD differently depending on where the ASD extensions are placed. If both the CSG's and the broker's behavior is changed then the broker can make smarter decisions. After receiving the candidate site set, the broker estimates the deployment and usage costs of the given service per candidate. For the estimation it queries the workspace service (WS – provided by the Nimbus project) that offers the virtualization capabilities – virtual machine creation, removal and management – of a given site as a service. This service should accept cost estimate queries with a repository entry (VA) reference as an input. The ASD should support different agents discovering different aspects of the deployment. If only the broker's behavior is changed, and the CSG remains untouched, then the ASD would generate deployment service calls on overloaded situations (e.g. when SLA requirements are endangered). These deployment service calls should use the workspace service with the overloaded service's repository reference.

Finally it is possible to alter the *information system's behavior*. This way the ASD provides information sources of sites, which can accept service calls after deployment. The ASD estimates and publishes the performance related entries (like estimated response time) of the information system. These entries are estimated for each service and site pair, and only those are published which are over a predefined threshold.

Regarding component interactions, the ASD needs to be extended with the following in order to communicate with brokers: Requested service constraints have to be forced independently from what Virtual Machine Monitor (or hypervisor [2]) is used. To help the brokers making their decisions about which site should be used the ASD has to offer deployment cost metrics which can even be incorporated on higher level SLAs. The ASD might initiate service deployment/decommission on its own when it can prevent service usage peaks/lows, to do so it should be aware of the agreements made on higher levels.

In SSV there is a bidirectional connection between the ASD and the service brokers. First the service brokers could instruct ASD to *deploy* a new service (we discussed this in detail in Section 8). However, deployments could also occur independently from the brokers as explained in the following. After these deployments the ASD has to *notify* the corresponding service brokers about the infrastructure changes. This notification is required, because information systems cache the state of the SBA for scalability. Thus even though a service has just been deployed on a new site, the broker will not direct service requests to the new site. This is especially needed when the deployment was initiated to avoid an SLA violation.

5.4. Autonomous case studies in the SSV architecture

This subsection introduces how the self-management and autonomous capabilities of the SSV architecture are used in representative use cases in each SSV layer. Figure 9 summarizes the SSV components and the corresponding self-management examples.

5.4.1. Negotiation Bootstrapping Case Study

A taxonomy of possible faults in this part of the architecture, and the autonomic reactions to these faults are summarized in Table 2.

Before using the service, the service consumer and the service provider have to establish an electronic contract defining the terms of use. Thus, they have to *negotiate* the detailed terms of contract, e.g. the execution time of the service. However, each service provides a unique negotiation protocol, often expressed using different languages representing an obstacle within the SOA architecture and especially in emerging Cloud computing infrastructures [6]. We propose novel concepts for *automatic bootstrapping* between different protocols and contract formats increasing the number of services a consumer may negotiate with. Consequently, the full potential of publicly available services could be exploited.

Figure 9 depicts how the principles of autonomic computing is be applied to negotiation bootstrapping [3]. The management is done through following steps: as a prerequisite of the negotiation bootstrapping users have to specify a *meta negotiation* (MN) document describing the requirements of a negotiation, as for example required negotiation protocols, required security infrastructure, provided document specification languages, etc. (More on meta negotiation can be read in [4].) The autonomic management phases are described in the following:

Monitoring. During the monitoring phase all candidate services are selected, where negotiation is possible or bootstrapping is required.

Analysis. During the analysis phase the existing knowledge base is queried and potential bootstrapping strategies are found (e.g. in order to bootstrap between WSLA and WS-Agreement).

Planning. In case of missing bootstrapping strategies users can define new strategies in a semi-automatic way.

Execution. Finally, during the execution phase the negotiation is started by utilizing appropriate bootstrapping strategies.

5.4.2. Broker Failures Case Study

Autonomic behaviour is needed by brokers basically in two cases: the first one is to survive failures of lower level components, the second is to regain healthy state after local failures. A taxonomy of the sensible failures and the autonomic reactions of the appropriate brokering components are gathered and shown in Table 3. The fourth case, the broker failure, is detailed in the next paragraph. To overcome some of these difficulties brokers in SSV use the help of the ASD component to re-deploy some services.

In the following we present a case study showing how autonomic principles are applied to the Meta-Broker to handle broker failures.

Monitoring. During the monitoring phase all the interconnected brokers are tracked: the IS Agent component of the Meta-Broker gathers state information about the brokers. The Matchmaker component also incorporates a feedback-based solution to keep track of the performances of the brokers.

Analysis. During the analysis phase the Information Collector of the Meta-Broker is queried for broker availability and performance results.

