FRAUNHOFER INSTITUTE FOR SCIENTIFIC COMPUTING AND ALGORITHMS

Orchestration of Resources in Distributed, Heterogeneous Grid Environments Using Dynamic Service Level Agreements

Oliver Wäldrich

Genehmigte Dissertation

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Fakultät für Elektrotechnik und Informationstechnik an der
Technischen Universität Dortmund

Sankt Augustin
07.12.2011

To Bernd and Gisela.

ACKNOWLEDGEMENTS

Writing this thesis was not always an easy process. As always in life there were good times and there were bad times. No matter what, some people were always on my side. Many thanks go therefore to my family. They never lost the faith in me and gave me the strength to accomplish this work.

I want to especially thank Wolfgang Ziegler. He provided me with a productive working environment and interesting projects; he gave me new insides to science and always found the time to listen to me. Thank you, Wolfgang.

Many thanks go to my supervisor, Prof. Uwe Schwiegelshohn, who guided me through the process of writing this dissertation. His criticism and comments were a big help in order to substantially improve this thesis.

I also want to thank all the (former and actual) members of the OGF GRAAP working group. Special thanks go to Dominic Battré for all the fruitful discussions on WS-Agreement that helped me to gain that deep understanding on that topic I have now. I want to thank Prof. Frances Brazier, Kassidy Clark, Michel Oey, Philipp Wieder, Alexander Papaspyrou, Wolfgang and Dominic for the productive work and discussions on WS-Agreement Negotiation. This work would not have been possible without you.

Finally I want to thank the authors of the WS-Agreement specification who set the corner stone for this thesis with their work.

1. Gutachter: Professor Dr.-Ing. Uwe Schwiegelshohn
2. Gutachter: Professor Dr. Frances Brazier

Tag der Einreichung: 09.06.2011

Tag der Prüfung: 06.12.2011

ABSTRACT

In recent decades the acceptance of the internet and the increase of network capacity have resulted in a situation in which it is now possible to transfer huge amounts of data efficiently and reliably between different computing systems worldwide. This enables new paradigms in provision and use of distributed IT resources. A well-known paradigm is grid computing where computing resources owned by various institutions and organizations are used in a coordinated way in order to solve scientific and economic problems. Access to these resources is realized through public or private networks. Besides computing resources also data, data storage or software resources are provided. In this context it becomes more and more important with which quality these resources are provided. This may be, for example, the minimal availability of computing resources, the maximum access time of a data storage or the maximum response time of a web-based application. Offering resources with a defined quality means for resource providers that they need to implement specific processes to assert the quality of the provisioning process. On the other hand, they can also provide their services at different quality levels. Services with lower quality levels can be offered cheaper than those with high quality levels. Therefore, service consumers can now select the required service with the appropriate service level in terms of their requirements and budget. This provides both parties, service provider and consumer, with more flexibility in the service provisioning process.

Service level agreements (SLAs) are an accepted approach to realize contracts for IT services and service qualities. They describe the functional and the non-functional requirements of IT services. Additionally, they define compensation and penalties for delivering services with the defined requirements respectively for failing to meet these quality criteria. This thesis examines methods for negotiation and management of SLAs in distributed systems based on the WS-Agreement standard. The focus is on methods for SLA declaration, automated SLA negotiation and creation processes, monitoring of SLA guarantees, and the application of SLAs for coordinated IT resource provisioning. Therefore, a protocol for dynamic negotiation or renegotiation of SLAs is developed as an extension to the WS-Agreement specification. This includes the definition of a negotiation model for the exchange of offers between the negotiating partners. The subsequent SLA creation process is an automated process in distributed systems. Since SLAs are a kind of electronic contracts a mechanism for validating the integrity of SLA offers was developed and is presented in detail. In addition, automatic methods for SLA guarantee evaluation are described. Finally, an orchestration service for co-allocating arbitrary resources such as computing and network resources is presented. The resource orchestration process has been realized using SLAs. The architecture of this service is evaluated and based on the evaluation result an advanced orchestration service architecture is conceived.

ZUSAMMENFASSUNG

Die Akzeptanz des Internets und der zunehmende Ausbau von Netzwerkkapazitäten führte in den vergangenen Jahrzehnten zu einer Situation in der es nun möglich ist riesige Datenmengen effizient und zuverlässig zwischen verschiedenen Rechnersystemen weltweit zu transferieren. Dies ermöglicht neue Paradigmen bei der Bereitstellung und Nutzung verteilter IT-Ressourcen. Ein wohlbekanntes Paradigma ist Grid-Computing bei dem Rechenressourcen verschiedener Institutionen bzw. Organisationen koordiniert zur Lösung wissenschaftlicher und wirtschaftlicher Problemstellungen genutzt werden. Der Zugang zu den benötigten Rechenressourcen wird dabei über öffentliche oder private Netze realisiert. Neben Rechenressourcen werden dabei auch Daten, Datenspeicher oder Software bereitgestellt. Dabei gewinnt auch die Qualität mit der diese Ressourcen bereitgestellt werden immer mehr an Bedeutung. Darunter versteht man zum Beispiel die minimale Verfügbarkeit von Rechenressourcen, die maximale Zugriffszeit eines Datenspeichers oder die maximale Antwortzeit einer web-basierten Anwendung. Für die Ressourcenanbieter bedeutet dies dass spezifische Prozesse implementiert werden müssen um IT-Dienste mit einer definierten Qualität bereitzustellen. Zudem können sie ihre Dienste in unterschiedlicher Dienstqualität bereitstellen. Dienste mit geringerer Qualität können so preiswerter angeboten werden als solche mit hoher Qualität. Anwender hingegen können den passenden Dienst hinsichtlich ihrer Anforderungen und ihres Budgets auswählen.

Service Level Agreements (SLAs) sind ein akzeptierter Ansatz um Verträge über IT-Dienste und Dienstqualitäten zu realisieren. SLAs beschreiben sowohl die funktionalen als auch die nicht-funktionalen Anforderungen von IT-Diensten. Darüber hinaus definieren sie Vergütung und Strafen bei Erfüllung bzw. Nichterfüllung der definierten Anforderungen. Diese Arbeit behandelt Methoden zur Verhandlung und Verwaltung von dynamischen SLAs in verteilten Systemen auf Basis des WS-Agreement Standards. Im Fokus steht hierbei die Deklaration von SLAs, deren automatisierte Verhandlung und Erstellung, das Monitoring von SLA Garantien, sowie die Verwendung von SLAs zur koordinierten Nutzung von IT-Ressourcen. Zu diesem Zweck wurde aufbauend auf die WS-Agreement Spezifikation ein Protokoll zur dynamischen Verhandlung bzw. Neuverhandlung von SLAs entwickelt. Dies beinhaltet die Definition eines Verhandlungsmodells zum Austausch von Angeboten zwischen den Verhandlungspartnern. Die anschließende Erstellung der SLAs basiert auf dem WS-Agreement Standard stellt einen automatisierter Prozess dar. Da es sich bei SLAs um elektronische Verträge handelt wurden Mechanismen zur Validierung von SLA Angeboten entwickelt und im Detail vorgestellt. Darüber hinaus werden Methoden zur automatisierten Evaluation von SLA Garantien beschrieben. Abschließend wird ein Orchestrierungsdienst zur Co-Allokation beliebiger Ressource wie z.B. Rechen- und Netzwerkressourcen vorgestellt. Die Orchestrierung der verschiedenen Ressourcen wurde mittels SLAs realisiert. Die Architektur des Orchestrierungsdienstes wird evaluiert und basierend auf dem Resultat der Evaluierung wird eine erweiterte Dienstarchitektur konzeptioniert.

Contents

INTRODUCTION 1

CHAPTER 1
SERVICE LEVEL AGREEMENTS IN GRIDS 5
1.1 Service Level Agreements in Service-Oriented Architectures 5
1.1.1 Approaches to Service Level Agreements 7
1.1.2 Requirements of a Generic SLA Management Layer 9
1.1.3 Coverage of WS-Agreement 10
1.2 WS-Agreement Overview 11
1.2.1 Related Standards 11
1.2.2 The WS-Agreement Model 12
1.2.3 The WS-Agreement Protocol 13
1.2.4 The WS-Agreement Language 17
1.3 Summary 21

CHAPTER 2
NEGOTIATION OF SERVICE LEVEL AGREEMENTS 23
2.1 Goals and Requirements 25
2.2 Notational Conventions and Terminology 26
2.3 Use Cases 29
2.4 WS-Agreement Negotiation Model 31
2.4.1 Negotiation Offer/Counter Offer Model 31
2.4.2 Layered Architectural Model 33
2.5 Negotiation 35
2.5.1 Negotiation Context 35
2.5.2 Negotiation Offer 39
2.6 Creation of Negotiated and Renegotiated Agreements 47
2.6.1 Negotiation Extension Document 48
2.6.2 Renegotiation Extension Document 49
2.7 Deployments of the Negotiation Port Types 50
2.7.1 Simple Client-Server Negotiation 51
2.7.2 Bilateral Negotiation with Asymmetric Agreement Layer 51
2.7.3 Re-Negotiation of Existing Agreements 53
2.8 Summary 54

CHAPTER 3
ARCHITECTURE OF THE SLA FRAMEWORK 55
3.1 Related Work 56
3.2 Overview of the WSAG4J Framework Architecture 58

3.3 Integration of the WSAG4J Engine into the WSRF Layer **59**
3.4 Architecture of the Agreement Factory **63**
3.4.1 WSAG4J Agreement Factory Components 63
3.4.2 Agreement Offer Validation 66
3.4.3 Constraint Validation in the WSAG4J Engine 73
3.5 Agreement Monitoring System Architecture **78**
3.5.1 Processing Agreement Resource Properties 79
3.5.2 Monitoring of the Agreement States 81
3.5.3 Guarantee Evaluation Example 85
3.6 Summary **90**

CHAPTER 4
ORCHESTRATION OF SERVICE LEVEL AGREEMENTS **91**
4.1 Related Work **91**
4.2 Orchestration Scenarios **92**
4.2.1 Co-allocation of Different Resource Types 93
4.2.2 Load Balancing of Web-Applications 93
4.3 A Generic Orchestration Service Architecture **94**
4.4 Evolution of the Orchestration Service **96**
4.5 The Phosphorus Orchestration Service **99**
4.5.1 Resource Layer Architecture 100
4.5.2 Implementation of the Orchestration Service 107
4.5.3 Co-Allocation of Computational and Network Resources 109
4.6 Evaluation of the Orchestration Service Architecture **111**
4.6.1 Dynamic File Transfer Scenario 112
4.6.2 Dynamic Load Balancing of Web-Applications 112
4.6.3 Evaluation Result 114
4.7 Enhanced Orchestration Service Architecture **116**
4.8 Summary **121**

CONCLUSION **123**

BIBLIOGRAPHY **127**

APPENDIX **133**
WS-Agreement Negotiation XML Schema 133
WS-Agreement Negotiation Factory WSDL 139
WS-Agreement Negotiation WSDL 142
WS-Agreement Negotiation Advertisement WSDL 147
Agreement Template Example 149
Agreement Properties Document Example 159

INTRODUCTION

Information and communication technology became an integral part of our daily life and our society during the last 20 years. Pervasive access to communication and information services has dramatically changed our way of accessing and interacting with data, information, and services. While modern telecommunication services provide us with the ability to communicate with everybody at any time, the Internet provides us with the capability to access information everywhere at any time. Computational Grids just provide another type of service, the transparent access to computational resources. Similar as communication and information services, ubiquitous access to computational services will also have a major impact on our daily life in future.