Planning. In case of incoming service request the MatchMaker component determines the ranking of the broker according to their performance data gathered in the previous phases, and the broker with the highest rank is selected. In case of a broker failure the ranks are recalculated and the failed broker is skipped. If the SLA cannot be fulfilled by the new broker, SLA renegotiation is propagated to MN. If no other matching broker is found a new broker deployment request is propagated to ASD.

Execution. Finally, during the execution phase the selected broker is invoked.

5.4.3. Self-initiated Deployment

To cope with the ever varying demand of the services in the service based application, service instances should form autonomous groups based on locality and on the service interfaces they offer (e.g., neighboring instances offering the same interface should belong to the same group). Using peer-to-peer mechanisms, the group can decide to *locally increase the processing power* of a given service. This decision could either involve to *reconfigure an underperforming service instance* in the group to allow heavier loads or it could also introduce the *deployment of a new instance* to handle the increased needs.

The smallest autonomous group our architecture can handle is formed by a single service instance. In case of such small groups, we refer to the previously listed techniques as *self-initiated deployment*. These single service instances are offering self-management interfaces (seen in Figure 4), thus they could identify erroneous situations that could be solved by deploying an identical service instance on another site. A summary of the possible erroneous situations and the autonomic reactions taken when they arise is summarized in Table 4. In the following, we

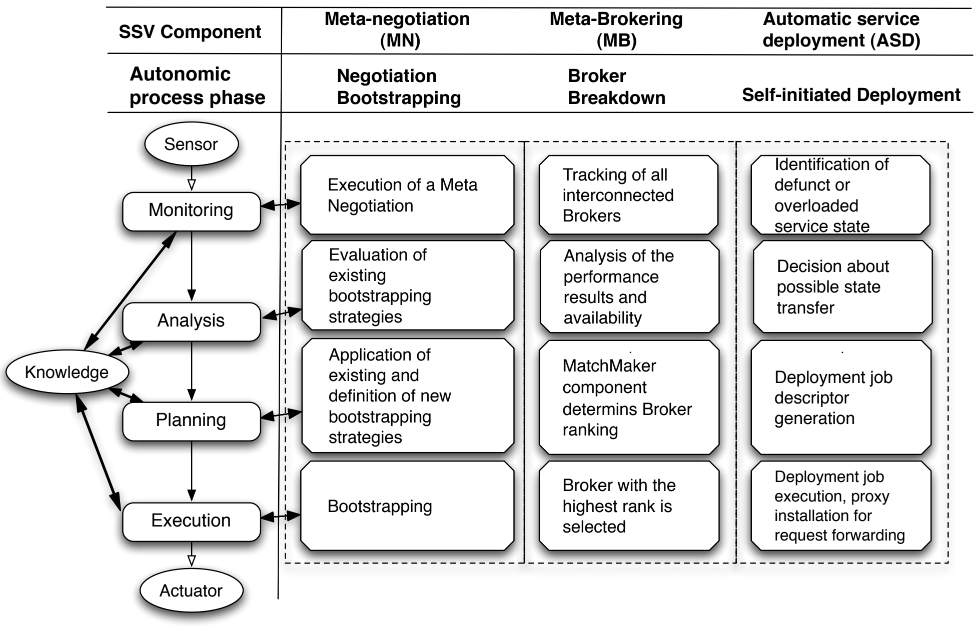
SSV Component
Meta-negotiation (MN)
Meta-Brokering (MB)
Automatic service deployment (ASD)
Autonomic process phase
Negotiation Bootstrapping
Broker Breakdown
Self-initiated Deployment
Sensor
Monitoring
Analysis
Knowledge
Planning
Execution
Actuator
Execution of a Meta Negotiation
Evaluation of existing bootstrapping strategies
Application of existing and definition of new bootstrapping strategies
Bootstrapping
Tracking of all interconnected Brokers
Analysis of the performance results and availability
MatchMaker component determins Broker ranking
Broker with the highest rank is selected
Identification of defunct or overloaded service state
Decision about possible state transfer
Deployment job descriptor generation
Deployment job execution, proxy installation for request forwarding

Figure 9: Use cases for autonomic processes in the SSV architecture

Table 2: Taxonomy of faults and autonomic reactions for Meta Negotiation.

Fault	**Autonomic reaction**	**Propagation**
non matching SLA templates	SLA Mapping as described in [5]	no, handled by the meta-negotiation layer
non matching SLA languages	bootstrapping as described in [3]	no, handled by the meta-negotiation layer

Table 3: Taxonomy of faults and autonomic reactions in Service brokering.

Fault	**Autonomic reaction**	**Propagation**
physical resource failure	new service selection	possible SLA renegotiation or redeployment with ASD
service failure	new service selection	possible SLA renegotiation or redeployment with ASD
wrong service response	new service selection	possible SLA renegotiation
broker failure	new broker selection	possible SLA renegotiation or possible deployment with ASD
no service found by some broker	initiate new service deployment	deployment with ASD
no service found by some broker	new broker selection	possible SLA renegotiation
broker overloading	initiate new broker deployment	deployment with ASD and possible SLA renegotiation
meta-broker overloading	initiate new meta-broker deployment	deployment with ASD

describe the autonomous behavior of a single service instance:

Monitoring. During the monitoring phase the ASD identifies two critical situations: (*i*) the autonomous service instance becomes defunct (it is not possible to modify the instance so it could serve requests again) and (*ii*) the instance turns overloaded on such extent that the underlying virtual machine cannot handle more requests. To identify these situations, service instances should offer interfaces to *share their health status* independently from an information service. This health status information does not need to be externally understood, the only requirement that other service instances, which offer the same interface should understand it.

Analysis. The ASD first decides whether the service initiated deployment is required (because it was overloaded) or the its replication is necessary (because it has became defunct). First in both cases the ASD identifies the service's virtual appliance to be deployed, then in the latter case the ASD also prepares for state transfer of the service before it is actually decommissioned.

Planning. After the ASD identified the service it generates a deployment job and requests its execution from the service broker.

Table 4: Taxonomy of faults and autonomic reactions in Self-Initiated Deployment.

Fault	**Autonomic reaction**	**Propagation**
Degraded service health state	Service reconfiguration	–
Reconfiguration fails	Initiate service cloning with state transfer	If the SLA will not be violated notify service broker about change, otherwise ask for renegotiation
Defunct service	Initiate service cloning	If the SLA will not be violated notify service broker about change, otherwise ask for renegotiation
Service decommissioned	Offer proxy	If the proxy receives request after redirecting the request it also notifies the broker about the new target to avoid later calls
Proxy lifetime expired	Decommission service proxy	–

Execution. If a service is decommissioned on the original site, then a service proxy is placed instead on the site. This proxy forwards the remaining service requests to the newly deployed service using WS-Addressing. The proxy decommissions itself when the frequency of the service requests to the proxy decreases under a predefined value.

6. Evaluation of the SSV architecture in a simulated environment

In order to evaluate our proposed solution, we decided to set up a simulation environment, in which the interoperable infrastructure management of our approach can be examined. For the evaluation of the performance gains of using our proposed SSV solution, we use a typical biochemical application as a case study called TINKER Conformer Generator application [37], gridified and tested on production Grids. The application generates conformers by unconstrained molecular dynamics at high temperature to overcome conformational bias then finishes each conformer by simulated annealing and energy minimization to obtain reliable structures. Its aim is to obtain conformation ensembles to be evaluated by multivariate statistical modeling.

The execution of the application consists of three phases: The first one is performed by a generator service responsible for the generation of input data for parameter studies (PS) in the next phase. The second phase consist of a PS sub-workflow, in which three PS services are defined for executing three different algorithms (dynamics, minimization and simulated annealing – we refer to these services and the generator service as TINKERALG), and an additional PS task that collects the outputs of the three threads and compresses them (COLL). Finally in the third phase, a collector service gathers the output files of the PS sub-workflows and uploads them in a single compressed file to the remote storage (UPLOAD). These phases contain 6 services, out of which four are parameter study tasks that are executed 50 times. Therefore the execution of the whole workflow means 202 service calls. We set up the simulation environment for executing a similar workflow.

For the evaluation, we have created a general simulation environment, in which all stages of service execution in the SSV architecture can be simulated and coordinated. We have created the simulation environment with the help of the CloudSim toolkit [8] (that includes and extends GridSim). It supports modeling and simulation of large scale Cloud computing infrastructure, including data centers, service brokers and provide scheduling and allocations policies. Our general simulation architecture that builds both on GridSim and on CloudSim, can be seen in Figure 10. On the left-bottom part we can see the GridSim components used for the simulated Grid infrastructures, and on the right-bottom part we can find CloudSim components. Grid resources can be defined with different Grid types, they consist of more machines, to which workloads can be set, while Cloud resources are organized into Datacenters, on which Virtual machines can be deployed. Here service requests are called as cloudlets, which can be executed on virtual machines. On top of this simulated Grid and Cloud infrastructures we can set up brokers. Grid brokers can be connected to one or more resources, on which they can execute so-called gridlets (ie. service requests). Different properties can be set to these brokers and various scheduling policies can also be defined. A Cloud broker can be connected to a data center with one or more virtual machines, and it is able to create and destroy virtual machines during simulation, and execute cloudlets on these virtual machines. The Simulator class is a CloudSim entity that can generate a requested number of service requests with different properties, start and run time. It is connected to the created brokers and able to submit these requests to them (so is acts as a user or workflow engine). It is also connected to the realized meta-broker component of SSV, the Grid Meta-Broker Service through its web service interface and able to call its matchmaking service for broker selection.