The term Grid was initially adopted as an analogy to the electric power grid, which provides us with dependable, pervasive, and inexpensive access to electricity. Accordingly, the following early definition of a Grid was given by Ian Foster and Carl Kesselman in [1]:

"A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities."

The envisioned Grid infrastructure consists of a set of heterogeneous, geographically distributed, computational resources from multiple administrative domains that are shared in order to achieve a common goal. Foster and Kesselman therefore extended their definition later on in the book "The Grid 2: Blueprint of a New Computing Infrastructure" [2] as follows:

"The sharing that we are concerned with is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem-solving and resource-brokering strategies emerging in industry, science, and engineering. This sharing is, necessarily, highly controlled, with resource providers and consumers defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which sharing occurs."

In order to address major challenges of Computational Grids, such as the heterogeneity of resources and resources management systems, different domains of ownership, or data management, a set of Grid middleware systems were developed. Globus [3] and Unicore [4] are two prominent representatives of such Grid middleware systems. They provide basic resource and data management capabilities in conjunction with an integrated security concept, which allows implementing sophisticated access policies. Therefore, these systems are a fundamental pillar for production Grids today.

In the last years, the paradigm of a Service-Oriented Architecture (SOA) became more and more important. Grid middleware systems adopted SOA as a basic architectural concept and offer their functionality in form of services. Moreover, SOA has an important impact on the Grid service provisioning model. Grid resources are often offered through Grid scheduler or

resource broker as abstract services, for example as infrastructure services or software services. As a result a set of new service provisioning models appeared, for example Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). These service provisioning models enable service providers to offer their capabilities in a flexible, transparent way. Service consumers can acquire computational services without dedicated knowledge of how exactly the service is provided. However, this requires that service consumer and provider must define in detail the service that is provided as well as potential quality aspects. Service Level Agreements (SLAs) are a suitable instrument for this purpose. The explicit definition of service and quality aspects makes it possible to compare services from different providers and to monitor the fulfillment of the service delivery.

Service level agreements are used in the IT service provisioning for quite some time. Providers of telecommunication services use SLAs since the late 1980s as part of the contracts with their customers. A telecommunication provider that offers internet access to its customers usually defines a SLA as part of its customer contracts. This SLA specifies for example the bandwidth and availability of the internet access. Since its appearance in the telecommunications area, the usage of SLAs has spread to a wide number of other application areas. Call centers define SLAs with their customers in order to specify for example the maximum rate of calls that are abandoned while waiting to be answered, the average time to answer, or the rate of calls that are answered in a defined timeframe. IT service providers define similar SLAs as part of their service contracts. These SLAs may specify the availability of a service during the core working hours (e.g. from 8am-6pm) or a maximum service recovery time in case of failure. Especially in outsourcing scenarios where customers move parts of their IT infrastructure to an external service provider the definition of SLAs is of essential importance.

Since service level agreements are service contracts between a service provider and a service consumer, they are an important step into the direction of transforming a scientific-oriented infrastructure such as the Grid into a commercial oriented-infrastructure as the Cloud. Grid and Cloud systems therefore currently adapt the SLA paradigm as part of their service provisioning strategies. In the past these systems focused on providing appropriate resource management capabilities to allocate compute resources and infrastructure services on demand. These capabilities are exposed as services that can be accessed over the network and are implemented as part of the corresponding middleware solutions. Based on these capabilities a wide set of distributed resource management systems were developed that implement strategies to optimize the usage of available resources, to detect failures and to recover from them in case they occur. Future systems should also acknowledge service quality as a fundamental service provisioning parameter. Service availability, failure recovery times, or minimum and average service response times could for example be used by SLA aware systems to dynamically select services. On the other hand, service providers could implement self-optimization strategies in order to use their resources even more efficiently and to minimize over-provisioning without violating an agreed service level. Since today's systems lack appropriate SLA management capabilities, this thesis investigates methods to dynamically provide and manage SLAs in distributed systems. Based on these SLA

management capabilities, an approach to dynamic resource orchestration using SLAs is presented.

The remainder of this thesis is structured as follows. Chapter 1 gives a general introduction to service level agreements and service-oriented architectures, and presents different approaches to SLAs in distributed systems. These approaches are then compared and WS-Agreement is motivated as the SLA management approach of choice in this thesis. Following, an overview of WS-Agreement is given and the basic protocol and language concepts are introduced. Then open issues of WS-Agreement are identified and an approach to solve these shortcomings is presented in Chapter 2. This chapter describes how to extend WS-Agreement with negotiation capabilities, without interfering with the existing SLA management capabilities. Therefore, a bilateral negotiation model based on the exchange of offers and counter offers is presented and a simple state model is defined that describes possible state transitions of offers and counter offers. Finally, we define a negotiation protocol and describe how negotiation can be used in different deployment scenarios. In Chapter 3, WSAG4J [5] is introduced as a generic SLA management framework based on WS-Agreement. The framework architecture and unique features such as dynamic compliance validation of agreement offers and agreement monitoring and evaluation are described in detail. The capability to dynamically validate agreement offers based on arbitrary constraints defined in an agreement template was implemented for the first time in the WSAG4J framework. The same applies to the dynamic guarantee evaluation process. Chapter 4 finally discusses how SLAs can be used for resource orchestration. We present the Phosphorus [6] orchestration service as an example that is capable to co-allocate different types of resources using SLAs. The architecture of this service is then described and evaluated in order to assess its applicability for more complex orchestration scenarios. Based on the evaluation results, an extended architecture is proposed that overcomes identified limitations.

Chapter 1
SERVICE LEVEL AGREEMENTS IN GRIDS

The concept of a service-oriented architecture (SOA) is a paradigm in distributed system design. Modern Grid systems are often implemented following the SOA paradigm. In section 1.1, we introduce the concept of a service-oriented architecture and describe the relevance of service level agreements (SLA) for SOAs. In section 1.1.1, we give an overview of existing approaches for defining SLAs in service-oriented architectures and we motivate the decision of using WS-Agreement as a standard for SLA management in this thesis. In section 1.1.2, we describe general requirements of a generic SLA layer. The coverage of these requirements by the WS-Agreement is then discussed in section 1.1.3. In the second part of this chapter we provide an introduction to WS-Agreement, its architectural model, the relevant port types and language concepts.

1.1 Service Level Agreements in Service-Oriented Architectures

In order to understand the relevance of service level agreements in service-oriented architectures, we first need to understand what exactly a SOA is. The Reference Model for Service-Oriented Architecture [7] defines a SOA as follows:

> "Service-Oriented Architecture is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations."

The reference model defines a service-oriented architecture on a very abstract level. In general, the reference model assumes that in distributed systems there are entities with needs and entities with capabilities. Services are the concept to deliver the capabilities to those entities with needs. The reference model therefore describes abstractly the relevant mechanisms to offer services, to discover and interact with them.

A more concrete definition of a service-oriented architecture is provided by Krafzig et al. [8]. They define a SOA as follows:

> "A Service-Oriented Architecture (SOA) is a software architecture that is based on the key concepts of an application frontend, service, service repository, and service bus. A service consists of a contract, one or more interfaces, and an implementation."

With this definition the authors introduce the concepts of *application frontend*, *service*, *service repository*, and *service bus*.

Application frontends are essentially the consumers of services in a SOA. They are active components in a SOA, they are not services by themselves, but they invoke and control the capabilities provided by SOA services. Application frontends are for example web-

applications that directly interact with the end-user, or batch processes that run on behalf of the end-user.

A *service* is a software component that provides capabilities to application frontends or other services. A service consists of a *contract*, an *interface*, and a *service implementation*. The *contract* constitutes the formal description of the purpose of a service, its functionality, and how the service is used. It optionally comprises a formal description of the service interface, for example in the form of WSDL [9]. The functionality of the service is accessed over the network via the *service interface*. This is the physical implementation of the contract, consisting of client stub and server skeleton. The *service implementation* is the realization of the service contract. It usually comprises the business logic and data required to deliver the capability defined in the contract.

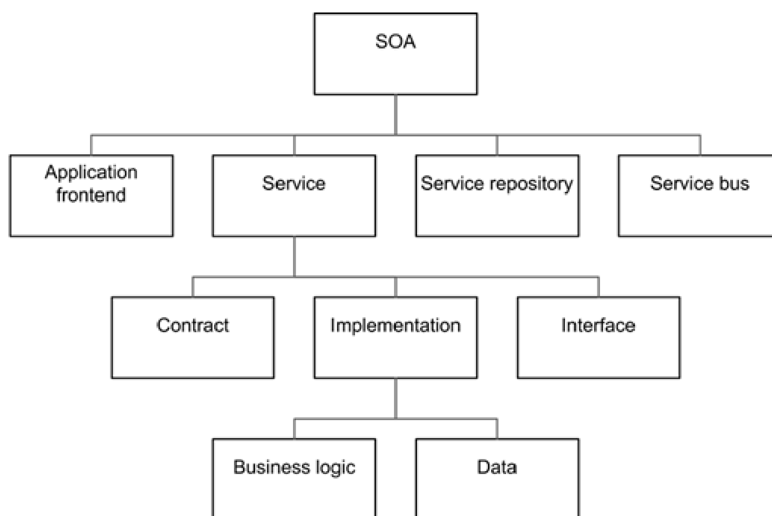


Figure 1: components of a service-oriented architecture [8]

The *service repository* provides the required functionality to discover services in a SOA. Besides the information contained in the service contract, the repository can contain additional information such as the location of a service, contact and support information, or license fees.

Finally, the *service bus* is the component that interconnects the entities of the SOA. The components of a service-oriented architecture are depicted in

.

Even though the SOA paradigm does not prescribe a particular technology to implement SOA services, web-services are often used for this purpose. The web-service architecture describes a simple workflow to publish, discover and use existing web-services. Service providers register their web-services in a public accessible service registry, also called service repository. This registry enables potential consumers to find available services. In the context of a SOA public accessible means that actors of the SOA have access to the registry. The service registry stores information such as the service description, the service interface definition (e.g. using the Web Service Description Language), and a description how to use the service. It can support different access mechanisms, for example a web-application for manually browsing the registry and a web-service-based interface in order to access it programmatically. The Universal Description, Discovery and Integration (UDDI) [10] standard is a well-known representative that defines an interface for accessing service registries programmatically. When a service consumer found an adequate web-service that fulfills its requirements, it can access it via its described interface. The description of the service interface definition is usually retrieved from the service registry during the service

discovery process. Web-services use the Web Service Description Language (WSDL) [9] to describe their interfaces. Service consumers use the WSDL of a web service to bind to that service. Service binding usually comprises the generation of the client stubs required for invoking the service. Client stubs are the components of a computer program that allow to access a service over the network. A web-service client communicates with a service using the SOAP protocol [11]. Client stubs are responsible for serializing the input parameters of a service invocation, to map a service call to the respective SOAP messages, and to de-serialize the service response. Figure 2 illustrates the process of service discovery and invocation.