We submitted the simulated workflow in three phases: in the first round 61 service requests for input generation, then 90 for executing various TINKER algorithms, finally in the third round 51 calls for output preparation. The simulation environment was set up similarly to the real Grid environment we used for testing the TINKER

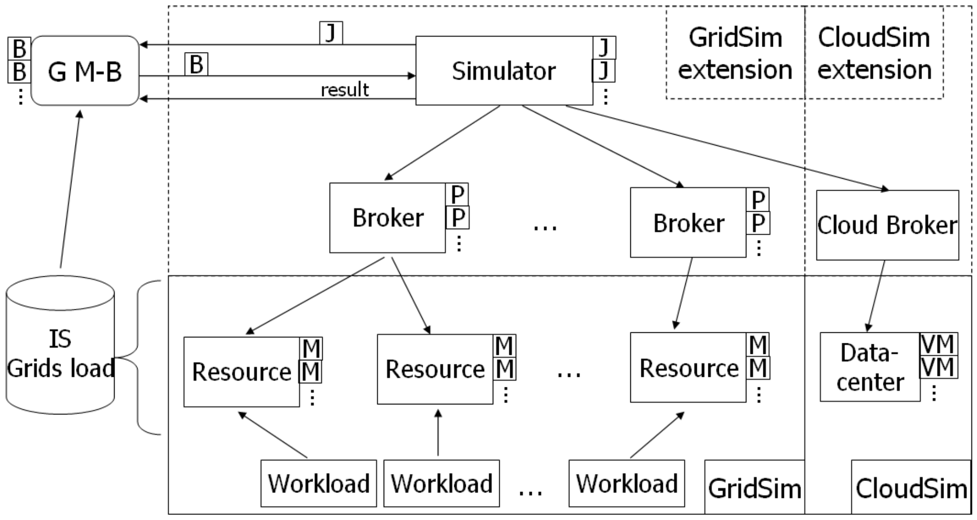

Figure 10: Simulation architecture with CloudSim.

Table 5: Deployment times of the different services in the TINKER application.

Service	Average deployment time	Standard deviation
GEN	8.2 sec	1.34 sec
TINKERALG	8.3 sec	1.48 sec
COLL	6.9 sec	0.84 sec
UPLOAD	6.9 sec	1.21 sec

workflow application. Estimating the real sizes of these distributed environments, we set up four simulated Grids (GILDA, VOCE, SEEGRID and BIOMED [10]) with 2, 4, 6 and 8 resources (each of them had 4 machines). Out of the 202 jobs 151 had special requirements: they use the TINKER library available in the last three Grids, which means these calls need to be submitted to these environments, or to Cloud resources (with pre-deployed TINKER environments). The simulated execution time of the 150 parameter study services were set to 30 minutes, the first generator service to 90 minutes, and the other 51 were set to 10 minutes. All of the four brokers (set to each simulated Grid one-by-one) used random resource selection policy, and all the resources had background workload, for which the traces were taken from the Grid Workloads Archive (GWA) [14] (we used the GWA-T-11 LCG Grid log file). In our simulation architecture we used 20 nodes (called resources in the simulation), therefore we partitioned the logs and created 20 workload files (out of the possible 170 according to the number of nodes in the log). The sorting of the job data to files from the original log file were done continuously, and their arrival times have not been modified, and the run time of the jobs also remained the same. According to these workload files the load of the simulated environments are shown in Figure 11 (which are also similar to the load experienced on the real Grids). One Cloud broker has also been set up. It managed four virtual machines deployed on a data center with four hosts of dual-core CPUs. In each simulation all the jobs were sent to the Meta-Broker to select an available broker for submission. It takes into account the actual background load and the previous performance results of the brokers for selection. If the selected Grid broker had a background load that exceeded a predefined threshold value, it selected the Cloud broker instead.