The service discovery and invocation process described before only considered the functional properties of a service. Nonfunctional properties such as the availability of a service, the average response time, or warranties and remedies are not covered in this approach. This problem is of special importance for service-oriented architectures. In SOAs service consumers are often services that automatically discover and invoke other services.

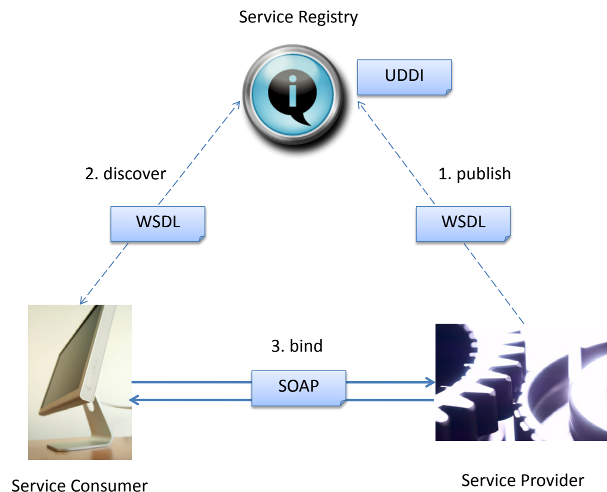


Figure 2: service discovery and invocation in SOAs

In general it is possible to define the service quality and the according service level agreements for SOA services "out of band", for example in form of paper contracts. However, this makes the overall system rather static, inflexible and inert. In highly dynamic systems such as the Grid, where services are requested and provided in an automatic way, this is not an appropriate way to address the system requirements. It is obvious that a dynamic way to negotiate, monitor and enforce the quality for a service is needed. Service level agreements (SLAs) are one way of solving this problem.

In general service level agreements are contractual relationships between service providers and service consumers that are related to a service delivery process. The IT Infrastructure Library (ITIL) [12] defines a SLA in its glossary as an "... Agreement between an IT Service Provider and a Customer. The SLA describes the IT Service, documents Service Level Targets, and specifies the responsibilities of the IT Service Provider and the Customer. A single SLA may cover multiple IT Services or multiple Customers." Service Level Agreements are currently adopted for service-oriented architectures as part of dynamic service delivery processes. The following section presents different approaches to service level agreements in SOAs.

1.1.1 Approaches to Service Level Agreements

The need to specify the quality of a service in service-oriented architectures is not a new one. Different approaches for defining and managing SLAs exist. These approaches comprise for

example languages to express SLAs, solutions to offer and discover services with dedicated service levels, and frameworks for dynamic SLA negotiation, creation and monitoring. Prominent examples are Web Service Level Agreement (WSLA), the Web Service Offerings Language (WSLO), SLAng [13], WS-QoS, the Web Service Management Network (WSMN) or WS-Agreement. In the following a brief overview of four SLA approaches is given, namely SLAng, WSLA, WSOL and WS-Agreement. A more comprehensive comparison of SLA approaches can be found in [14] and [15]. Pontz et. al compare in [14] WSLA, WS-Agreement, WSLO, WS-QoS and WSMN. They investigate their applicability for portfolio management in the financial industry. Tang et. al give in [15] a comparison of WS-Agreement, WSLA, WSOL, WSMN and WS-QoS with respect to their support of the C-MAPE model.

SLAng [13] is an XML-based language to define service level agreements for application service provisioning, in particular for the provisioning of distributed applications. The authors assume that such applications are assembled from components, using a component oriented middleware. In order to guarantee end-to-end QoS for such distributed applications, each application component must satisfy certain QoS constraints. SLAng therefore describes a service provisioning reference model, consisting of an application tier, a middle tier and a resource tier. The application tier comprises the applications to provide. These applications are assembled by the components and services offered at the middle tier. The middle tier consists of middleware components such as web servers, application components and component containers. The resource tier comprises the underlying resources, for example storage and network resources. SLAng anticipates the use of vertical and horizontal SLAs in order to provide end-to-end QoS for an application. Vertical SLAs describe contracts between components of different layers, while horizontal SLAs do the same for components of one layer.

Web Service Level Agreement (WSLA) [16] is a SLA management framework specification created by IBM [17]. It provides the required capabilities for creating and monitoring SLAs for web services, but it is also applicable for any inter-domain management scenario as well, for example the management of networks, systems or applications in general. WSLA provides a flexible and extensible language. It enables service customers and providers to unambiguously define SLAs and to specify SLA parameters and how they are measured [18]. The WSLA framework subsequently influenced the work on the WS-Agreement specification and basic concepts of WSLA were incorporated into the WS-Agreement later on.

The Web Service Offering Language (WSOL) [19] is a XML-based language definition that can be used to offer web-services at different service levels. WSOL therefore defines so called service offerings. A service offering represents one service class with a defined quality of service. Additionally, a service offering can comprise functional and nonfunctional constraints, simple access rights, and pricing data. The basic idea is that similar web-services can in general be offered at different service levels, for example with different response times. Applications that search a service registry for particular service will therefore find multiple service instances, where each service instance is offered with a different service quality. Each service instance is therefore associated with a service offering. Depending on the application requirements and the service offerings, the application can now choose the service instance

that meets its requirements best, for example the cheapest service instance or the instance with the fastest response time. WSOL therefore supports dynamic adaptation services with respect to the service offerings.

The WS-Agreement [20] specification is a standardization approach of the Grid Resource Allocation and Agreement Protocol Working Group (GRAAP-WG) of the Open Grid Forum (OGF) [21]. It defines a protocol and a language to dynamically create and monitor bi-lateral service level agreements in distributed systems. It is designed to be domain independent; it can therefore be used to create and manage SLAs for arbitrary types of services. Domain specific service description languages can be plugged into WS-Agreement as needed. The specification defines mechanisms for service monitoring, SLA monitoring, and SLA accounting.

In the remainder of this thesis we rely on WS-Agreement as SLA management framework. It is an open standard with a well-established, active community. A wide range of systems already implemented the WS-Agreement standard [22]. It therefore provides a good basis for further research and development.

1.1.2 Requirements of a Generic SLA Management Layer

In order to create a generic, multi-purpose SLA layer that supports a wide variety of application scenarios, protocol and language of the SLA layer need to fulfill a set of basic requirements.

Flexibility and Domain Independence of the SLA Layer

A multi-purpose SLA layer must provide a generic language that can be used to express different aspects of service level agreements. It should provide capabilities to monitor the different aspects of the service and to evaluate the associated service level objectives at service provisioning time. In order to achieve domain independence, these capabilities must not be coupled to a specific application domain. Instead they should be designed domain independent to support a wide range of use cases, for example application service provisioning (ASP), web hosting, or dynamic resource provisioning to just name a few. Therefore, it should be possible to design SLAs for arbitrary domains by using the SLA framework and language in conjunction with domain specific languages (DSL) to describe domain specific services and associated guarantees.

Protocol must support symmetric and asymmetric deployment

A multi-purpose agreement protocol must support a wide variety of deployment scenarios. In the simplest case, the service provider exposes agreement management as a separate service. Service consumers can use this service to negotiate and create SLAs. This scenario corresponds to a simple client-server deployment of the SLA management layer. However, more complex scenarios exist where such an asymmetric deployment of the SLA layer is not sufficient anymore. One example is the scenario of a service provider that wants to create an agreement with a service consumer. In a second scenario service provider and consumer require a signed copy of an agreement due to legal limitations. Moreover, both parties want to

expose their view on the fulfillment of the agreed service. A symmetric deployment of the SLA management layer is therefore required.

Must support negotiation and renegotiation of SLAs

In order to create service level agreements between service consumers and service providers, both parties must achieve a common understanding of the provided service. This common understanding can either be achieved in a static or a dynamic fashion. The static approach comprises the instantiation of services with static service levels. This approach is similar to shopping in a super market. Consumers do not customize SLA according to their needs; instead they choose the required service level for a SLA out of a predefined set. Service provider then instantiate the requested services with respect to the selected service level. However, a generic SLA management layer must also support dynamic negotiation of SLAs. Service consumers should be able to dynamically adopt the content of SLAs within certain constraints. Service consumer and provider must therefore actively exchange information in order to agree on the capabilities and restriction of the provided service. Moreover, it should be possible to renegotiate existing SLAs in order to adapt to changing requirements of the service consumer or provider.

1.1.3 Coverage of WS-Agreement

The current WS-Agreement specification already addresses big parts of these high level requirements. In order to identify the parts that are currently not covered by the WS-Agreement, the generic requirements of the SLA layer are compared with the design goals of WS-Agreement. The following list represents the design goals of the WS-Agreement protocol and language. A more detailed description can be found in section *Requirements* of the WS-Agreement specification [20].

1. Must allow use of any service term
2. Must allow creation of agreements for existing and new services
3. Must allow use of any condition specification language (to define service level objectives)
4. Must provide symmetry of protocol
5. Must be composable with various negotiation models
6. Must be standalone (independent of respectively without any negotiation model)
7. Must allow independent use of different parts of the specification

When comparing the design goals of WS-Agreement with the high level requirements of the generic SLA layer, the following requirements are already supported:

Flexibility and Domain Independence of the SLA Layer

The design goals 1 and 3 explicitly target the flexibility and domain independence of the SLA layer. By explicitly decoupling the language that is used to describe a service in a specific domain from the agreement layer (1.), WS-Agreement provides a domain independent language for describing SLAs in general. Moreover, it supports a huge degree of flexibility in terms of how service level objectives (SLO) are expressed and evaluated (3.). Since the definition and evaluation of SLOs may vary in different domains, this design goal is crucial to achieve the required flexibility on the agreement layer.

Protocol must support symmetric and asymmetric deployments

Point 4 of the WS-Agreement requirements explicitly addresses the possibility to use the WS-Agreement port types in symmetric or asymmetric deployment scenarios. WS-Agreement can therefore be used in a wide range of deployments, starting from client-server deployment up to peer-to-peer style deployment.

Must support negotiation and renegotiation of SLAs

Negotiation and renegotiation of SLAs is not an explicit goal of the WS-Agreement specification. However, (6.) states that the specification is designed to be agnostic to any negotiation model that is used in conjunction with WS-Agreement. Moreover, the definition of a negotiation protocol is explicitly defined as non-goal. (Re-)Negotiation of service level agreements is a missing concept in the WS-Agreement specification that was intentionally left out by the authors of the specification. This thesis covers this gap by proposing a generic negotiation protocol that can be used in conjunction with WS-Agreement.

1.2 WS-Agreement Overview

In this section, we provide an overview of the WS-Agreement architecture, the protocol and the relevant language concepts. We start with a brief overview of the relevant standards that are used in conjunction with WS-Agreement. Then we describe the architectural model, followed by a description of the protocol and language concepts.

1.2.1 Related Standards

The WS-Agreement specification relies on a set of well established standards like XML, SOAP, WSDL and WSRF. In the following an overview and a short explanation of the relevant standards are given.

The Extensible Markup Language (XML) defines a language to hierarchically structure data in a text based format. Its main purpose is to exchange data between computer systems in a platform and implementation independent way. Moreover, XML is designed to be a simple, general data format that supports a wide range of applications and is usable over the internet. XML is standardized by the World Wide Web Consortium (W3C) as a W3C recommendation [23]. Documents that adhere to the syntactic rules defined by the XML specification are called well-formed. Additionally it is possible to associate a certain grammar with an XML document that defines the structure of the data contained in the document. A document that adheres to the provided grammar is called valid.

XML schema is a way to define a set of rules that describe the structure and the content of a XML document. The elements of an XML document must adhere to these rules in order to be valid. The XML Schema specification is published as W3C Recommendation [24] [25].

SOAP [11] is a network protocol for exchanging structured information and to realize Remote Procedure Calls (RPC) in computer networks. XML is used as data format. SOAP defines an extensible messaging framework that is usable with a variety of transport protocols.

The Web Service Description Language (WSDL) defines a platform and implementation independent format to describe service interfaces. These interface descriptions comprise a definition of the methods exposed by a service, the definition of the method parameters, the

return type, faults, the exchanged messages, and the protocol binding. WSDL is specified by the W3C in [9].

The Web Service Resource Framework (WSRF) comprises a set of specification that are defined and maintained by the Organization for the Advancement of Structured Information Standards OASIS [26]. The goal of WSRF is to "define an open framework for modeling and accessing stateful resources using Web services" [27]. The specifications comprise the following:

- *WS-Resource*: defines a web service resource and how to access it
- *WS-ResourceProperties*: defines how to access the properties of a WS-resource
- *WS-ResourceLifetime*: defines how to manage the lifecycle of a resource
- *WS-ServiceGroup*: defines how to store WS-resources in collections (registries)
- *WS-BaseFaults*: defines a basic data format for fault messages

In the following sections, a brief overview of WS-Agreement is given. First of all, the WS-Agreement protocol is described. The protocol defines the WS-Agreement services and their associated methods. Then the WS-Agreement language is described. The language defines the data structures that are used in WS-Agreement.

1.2.2 The WS-Agreement Model

WS-Agreement extends the classical service discovery and usage model since it allows service consumers not only to discover and use services, but also to dynamically negotiate the quality with which the service is provided. From a conceptual point of view, one can distinguish between the agreement layer and the service layer. The service layer focuses on the functionality provided by the service, while the agreement layer focuses on the quality of the provided service. The agreement layer introduces a new entity on side of the service provider and consumer, the agreement management. In order to discover services with a specific QoS a service consumer now contacts the agreement management component of its domain. The agreement management component looks up available services in a public registry. When an adequate service was found the consumer's agreement management component contacts the agreement management of the service provider. Then the conditions of the service provisioning are negotiated at the agreement layer. Once the service consumer and the service provider achieved a common understanding of the service provisioning, a SLA is created that serves as a formal contract between the two parties and describes the rights and obligations of each party in the context of the service provisioning process. The service provider now instantiates the agreed service at the service layer. The service adheres to the nonfunctional properties defined in the SLA. Once the service is in place, the service consumer can start to use it. During the provisioning of the service, the compliance of the SLA is monitored at the agreement layer. Figure 3 shows the WS-Agreement architectural model and the interactions at the different layers.

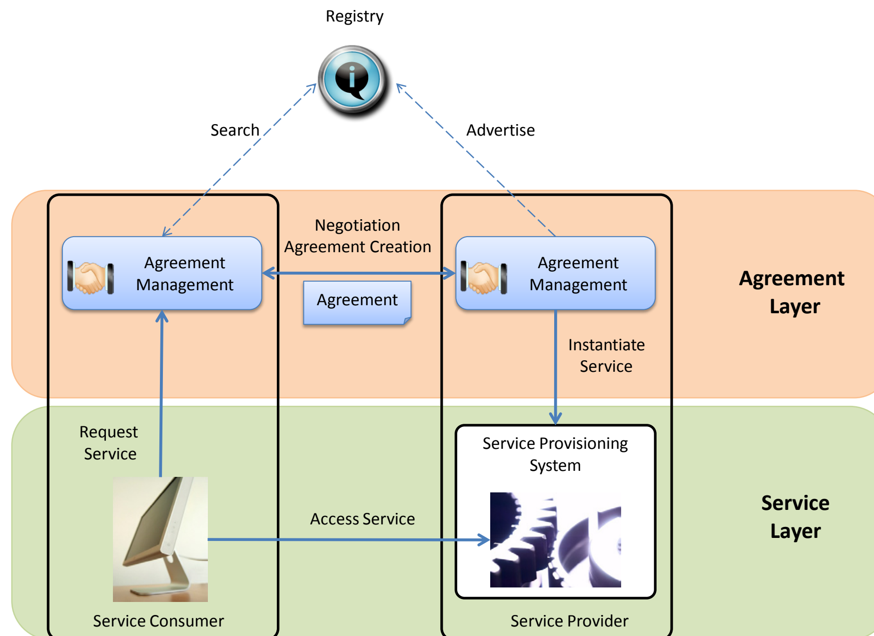


Figure 3: SLA aware service discovery

1.2.3 The WS-Agreement Protocol

The WS-Agreement protocol defines the required services and operations to create and monitor service level agreements in distributed systems. The WS-Agreement model therefore defines two types of services, the agreement factory service and the agreement service. The agreement factory service is responsible for creating agreements between a service consumer and provider and for instantiating the associated service with the agreed QoS. The agreement service is responsible for monitoring the compliance of agreements and of the associated services. Figure 4 gives an overview of the layered service model in WS-Agreement.

The WS-Agreement specification specifies a set of interfaces in order to interact with the agreement factory service and the agreement service. These interfaces descriptions are provided in the form of WSDL. One specific interface is defined by one WSDL port type. For the agreement factory service the *AgreementFactory* port type and the *PendingAgreementFactory* port type are defined. For the agreement service the *Agreement* port type and the *AgreementState* port type are specified. Both services are modeled as web service resources conforming to the Web Service Resource Framework [27] specification. A web service resource is a web service instance that is uniquely identified by an endpoint reference (EPR). Endpoint references are defined in the WS-Addressing specification [28]. WSRF defines the methods to access the properties of a web service resource, to manage its life cycle, to organize web service resources in groups, and to provide extensible fault mechanisms.

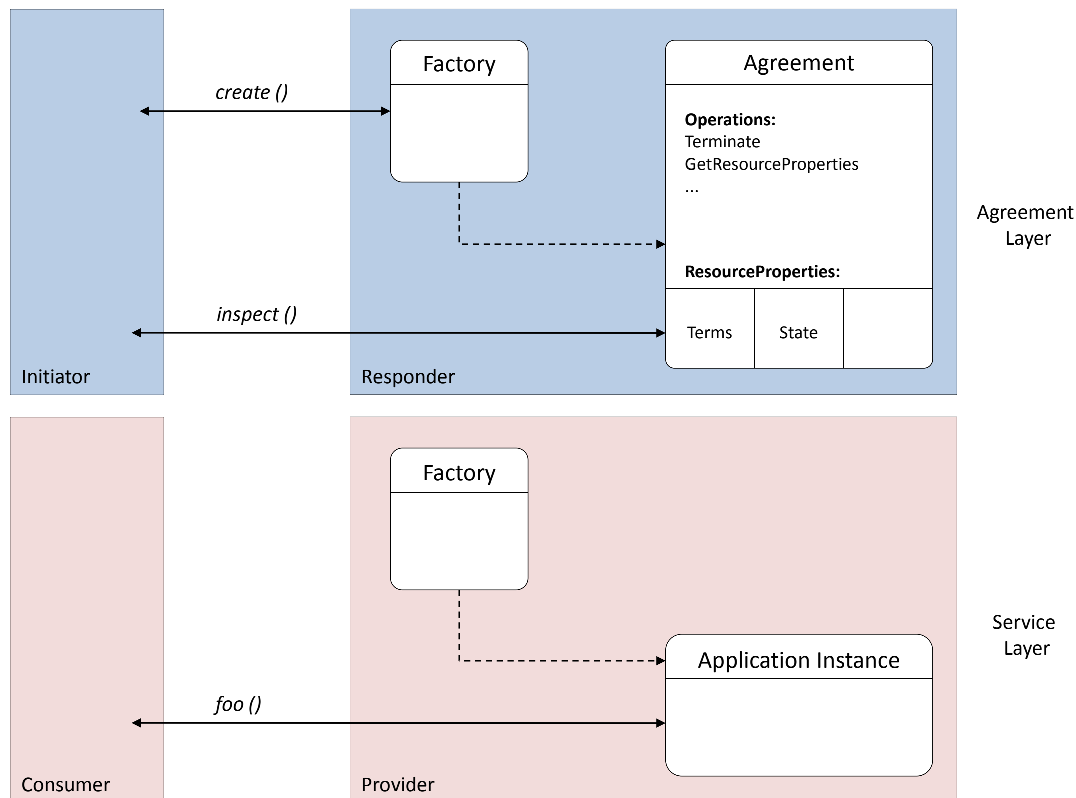


Figure 4: layered service model of WS-Agreement [20]

1.2.3.1 The Agreement Factory Service

As described before, an agreement is a bi-lateral, contractual relationship between a service provider and a customer. The agreement describes the service to deliver, the service level objectives, and the responsibilities of each party. In the context of WS-Agreement, the party that creates an agreement is called agreement initiator and the party that responds to the agreement creation request is called agreement responder. In order to describe the responsibilities of each party in the service delivery process, the roles agreement initiator and agreement responder are associated with the service consumer and service provider role. Depending on the scenario, this mapping can vary. In the following it is assumed that the agreement initiator is the service consumer and the agreement responder is the service provider.