Out of the 202 workflow services 151 use TINKER binaries (three different algorithms are executed 50 times plus one generator job). These requirements can be formulated in SLA terms, therefore each service of the workflow has an SLA request. If one of these requests are sent to the Cloud broker, it has to check if a virtual machine (VM) has already been created that is able to fulfil this request. If there is no such VM, it deploys one on-the-fly. For the evaluation we used three different Cloud broker configurations: in the first one four pre-deployed VMs are used – one for each TINKER algorithm (TINKERALG) and one for data collecting (capable for both COLL and UPLOAD, used by the last 51 jobs). In the second case we used only one pre-deployed VM, and deployed the rest on-the-fly, when the first call arrived with an SLA. Finally in the third case, when a VM received more then 15 requests the Cloud broker duplicated it (in order to minimize the overall execution time).

Regarding on-demand deployment, we have created 4 virtual appliances encapsulating the four different services our TINKER workflow is based on (namely TINKERALG, COLL and UPLOAD – we defined them in the beginning of this section). Then we have reduced the size of the cre-

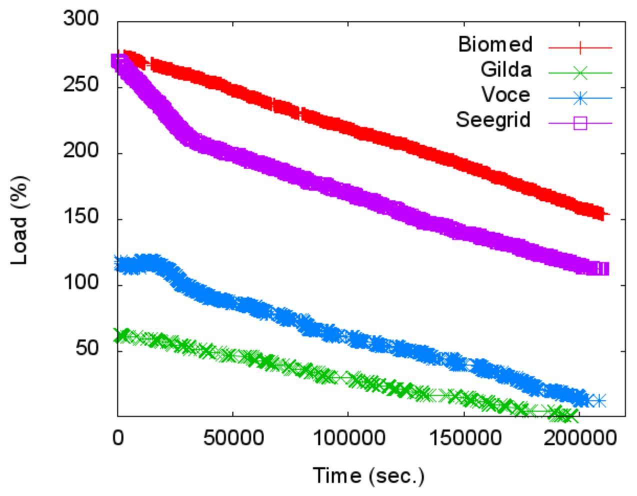

Figure 11: Workload of simulated Grids.

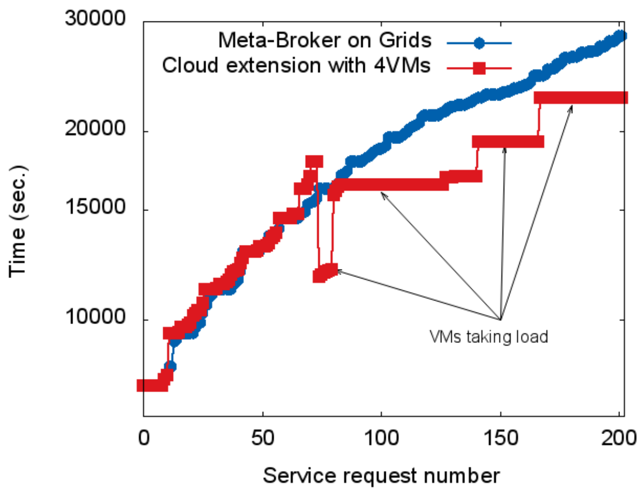

Figure 12: Detailed run times for Cloud bursting with 4 VMs.

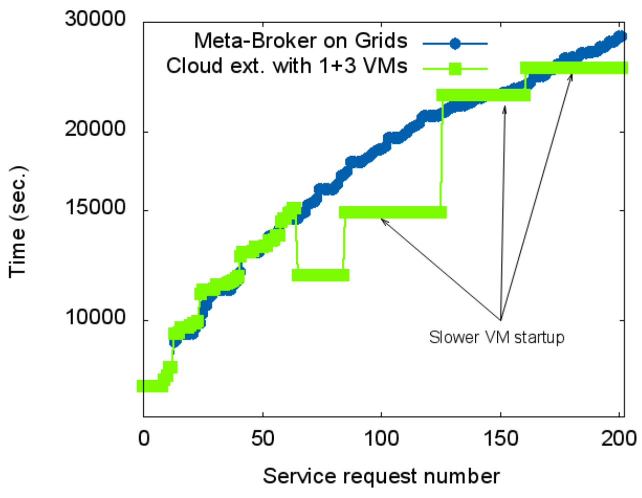

Figure 13: Detailed run times for Cloud bursting with 1+3 VMs.

ated appliances with ASD's virtual appliance optimization facility. Finally we have deployed each service 50 times on an 8 node (32 CPU) Eucalyptus [27] cluster, and measured the interval between the deployment request and the service's first availability. Table 5 shows the measurement results for the TINKERALG, COLL and UPLOAD images. These latencies were also applied in the simulation environment within the Cloud broker.