In the WS-Agreement model the agreement factory service is the component that is used by the agreement initiator to create SLAs with the agreement responder. The initiator expresses its requirements in the form of agreement offers. An offer describes the service to provide, guarantees that are associated with the service, compensation methods for fulfilling or violating the guarantees, and the rights and obligations of each party. Since the content of an agreement offer can potentially be very complex, WS-Agreement defines a simple template mechanism that guides an initiator in creating agreement offers. The service provider therefore publishes a set of agreement templates that can be used by initiators to create new offers. A template is essentially a prototype of an agreement offer, as described by the prototype design pattern [29]. It specifies the structure and the content of an agreement offer, along with a set of constraints that valid agreement offer must adhere to. The agreement

factory service publishes the agreement templates it supports as WSRF resource properties as defined in [30]. In order to create a new agreement, the service consumer first queries the available templates from the service provider's agreement factory service using the WSRF GetResourceProperty method. The consumer chooses the template that fulfills its requirements best and creates a new agreement offer from it. It then changes the offer according to its requirements and sends it to the service provider in order to create a new agreement. As soon the agreement offer is sent to the agreement responder, the initiator is bound to it. WS-Agreement defines two methods for creating agreements with the agreement factory service, the synchronous *createAgreement* method that is specified by the AgreementFactory port type and the asynchronous *createPendingAgreement* method that is specified by the PendingAgreementFactory port type. In the synchronous case the agreement responder must immediately decide whether to accept or reject an incoming agreement offer. If it accepts the offer it returns a reference to a new agreement instance, otherwise it returns an error. This process is shown in Figure 5. In the asynchronous case the decision of accepting the agreement offer is deferred to a later time. The createPendingAgreement method therefore returns a reference to a new agreement instance which is in the pending state. Once the acceptance decision is made, the agreement state is changed either to observed in case of acceptance, or to rejected. Optionally, the agreement initiator is notified of the responder's decision. The AgreementFactory port type and the PendingAgreementFactory port type implement the abstract factory pattern as described in [29].

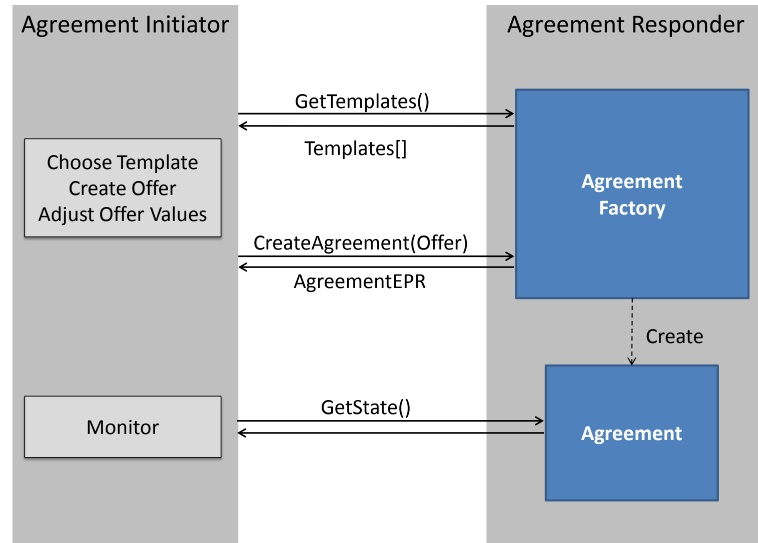


Figure 5: agreement creation using the agreement factory port type

The architectural features of the agreement factory service can be summed up as follows:

1. Agreement factory service is modeled as WSRF resource
2. Creation of agreement offers based on templates
3. Synchronous creation of agreements via the AgreementFactory port type
4. Asynchronous creation of agreements via the PendingAgreementFactory port type

1.2.3.2 The Agreement Service

The agreement service implements the required mechanisms to access the content of an existing agreement, to monitor the agreement at runtime and to manage its lifecycle. For each agreement that is created by the agreement factory service a new agreement service instance is instantiated. The agreement service is modeled as WSRF resource and the properties of an agreement service instance are modeled as WSRF resource properties. The WSRF resource properties can be accessed via the methods specified in the WSRF Resource Properties

specification [30]. WS-Agreement defines two port types for the agreement service, the *Agreement* port type and the *AgreementState* port type. The Agreement port type defines a method to terminate an agreement if permissible and a set of agreement resource properties, such as the agreement context, terms and id. The agreement attributes are in general specified by the agreement offer. The AgreementState port type defines an additional set of agreement resource properties in order to support the monitoring of an agreement. These additional resource properties comprise the state of the agreement, the state of the distinct service terms, and the state of the guarantee terms of an agreement. Figure 6 illustrates the possible states of an agreement instance.

The service states and guarantee states are fundamental for monitoring SLAs. Service term states provide the basic mechanisms to couple service monitoring with SLA monitoring and evaluation. They may include relevant information from the service monitoring, for example average response time of a service in the last 15 minutes of the current state of a provided resource. Based on this information a SLA management system can assess the guarantees of an agreement. Moreover, the service term states can include service deployment information. As soon as a dynamically provided service is deployed for a consumer, the service provider can expose the corresponding service deployment information as part of the service term states. By that, a dynamic coupling of agreement and service layer is achieved.

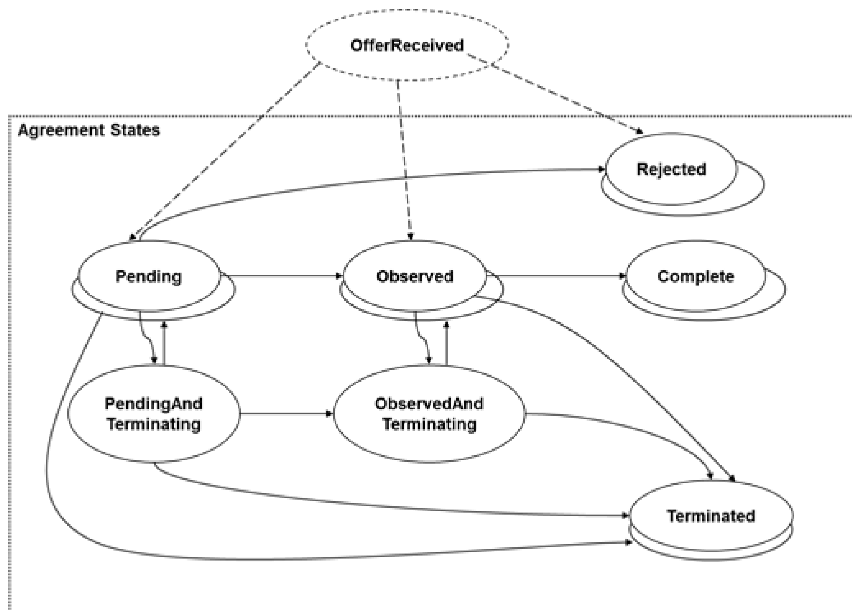


Figure 6: states exposed by an agreement [20]

The architectural features of the agreement service can be summed up as follows:

1. Agreement service is modeled as WSRF resource
2. Agreement port type defines method for terminating an agreement
3. General resource properties are defined by the Agreement port type
4. Resource properties for monitoring are defined by the AgreementState port type

1.2.3.3 Protocol Symmetry

WS-Agreement protocol supports the symmetric deployment of its port types. In peer-to-peer style, agent based scenarios each entity may either act as agreement initiator or as agreement responder. Therefore, each entity needs to implement a corresponding agreement factory service. In such a scenario, an agent either acts as agreement initiator that creates new agreements with other agents, or as agreement responder that is contacted by other agents.

Since WS-Agreement does not prescribe whether the agreement initiator or the agreement responder is the provider of an agreed service, a huge degree of flexibility is achieved.

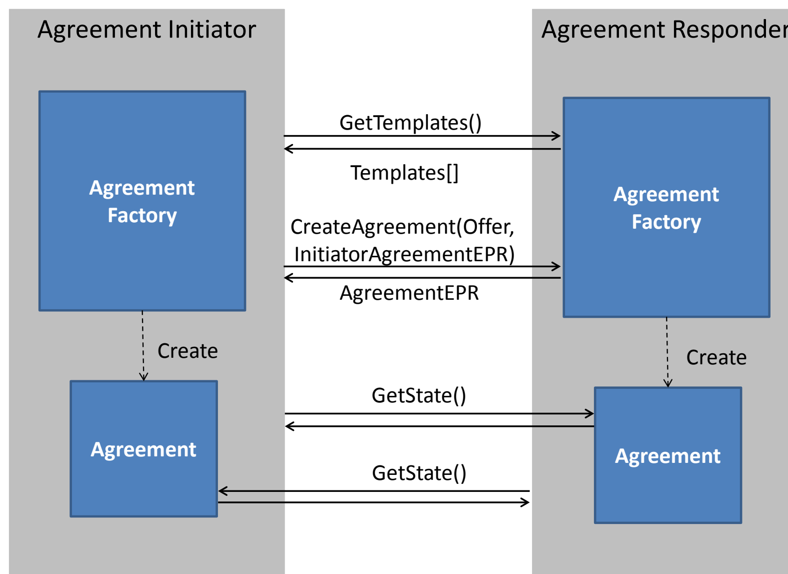


Figure 7: symmetric deployment of agreement resources to support bilateral agreement monitoring

In other scenarios, the bi-lateral monitoring of agreements might be critical, for example to identify and solve disputes in service provisioning process. The WS-Agreement protocol supports this through the symmetric deployment of the agreement port types. In this process an agreement initiator queries the templates from the responder's agreement factory service. It selects the best suited template and creates an agreement offer based on it. Then, the initiator creates a new agreement resource in its domain, the initiator agreement. This agreement reflects the content of the agreement offer and since it is not accepted by the responder yet, it remains in pending state. Now, the initiator starts the agreement creation process. It therefore sends the agreement offer to the agreement responder, along with the endpoint reference to the initiator agreement. If the responder accepts the agreement offer, it creates a new agreement resource in its domain (responder agreement) and returns the endpoint reference to the initiator, otherwise it returns an error. As soon as the agreement initiator gets informed of the agreement acceptance, it changes the state of the initiator agreement to *Observed*. In case the agreement offer was rejected, the initiator agreement state must be changed to *Rejected*. If the agreement was created successfully, both parties have a reference to the opposite agreement. Therefore, mutual monitoring of the agreement is possible. Figure 7 illustrates this process. The state of the agreement can either be replicated between the parties, or domain specific monitoring mechanisms can be implemented. Such monitoring mechanisms could expose the different views of the agreement initiator and responder on the service provisioning process. In case of an agreement that guarantees a minimal service response time, the service provider may determine the agreement state by asserting that the required resources for providing the guaranteed service quality are available, while a service consumer measures the response time of the service in its domain. Differences in the agreement states may be solved by the parties in domain specific ways, for example by dispute handling.

1.2.4 The WS-Agreement Language

While the WS-Agreement protocol defines the services and methods required for creating and monitoring agreements, the WS-Agreement language defines the data types for expressing the

content of an agreement. The WS-Agreement language is defined independent from the WS-Agreement protocol and can therefore be used in a wide set of scenarios, for example with other protocol bindings. The agreement language is defined in form of an XML schema. It describes the data types and the structure of the WS-Agreement core documents such as the Agreement document, the Agreement Template document, and the Agreement Offer document. The WS-Agreement specification defines two separate schemas, the agreement schema and the agreement state schema. The agreement schema defines the WS-Agreement core data types. The agreement state schema includes the data types for the dynamic agreement monitoring, namely the agreement states, service term states and guarantee term states.

Figure 8 depicts the basic structure of an agreement. An agreement contains an agreement identifier, its name, an agreement context, and a term compositor with a detailed description of the service to provide. The following sections give an overview of the most important language constructs of WS-Agreement. We start with a description of the agreement context. Then we continue with the concept of agreement terms and term composition and conclude with a discussion of agreement templates and creation constraints.