In order to evaluate the performance of our proposed SSV solution we compare it to a general meta-brokering architecture used in Grid environments. Using this approach we created four simulations: in the first one we use only grid brokers by the Meta-Broker (denoted by MB in the figures) to reach grid resources of the simulated Grids. In the second, third and fourth case we extend the matchmaking of the Meta-Broker (in order to simulate the whole SSV): when the background load of the selected grid broker exceeds 113%, it selects the Cloud broker instead to perform Cloud bursting. In the second case the Cloud broker has four pre-deployed VMs (4VMs), while in the third case only one, and later creates three more as described before (1+3VMs), and in the fourth it has one pre-deployed and creates at most 7 more on demand (1+3+4VMs).

In Figure 12 we can see detailed execution times for the simulations using purely Grid resources and the first strategy using 4 VMs for Cloud bursting. The first phase of the workflow has been executed only on Grid resources in both strategies. Around the 60th job submission the load of Grid resources exceeded the predefined overload threshold, therefore the Cloud Broker has been selected by the GMBS to move further job submissions to Cloud resources. The arrows in the figure denote the places, where Cloud VMs managed to keep constant load by taking over jobs from the Grids. In Figure 13 we denoted the second strategy, where initially only 1 VM was made available for Cloud bursting. The rest 3 VMs have been started on-demand, after the first VM got overloaded. In this case we could save some operational costs for 3 VMs for a short period, but later delayed startup times of the 3 VMs resulted in longer run times for the last 60 jobs of the workflow. Finally, in Figure 14 we can see the third strategy, in which we performed service duplication with 4 additional VMs. In this case we also used on-demand VM startups, but the doubled service instances managed to keep the overall Cloud load down.

In Figure 15 we can see the evaluation results denoting the average aggregated execution times of the service requests. From these results we can clearly see that the simulated SSV architecture overperforms the former meta-brokering solution using only Grid resources. Comparing the different deployment strategies we can see that on demand deployment introduces some overhead (4VMs was faster then 1+3VMs), but service duplication (1+3+4VMs) can enhance the performance and help to avoid SLA violations with additional VM deployment costs.

We have to mention that the SSV architecture puts

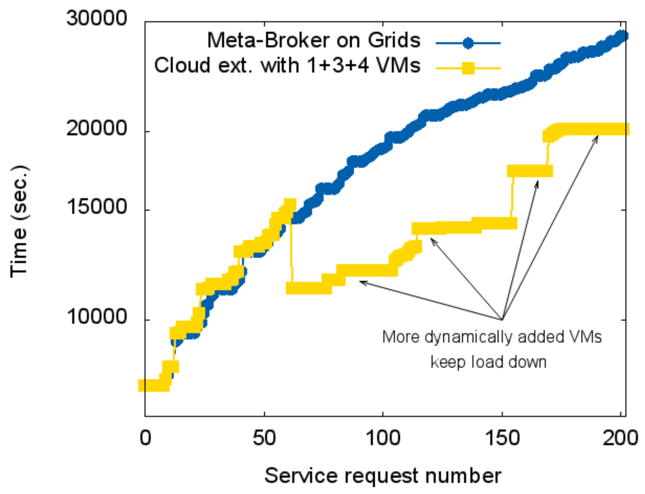

Figure 14: Detailed run times for Cloud bursting with 1+3+4 VMs.

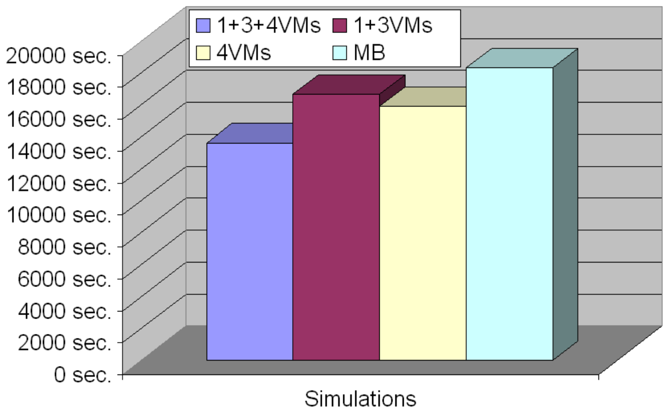

Figure 15: Average request run times.

some overhead on service executions. Regarding the overhead the meta-brokering layer generates, we only need to consider the latency of the web service calls and the matchmaking time of the GMBS. In this evaluation, this latency took up around 200 milliseconds for a call (which is less than 0.1% for all the calls of the application compared to the total execution time we measured in the simulation), and it is also negligible comparing to general brokering response times (which can last up to several minutes in grid environments). At the cloud brokering layer, VM startup times also affect the overall execution times, as we have seen in Figure 13. In our opinion, these latencies are acceptable, hence our automated service execution can manage a federation of heterogeneous infrastructure environments by providing SLA-based service executions under such conditions, where a manual user interaction with a single infrastructure would fail.

7. Conclusions

In a unified system consisting of heterogeneous, distributed service-based environments such as Grids, SBAs and Clouds, there is an emerging need for transparent, business-oriented autonomic service execution. In the future more and more companies will face the problem of unforeseen, occasional demand for a high number of computing resources. In this paper we investigated how such problems could arise and gathered the requirements for a service architecture that is able to cope with these demands. We addressed these requirements to develop a functionally and architecturally well-designed solution, and proposed a novel approach called Service-level agreement-based Service Virtualization (SSV). The presented general architecture is built on three main components: the Meta-Negotiatior responsible for agreement negotiations, the Meta-Broker for selecting the proper execution environment, and the Automatic Service Deployer for service virtualization and on-demand deployment.

We have also discussed how the principles of autonomic computing are incorporated to the SSV architecture to cope with the error-prone virtualization environments, and demonstrated how autonomic actions are triggered responding to various possible failures in the service infrastructure. The autonomic reactions are demonstrated with corresponding case studies. The operation of the proposed architecture is exemplified through highlighting the cases where the autonomous properties of the components of the architecture were activated in order to cope with failures during service executions.

Finally the SSV architecture is validated in a simulation environment based on CloudSim, using a general biochemical application as a case study. The evaluation results clearly fulfil the expected utilization gains compared to a less heterogeneous Grid solution.

8. Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

References

[1] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, and A. Savva. Job submission description language (JSDL) specification, version 1.0. Technical report, 2005. http://www.gridforum.org/documents/GFD.56.pdf.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[3] I. Brandic, D. Music, S. Dustdar. Service Mediation and Negotiation Bootstrapping as First Achievements Towards Self-adaptable Grid and Cloud Services. In *Proceedings of Grids meet Autonomic Computing Workshop*. ACM. June, 2009.

[4] I. Brandic, D. Music, S. Dustdar, S. Venugopal, and R. Buyya. Advanced qos methods for grid workflows based on meta-negotiations and sla-mappings. In *The 3rd Workshop on Workflows in Support of Large-Scale Science*, pages 1–10, November 2008.

[5] I. Brandic, D. Music, P. Leitner, S. Dustdar. VieSLAF Framework: Enabling Adaptive and Versatile SLA-Management. In *the 6th International Workshop on Grid Economics and Business Models 2009 (Gecon09)*, 2009.

[6] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 2009.

[7] R. Buyya, R. Ranjan, and R. N. Calheiros. InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services. Lecture Notes in Computer Science: Algorithms and Architectures for Parallel Processing. Volume 6081, 2010.

[8] R. Buyya, R. Ranjan and R. N. Calheiros. Modeling and Simulation of Scalable Cloud Computing Environments and the CloudSim Toolkit: Challenges and Opportunities. In proc. of the 7th High Performance Computing and Simulation Conference, 2009.

[9] M. Q. Dang and J. Altmann. Resource allocation algorithm for light communication grid-based workflows within an sla context. *Int. J. Parallel Emerg. Distrib. Syst.*, 24(1):31–48, 2009.

[10] Enabling Grids for E-sciencE (EGEE) project website. http://public.eu-egee.org/, 2008.

[11] A.J. Ferrer et. al. OPTIMIS: a Holistic Approach to Cloud Service Provisioning. Future Generation Computer Systems, vol. 28, pp. 66–77, 2012.

[12] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 83–92, New York, NY, USA, 1998. ACM.

[13] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Reich. The open grid services architecture, version 1.5. Technical report, 2006. http://www.ogf.org/documents/GFD.80.pdf.

[14] The Grid Workloads Archive website. http://gwa.ewi.tudelft.nl, 2010.

[15] R. Howard and L. Kerschberg. A knowledge-based framework for dynamic semantic web services brokering and management. In *DEXA '04: Proceedings of the Database and Expert Systems Applications, 15th International Workshop*, pages 174–178, Washington, DC, USA, 2004. IEEE Computer Society.

[16] A. Iosup, T. Tannenbaum, M. Farrellee, D. Epema, and M. Livny. Inter-operating grids through delegated matchmaking. *Sci. Program.*, 16(2-3):233–253, 2008.