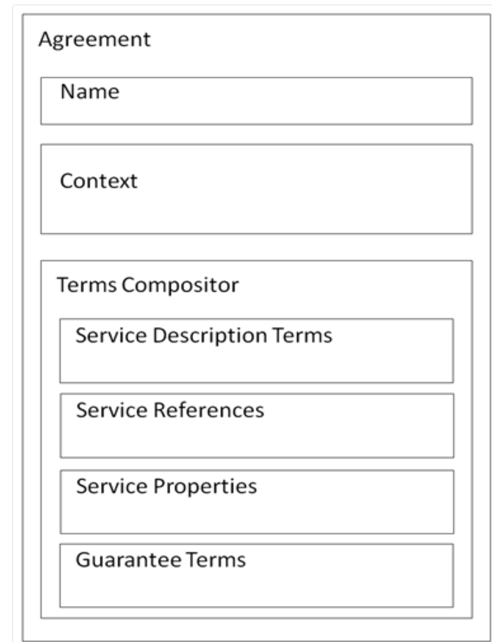


Figure 8: structure of an agreement [20]

1.2.4.1 Agreement Context

The agreement context contains a set of metadata that is associated with an agreement. First of all, the context provides information on the parties that created the agreement. The context therefore contains two elements: *Agreement Initiator* and *Agreement Responder*. The content of these two elements is not further specified. They can contain an arbitrary, domain specific description of each party in order to resolve them to real world entities. Such a description can, for example, be an endpoint reference or a distinguished name that identifies the party in a security context. Additionally, the agreement context associates each party with its respective role in the agreement by specifying which party is the service provider and which is the service consumer. By that, the involved parties in the agreement are identified and bound to their roles. The agreement context may also contain an expiration time that defines how long an agreement (and an associated service) is valid. The context should also reference the template that was used to create the agreement. This is of particular importance for the agreement offer validation process. Besides that, the agreement context may contain any domain specific data.

1.2.4.2 Agreement Terms

The most important requirement of a SLA language is the capability to describe the services to deliver and their associate guarantees. For this purpose the WS-Agreement language defines so called agreement terms. The agreement terms split up into two groups, *Service*

Terms and *Guarantee Terms*. Service terms describe the different aspects of a service, while guarantee terms, as indicated by their name, specify the guarantees that apply to a service along with compensation methods for fulfilling or violating these guarantees.

In general, each term in WS-Agreement is identified by a term name. Service terms are a subset of the agreement terms that additionally contain a service name. The service name can be used to semantically group multiple service terms in an agreement. A single service in an agreement can therefore be described by multiple service terms. Each service term describes a different aspect of the service. Service terms are Service Description Terms, Service References, and Service Properties.

Service Description Terms

Service Description Terms comprise a functional description of the service to provide. Therefore, the service description terms contain a domain specific description of the service. This can comprise a complete or a partial description. Since the WS-Agreement is designed to be domain independent, the content of a service description term can be any valid XML document. Both parties involved in the agreement (agreement initiator and agreement responder) must understand the domain specific service description, e.g. the XML schema for the domain specific language must be known to both parties.

Service References

Service References provide a way to refer to existing services within an agreement. This can be useful when a certain service quality should be provided for an existing service rather than for a new one. Similar as in service description terms, service references can contain an arbitrary XML document that describes the service reference. Here the same rules and restrictions apply as for service description terms.

Service Properties

Service Properties are the last type of service terms. From an abstract point of view, they provide a way to define variables in the context of an agreement. A variable definition comprises a name, a metric, and a location. The location refers to a distinct element in the agreement, e.g. by using an XML query language such as XPath [31]. Service Properties are used to define and evaluate guarantees in WS-Agreement.

Guarantee Terms

Guarantee Terms are the second group of agreement terms. They provide the required capabilities to express service guarantees in agreements, define how guarantees are assessed and which compensation methods apply in case of meeting or violating the service guarantees. Guarantee terms consist of four elements: a Service Scope, a Qualifying Condition, a Service Level Objective and a Business Value List.

The *Service Scope* specifies which services in the agreement are covered by the guarantee. Since a single agreement can comprise a set of different services, one guarantee may apply to one or more services.

The *Qualifying Condition* specifies preconditions that must be fulfilled before a guarantee applies. Not all guarantees apply during the whole lifetime of an agreement. The qualifying condition specifies the preconditions that must be met before a guarantee is evaluated.

The *Service Level Objective* (SLO) defines an objective that must be met in order to provide a service with a particular service level or with a particular quality of service (QoS). The QoS properties of a service are defined in the service description terms of an agreement. These are the agreed service properties. At the service provisioning time the actual QoS properties are derived from the service monitoring system. This service level objective essentially defines how the agreed QoS properties are related to the actual QoS properties. It defines a logical expression that can be assessed in order to determine the fulfillment of a guarantee.

The *Business Value List* defines the penalties and rewards that are associated with a guarantee. WS-Agreement already defines a model to express business values for guarantees. These predefined business values range an abstract importance of a guarantee to arbitrary value expressions, such as Euro or US-Dollar.

1.2.4.3 Term Composition

Besides the definition of service and guarantee terms, the WS-Agreement language provides a simple grammar to structure these terms in an agreement. WS-Agreement defines therefore a XML data structure that can easily be interpreted and that supports the composition of agreement terms in a flexible way. The different agreement terms are structured in an expression tree, the so called *term tree*. The expression tree implements the composite pattern [29]. It contains a so called term compositor. The term compositor represents a non-terminal of the WS-Agreement language. WS-Agreement defines three types on term compositors: the *All* compositor, the *OneOrMore* compositor, and the *ExactlyOne* compositor. These term compositors correspond to the logical AND, OR, and XOR functions. A term compositor can contain zero or more term compositors or agreement terms, but it must contain at least one element. A term compositor can also contain agreement terms, which are the terminals in the WS-Agreement language. Agreement terms are service description terms, service references, service properties, and guarantee terms.

Through the flexibility of the language, WS-Agreement supports a wide range of usage scenarios. Agreement responder can use the language to easily express valid combinations of services, guarantees, etc. in agreement templates. Agreement initiators can use this feature to describe alternate services they would like to purchase, thus leaving more flexibility to the agreement responder when creating the agreement.

1.2.4.4 Agreement Templates and Creation Constraints

As mentioned before, WS-Agreement anticipates a template mechanism for creating agreements. Agreement templates help the agreement initiator to create agreement offers that match the expectations of the agreement responder. They define the structure of an agreement offer, specify valid combinations of agreement terms, include a description of the different aspects of a service, and define guarantees and compensation methods. Agreement initiators may process agreement templates in order to find valid service combinations for an agreement offer.

Additionally, agreement templates may also contain a *Creation Constraint* section. Creation Constraints are a very important concept in WS-Agreement. They define a set of rules and restrictions that agreement offers must adhere in order to be compliant with a template. Agreement offers that adhere to the creation constraints defined in a template are subsequently called *valid agreement offers* or *compliant agreement offers*. WS-Agreement defines two different kinds of creation constraints, *Offer Items* and *Free Form Constraints*. Offer items are the default way to describe creation constraints in an agreement template. They can be used to restrict structure and values of valid offers. Free form constraints are an extension mechanism. They enable implementations to include custom creation constraints that use other restriction models and validation methods than the ones anticipated by WS-Agreement.

A valid agreement offer must comply to the creation constraints defined in a template in order to be accepted by the agreement responder. However, the creation constrains defined in a template only constitute hints what kinds of offers an agreement responder will accept. They do not imply promises that a responder will accept a compliant offer. The acceptance of an agreement offer still depends on the local acceptance policy of the agreement responder. On the other hand, the agreement responder may also accept agreement offers that are not based on a specific template, when permitted by the local policy.

1.3 Summary

In this chapter we introduced the general concept of a service-oriented architecture for distributed system design. Today, service-oriented architectures are often implemented using web-service technologies. We discussed basic concepts of a web-service based architecture, such as service discovery, binding and invocation. In that context we described the relevance of quality of service in service-oriented architectures. Service level agreements are a way to define such quality of service in a dynamic service delivery process. Then we examined existing approaches for SLA management in distributed systems. We motivated the decision for WS-Agreement as the SLA management standard for this thesis. Subsequently, we defined general requirements for a generic SLA layer in a Grid. We compared these requirements with the capabilities already provided by WS-Agreement and identified the missing negotiation capabilities as a gap that is addressed in the following chapter. Finally, an overview of the WS-Agreement standard was given, including an overview of related standards, the WS-Agreement architectural model, the relevant port types, and the basic language constructs.

Chapter 2
NEGOTIATION OF SERVICE LEVEL AGREEMENTS

In Chapter 1, WS-Agreement [20] was introduced as a standard for creating and monitoring service level agreements in distributed systems. In this chapter we present the WS-Agreement Negotiation specification [32], which adds additional negotiation and renegotiation capabilities on top of the WS-Agreement specification. This chapter therefore presents the outcome of work conducted in the OGF GRAAP working group in order to standardize an agreement negotiation and renegotiation protocol and the required term language for negotiation and renegotiation processes. WS-Agreement Negotiation is intended to be used in conjunction with WS-Agreement. It describes the basic agreement negotiation and renegotiation concepts, defines the components that are involved in negotiation processes, and specifies their interfaces as well as the relevant data structures.

In distributed service-oriented systems different services are offered by service providers and used by service consumers. Service consumers use these services as they or compose them in order to provide new services with added functionality. Since services are often acquired on demand, service consumers need to predict the behavior of these services before they actually acquire them. This problem leads to a situation in which service consumers do not only have functional requirements for a service, but also demands regarding to the non-functional service properties, such as the average response time of a service, the service availability, or the average recovery time in case of failure. They need standardized ways of defining the required service properties, and guarantees of the service provider to deliver a service with the defined quality, capabilities to monitor the service properties at provisioning time, and enforcement mechanisms in case a service was not provided with the agreed service quality. Service level agreements are one approach to solve this problem. They are bilateral contracts between a service provider and a service consumer that describe the service to provide and define guarantees regarding the quality with which this service is provided.

WS-Agreement is one approach for using service level agreements in distributed service-oriented environments. It allows service consumers to dynamically create service level agreements with service providers in order to acquire services with a well-defined quality of service. Moreover, it defines the basic mechanisms to monitor the state of an agreement and to evaluate the guarantees that are associated with an agreement. WS-Agreement supports the agreement creation over a template mechanism. Service providers can offer their services in the form of agreement templates. These template guides service consumers in the process of creating a valid agreement offers. An agreement template may, for example, contain a number of alternative service descriptions, where each service description offers the same service with a different service quality. In that way the same service is for example offered with 99.9%, 99% and 98% availability. The service consumer can choose the service offering that fulfills

its requirements best and create a new agreement with the service provider. This approach is comparable to a super-market, where consumers choose the desired product out of a set of available products. Even though the template approach is sufficient for a wide range of application scenarios, there are still a number of scenarios that requires more flexible and dynamic negotiation capabilities, for instance multi-round negotiation capabilities. A typical example is the negotiation of a service provisioning time in co-allocation scenarios, the renegotiation of existing agreements in order to cope with peaks in a service usage, or the negotiation of related service parameters such as the number of resources that is provided by a service and the price of the service. WS-Agreement Negotiation adds the required functionality for agreement negotiation on top of the WS-Agreement specification. It can therefore be used in conjunction with WS-Agreement without breaking existing systems.

In the WS-Agreement Negotiation model negotiation is done in the context of a separate negotiation processes. A negotiation process represents a relationship between a service consumer and a service provider in order to dynamically exchange information with the goal of creating a valid agreement offer that subsequently leads to an agreement. Negotiation processes are created by a *Negotiation Factory*, which implements the *Negotiation Factory Port Type*. A negotiation process is represented by a *Negotiation Instance*, which implements the *Negotiation Port Type* and optionally the *Advertisement Port Type*. The negotiation port type defines the basic properties of a negotiation instance, a method for exchanging offers and counter offers, and a method to terminate the negotiation process. The advertisement port type additionally specifies a method to notify a negotiation participator of a specific offer. The basic components involved in a negotiation process are depicted in Figure 9.

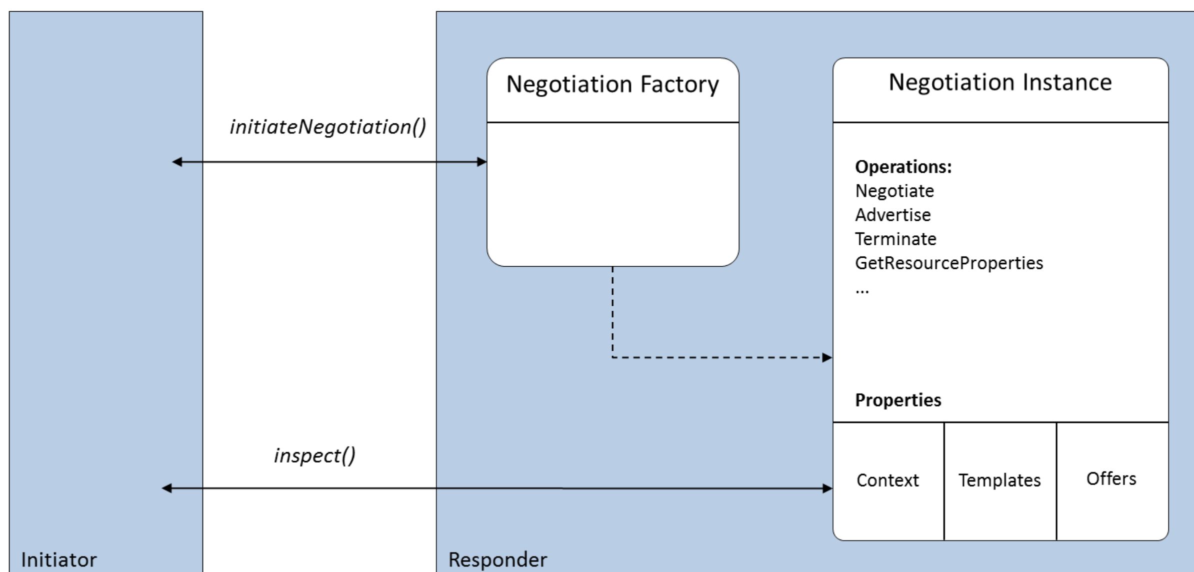


Figure 9: overview of the WS-Agreement Negotiation components.

The remainder of this chapter is structured as follows. In section 2.1, the requirements and limitations of WS-Agreement Negotiation are described. Section 2.2 introduces the terms used in the specification and section 2.3 describes a set of negotiation use cases in more detail. The negotiation model is described in section 2.4. It consists of two parts, the description of the negotiation offer/counter offer model and the description of the layered negotiation model. In section 2.5, the properties of the negotiation instance are described. The

structure of negotiation offers and counter offers is then described in section 2.5.2. In section 2.6, we describe how the negotiation layer is finally coupled with the agreement layer and how negotiated and renegotiated agreements are created. In section 2.7, we discuss possible deployment scenarios of the WS-Agreement Negotiation port types. The detailed specification of the port types and operations can be found in the Appendix.

2.1 Goals and Requirements

The WS-Agreement Negotiation defines a set of requirements that are covered by the specification as well as a set of limitations which are considered out of scope. The requirements and limitations are described below:

Requirements

- *Must build on top of the WS-Agreement specification*

  WS-Agreement Negotiation must work seamlessly with WS-Agreement. Therefore, the WS-Agreement language must be used to define re-/negotiation offers and to express negotiation constraints. Moreover, the protocol must be defined as an extension to the WS-Agreement protocol. It must still be possible to use other negotiation protocols with an agreement factory.

- *Must allow negotiation of new and renegotiation of existing agreements*

  The protocol must specify the required interfaces to negotiate new and to renegotiate existing agreements. In the context of this specification, (re-)negotiation of agreements is considered to be a bilateral process, which results in a (re-)negotiated agreement. The specification must define the basic capabilities to create (re-)negotiated agreements based on (re-)negotiation offers.

- *Must provide a symmetric protocol*

  There are a wide number of negotiation scenarios, depending on whether a service consumer or a provider initiates the negotiation process, which party creates the negotiated agreement, and where the resulting agreement state is hosted. The same applies to renegotiation scenarios. The interfaces defined in this specification must therefore support symmetric and asymmetric protocol layouts in order to support various usage scenarios.

- *Must provide a negotiation state machine*

  The specification must provide a simple state machine that describes valid state transitions of negotiation/renegotiation offers and counter offers.

- *Must support binding and non-binding negotiations*

  The specification must be usable in binding and non-binding (re-)negotiation scenarios. By default, this specification treats (re-)negotiation as a non-binding process. Binding negotiations are expected to be defined as an extension to this specification.

Out of Scope

- *Definition of compensation methods for negotiated offers*

  Even though binding (re-)negotiation of agreements is in principle foreseen by this specification, there is no compensation model defined for this type of negotiation. It is expected that such models will appear as domain specific extension to this specification.

- *Definition of Auction Protocols*

  This specification focuses on the bilateral (re-)negotiation of agreements. Since auction protocols are one-to-many negotiations they are regarded as alternative negotiation approach.

2.2 Notational Conventions and Terminology

RFC 2119 [33] specifies the best practices to signify specification requirements in ITEF documents. The WS-Agreement Negotiation specification adheres to these best practices by using the key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” as specified. The following definition of these key words is given in RFC 2119:

MUST - This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification.

MUST NOT - This phrase, or the phrase "SHALL NOT", mean that the definition is an absolute prohibition of the specification.

SHOULD - This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

SHOULD NOT - This phrase, or the phrase "NOT RECOMMENDED" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

MAY - This word, or the adjective "OPTIONAL", mean that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides.)

Besides the key words to indicate requirements the following terms are used throughout the specification:

Negotiation

Negotiation is a process between an agreement initiator and an agreement responder to reach an acceptable agreement offer from an initial agreement template. Agreement offer negotiation is a non-binding, bi-lateral process that comprises exchange of information in order to find a consensus for acceptable agreement offers.

Negotiation Offer

A negotiation offer is a non-binding proposal for an agreement offer made by one negotiation party to another. Negotiation offers are used to dynamically exchange information in order to reach an acceptable agreement offer. Zero or more negotiation offers may precede a binding agreement offer as defined in the WS-Agreement specification. Negotiation offers describe the services of a SLA, the quality of service properties, and the associated guarantees. Negotiation offers may also contain negotiation constraints that restrict the negotiable terms and their value spaces.

Negotiable Template

Negotiable templates are provided by a negotiation participator in the context of a particular (re)negotiation process. They define which types of agreement offers can be negotiated, the basic structure of these offers, and the basic constraints that each offer must adhere to.

Negotiation Counter Offer

Negotiation offers that are created on the base of a previous negotiation offer are called Negotiation Counter Offers. Counter offers must adhere to the negotiation constraints of the offer they are related to. In a negotiation process each negotiation offer is either created on the base of an agreement template or on the base of another negotiation offer. In the context of this specification the term counter offer describes a negotiation offer that is based on another negotiation offer. It therefore reflects the relationship of a negotiation offer to the offer that it is based on.

Negotiated Offer

The term negotiated offer describes an offer that has reached the acceptable state. Negotiated offers can be used as valid agreement offers in order to create new agreements or to replace existing agreements.

Agreement Initiator

The agreement initiator is the entity in a negotiation process that creates an agreement based on a negotiated offer. This role corresponds to the *agreement initiator* role as defined in the WS-Agreement specification.

Agreement Responder

The agreement responder is the entity in a negotiation process that responds to an agreement creation request based on a negotiated offer. This role corresponds to the agreement responder role as defined in the WS-Agreement specification.

Negotiation Initiator

The negotiation initiator is the party that initiates the negotiation process. It acts on behalf of the agreement initiator or the agreement responder. The negotiation initiator invokes the negotiation responder's initiateNegotiation method, which is defined in this specification.

Negotiation Responder

The negotiation responder is the party in a negotiation process that responds to an initiate-Negotiation request. It acts on behalf of the agreement initiator or the agreement responder. The negotiation responder implements the NegotiationFactory and Negotiation port types defined in this specification.

Negotiation Participator

The negotiation participator is an entity that takes part in the negotiation process. The negotiation participator is either the negotiation initiator or the negotiation responder.

Negotiation Context

The negotiation context defines the type of the negotiation, identifies the negotiation participators, their roles and responsibilities, and optionally specifies additional domain specific negotiation parameters, such as maximum of negotiation rounds or expiration time.

Negotiation Offer Context

The negotiation offer context represents metadata associated with a specific negotiation offer. It contains information such as the id of the originating negotiation offer, its expiration time, and its state. It may also contain domain specific extensions in order to define augmented negotiation protocols.

Negotiation Constraints

The negotiation constraints are a method to control the negotiation process. A negotiation participator uses negotiation constraints in order to define structure and value spaces for compliant negotiation counter offers. Negotiation constraints are therefore used to express the requirements of a negotiation participator.

Negotiation Offer State

The negotiation offer state describes the specific state of a negotiation offer. It may include domain specific data that is used by the negotiation participators to exchange state-specific information and to advance the negotiation process. The reason for rejecting a negotiation offer is an example for such state-specific information.