[17] K. Keahey, I. Foster, T. Freeman, and X. Zhang. Virtual workspaces: Achieving quality of service and quality of life in the grid. *Sci. Program.*, 13(4):265–275, 2005.

[18] G. Kecskemeti, G. Terstyanszky, P. Kacsuk, and Zs. Nemeth. An Approach for Virtual Appliance Distribution for Service Deployment. Future Generation Computer Systems, vol. 27, issue 3, pp 280–289, 2011.

[19] J.O. Kephart, D.M. Chess. The vision of autonomic computing. *Computer* . 36:(1) pp. 41-50, Jan 2003.

[20] A. Kertesz and P. Kacsuk. GMBS: A New Middleware Service for Making Grids Interoperable. In *Future Generation Computer Systems*, vol. 26, no. 4, pp. 542-553, 2010.

[21] C. Kesselman and I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[22] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience*, 32(2):135–164, 2002.

[23] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2004. IEEE Computer Society.

[24] K. Leal, E. Huedo, I. M. Llorente. A decentralized model for scheduling independent tasks in Federated Grids. Future Generation Computer Systems, Volume 25, Issue 8, pp. 840–852, 2009.

[25] K. Lee, N.W. Paton, R. Sakellariou, E. Deelman, A.A.A. Fernandes, G. Mehta. Adaptive Workflow Processing and Execution in Pegasus. In *Proceedings of 3rd International Workshop on Workflow Management and Applications in Grid Environments*, pages 99–106, 2008.

[26] Y. Liu, A. H. Ngu, and L. Z. Zeng. Qos computation and policing in dynamic web service selection. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 66–73, New York, NY, USA, 2004.

[27] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *CCGRID*, pages 124–131. IEEE Computer Society, 2009.

[28] The Open Grid Forum website. http://www.ogf.org, 2010.

[29] D. Ouelhadj, J. Garibaldi, J. MacLaren, R. Sakellariou, and K. Krishnakumar. A multi-agent infrastructure and a service level agreement negotiation protocol for robust scheduling in grid computing. In *Proceedings of the 2005 European Grid Computing Conference (EGC 2005)*, February 2005.

[30] M. Parkin, D. Kuo, J. Brooke, and A. MacCulloch. Challenges in eu grid contracts. In *Proceedings of the 4th eChallenges Conference*, pages 67–75, 2006.

[31] D. M. Quan and J. Altmann. Mapping a group of jobs in the error recovery of the grid-based workflow within sla context. *Advanced Information Networking and Applications, International Conference on*, 0:986–993, 2007.

[32] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: Accountable execution of untrusted programs. In *In Workshop on Hot Topics in Operating Systems*, pages 136–141, 1999.

[33] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin. I. Lloriente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda W. Emmerich, F. Galan. The RESERVOIR Model and Architecture for Open Federated Cloud Computing. IBM Journal of Research and Development, 53(4), 2009.

[34] I. Rodero, F. Guim, J. Corbalan, L. Fong, Y. Liu, and S. Sadjadi. Looking for an evolution of grid scheduling: Meta-brokering. In *Grid Middleware and Services Challenges and Solutions*, pages 105–119. Springer US, 2008.

[35] M. Surridge, S. Taylor, D. De Roure, and E. Zaluska. Experiences with gria – industrial applications on a web services grid. In *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing*, pages 98–105, Washington, DC, USA, 2005. IEEE Computer Society.

[36] M. Taylor, C. Matuszek, B. Klimt, and M. Witbrock. Autonomous classification of knowledge into an ontology. In *The 20th International FLAIRS Conference (FLAIRS)*, 2007.

[37] TINKER Conformer Generator workflow. http://www.lpds.sztaki.hu/gasuc/index.php?m=6&r=12, 2011.

[38] H. N. Van, F. D. Tran, and J. Menaud. Autonomic virtual resource management for service hosting platforms. In *Proceedings of the ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 1–8, 2009.

[39] C. Vazquez, E. Huedo, R. S. Montero et I. M. Llorente. Federation of TeraGrid, EGEE and OSG Infrastructures through a Metascheduler. Future Generation Computer Systems 26, pp. 979–985, 2010.

[40] S. Venugopal, R. Buyya, and L. Winton. A grid service broker for scheduling e-science applications on global data grids. *Concurrency and Computation: Practice and Experience*, 18(6):685–699, 2006.

[41] C. A. Yfoulis, and A. Gounaris. Honoring SLAs on cloud computing services: a control perspective. In *Proceedings of the European Control Conference*, 2009.