Namespaces

This chapter comprises a set of XML code examples in order to visualize XML data structures. The following is an example for XML or other code:

```
http://schemas.ogf.org/graap/2009/11/ws-agreement-negotiation
```

The following namespaces are used in XML code examples in this chapter:

Prefix	Namespace
wsag-neg	http://schemas.ogf.org/graap/2009/11/ws-agreement-negotiation
wsag	http://schemas.ggf.org/graap/2007/03/ws-agreement
wsa	http://www.w3.org/2005/08/addressing
wsrf-rp	http://docs.oasis-open.org/wsrf/rp-2
wsrf-rw	http://docs.oasis-open.org/wsrf/rw-2
xs/xsd	http://www.w3.org/2001/XMLSchema
xsi	http://www.w3.org/2001/XMLSchema-instance
wsdl	http://schemas.xmlsoap.org/wsdl

2.3 Use Cases

WS-Agreement Negotiation supports a wide set of use cases where two parties need to dynamically exchange information in order to reach an agreement. One example is the dynamic parameterization of complex services. Another typical example is the reservation of computational resources, which is described in the following. In this scenario a service provider offers computational resources to its customers, which can be reserved for specific time frames. It provides a job submission service to access the reserved resources, and a portal application to manage the job submission service. The job submission service is implemented as a web service that provides the required methods for submitting and managing computational jobs, such as submit a job, start a job, query the state of a job, and cancel a job. These methods are exposed via the Web Service Description Language (WSDL). The portal application provides additional methods to manage the job submission service, such as update the profiles of registered users, query the current resource availability, query usage data for the provided resources, deploy a new application, or manage the storage on the resources.

Agreements that comprise the advance reservation of computational resources define ongoing relationships between a resource provider and a resource consumer. They constitute the general conditions for jobs that are subsequently executed in the context of the agreement. The resource provisioning model is thereby implementation specific; whether resources are

exclusively dedicated to a user, prediction models or preemption is used is up to the resource provider.

The computational resource provider offers available resources via an agreement template. The template includes the service description and a set of service levels possible service levels. The service description contains the specification of the available computational resources and the timeframe in which these resources are available. The offered resources may differ in hardware; e.g. they may have different CPU architectures, CPU speed, memory, or hard disk space. The service consumer may compose the offered resources in order to satisfy its needs. Moreover, the customer can select the desired service levels for resource availability, and availability and average response times of the job submission service and the portal application. The availability of the job submission service is for example 95%, 98%, 99% or 99.9%. It is defined as the probability that a request is processed within 15 seconds. For the average response time of the job submission service, the customer may select a value of 0.5, 1, or 2 seconds and the number of requests per minute for which this guarantee must hold. These QoS parameters can be specified separately for the job submission service, the portal application, and the reserved resources. The pricing of the overall service is dependent on the selected computational resources and the selected QoS levels.

The described template provides many possibilities to parameterize the computational resource service. Moreover, it contains dynamic parameters, such as pricing, that are dependent on the chosen resources and the QoS guarantees. Once the consumer filled in all its requirements, it sends the offer to the resource provider. The provider then checks whether it is capable to provide the requested service. In case the requested resources are available, the provider sends back a completed counter offer with the updated pricing information. The customer can now choose to create an agreement based on this negotiated offer. If the resource provider is not capable to fulfill the requirements stated in the negotiation offer, it can also send back a counter offer indicating an alternative service that can be provided instead. For example, the service customer has requested 128 nodes with 8GB memory in a given timeframe, but the resource provider could not fulfill this request at this time. Instead the provider sends back a counter offers for 96 nodes with 8GB memory and 32 nodes with 6GB memory for a lower price. The consumer may choose to accept the counter offer, to reserve only the 96 nodes that meet its requirements and to purchase the remaining capacity somewhere else. The process of filling in all required fields of a negotiation offer may take multiple rounds.

At a later point in time, the customer may recognize that it requires more or less resources to efficiently complete its computation. In that case it may start a renegotiation of the agreement in order to scale the resources up or down, according to its requirements.

2.4 WS-Agreement Negotiation Model

In this section we describe the WS-Agreement negotiation model. The model consists of two parts, the Negotiation Offer/Counter Offer model, and the layered architecture model. The Negotiation Offer/Counter Offer model describes the dynamic exchange of information in order to reach an acceptable agreement offer that can be used subsequently to create a new agreement, or to create a renegotiated agreement respectively. The layered architecture model describes the relationship of the WS-Agreement Negotiation layer to the WS-Agreement layer and the service layer.

2.4.1 Negotiation Offer/Counter Offer Model

The WS-Agreement Negotiation Offer/Counter Offer model describes the dynamic exchange of information between the negotiation initiator and responder in order to agree on an acceptable agreement offer. A negotiation participator sends a negotiation offer to the other party, which in turn creates a counter offer for the negotiation offer received. Counter offers are always based on a negotiation offer that was previously received from the opposite negotiation party. The only exceptions are initial negotiation offers, which are based on a negotiation template. These initial offers can be regarded as counter offers to negotiation templates.

Each negotiation offer has an associated state, which reflects the view of the party that created that particular offer with respect to its acceptability. The possible state transitions that may occur when a counter offer is created for a particular offer are described in section 2.5.2.3.

An offer negotiation process may comprise multiple rounds of negotiation. In each negotiation round offers and counter offers are exchanged. The exchanged negotiation offers can therefore be modeled as a rooted tree with a negotiable template as root node. Each negotiation offer in this negotiation tree is a counter offer to its parent node. Children of the root node are initial negotiation offers, since they are based on a negotiable template. Leaf nodes are negotiation offers where either no further negotiation is required or that are in the terminal rejected state. If a negotiation offer does not require further negotiation it can be one of the following cases:

1. The negotiation offer is in the acceptable state and is used to create an agreement.
2. The negotiation participator does not follow this negotiation branch anymore, e.g. the participator decides that this negotiation branch does not lead to the expected results.

A negotiation process may include the exchange of negotiation offers that are based on different templates. A negotiation process can therefore comprise multiple negotiation trees. In the following example illustrates the concept of a negotiation tree in detail.

A negotiation initiator receives a negotiable template from the negotiation responder. Based on the negotiable template the initiator creates an initial negotiation offer with an offer id 1 (OID 1). This offer is then send to the negotiation responder using the responder’s negotiate method. After the negotiation responder received the initial negotiation offer (*OID 1*), it examines the incoming offer (*OID 1*) and creates two counter offers with *OID 2* and *OID 3*. These counter offers are returned to the negotiation initiator as result of the negotiate call. The negotiation initiator processes the returned counter offers and decides that both counter offers

do not lead to the desired agreement. The negotiation initiator therefore decides start a new negotiation branch by creating another negotiation offer (*OID 4*) based on the template *T1*. This offer is again send to the negotiation responder that decides that this particular offer is unacceptable. The responder therefore creates a counter offer (*OID 5*), which is in the rejected state. Finally, the negotiation initiator creates a third negotiation branch by generating another negotiation offer based on *T1*. After several rounds of negotiation the negotiation responder returns a counter offer (*OID 9*), which is in the acceptable state. This offer is subsequently used by the negotiation initiator to create a new agreement. This process is depicted in Figure 10.

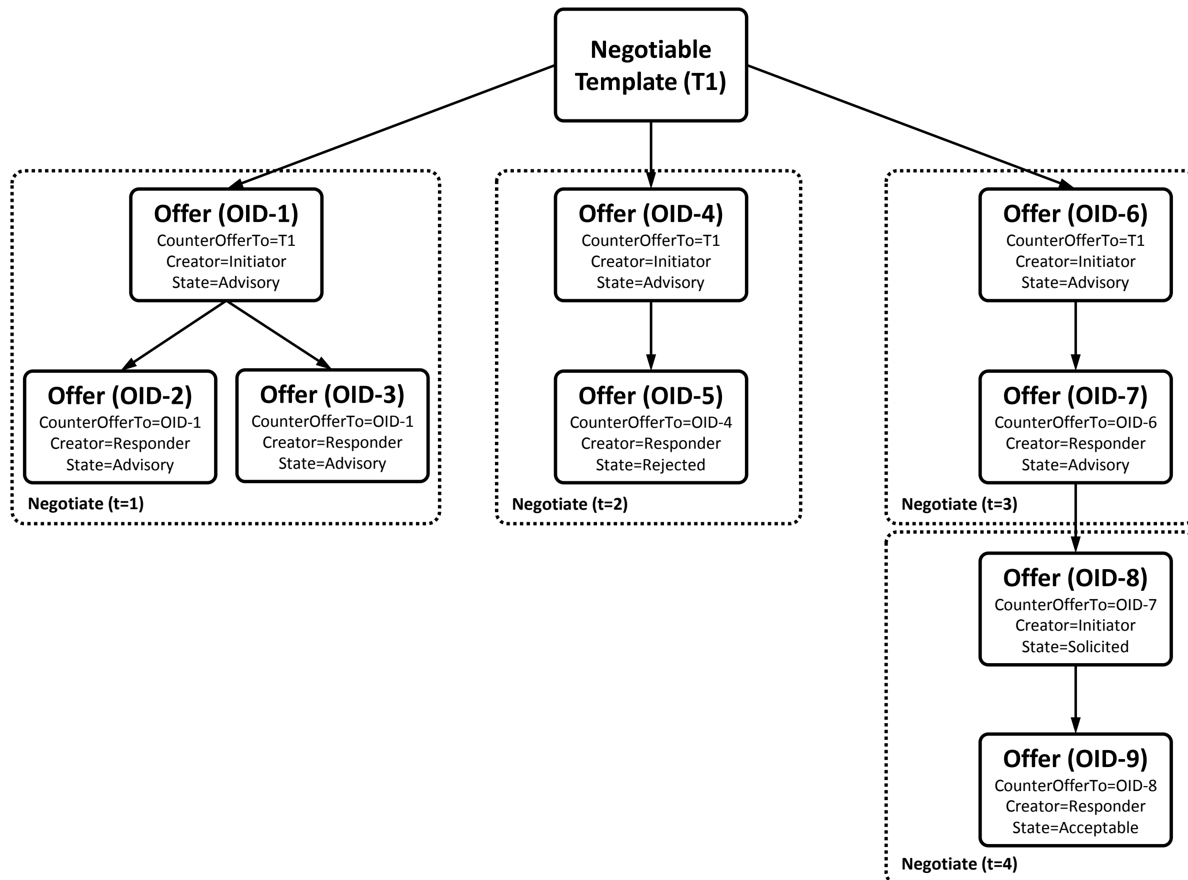


Figure 10: the exchange of multiple negotiation offers and counter offers results in the creation of a negotiation tree

The terms negotiation offer and negotiation counter offers both describe specific negotiation offers that are exchanged in a negotiation process. The distinction of what is a negotiation offer and what is a counter offer depends on the particular view of a negotiation participator. A negotiable template (the root node of a negotiation tree) is always considered as initial negotiation offer. All negotiation offers that are created based on this template are therefore counter offers to this template.

If a negotiation offer with *OID-1* was created based on a template *T1*, then *OID-1* is a counter offer to *T1*. If subsequently a negotiation offer *OID-2* is created based on offer *OID-1*, then OID-2 is a counter offer to *OID-1*. In case the negotiation responder provides the negotiable template *T1*, it provides an initial negotiation offer to the negotiation initiator. The initiator receives the template T1 and creates a counter offer with *OID-1* on the base of this template.

This counter offer is sent to the negotiation responder. From the negotiation responder's point of view, *OID-1* is a new negotiation offer from the negotiation initiator. The responder therefore creates a counter offer with *OID-2*. This process of creating counter offers based on previously received negotiation offers with the different viewpoints is depicted in Figure 11.

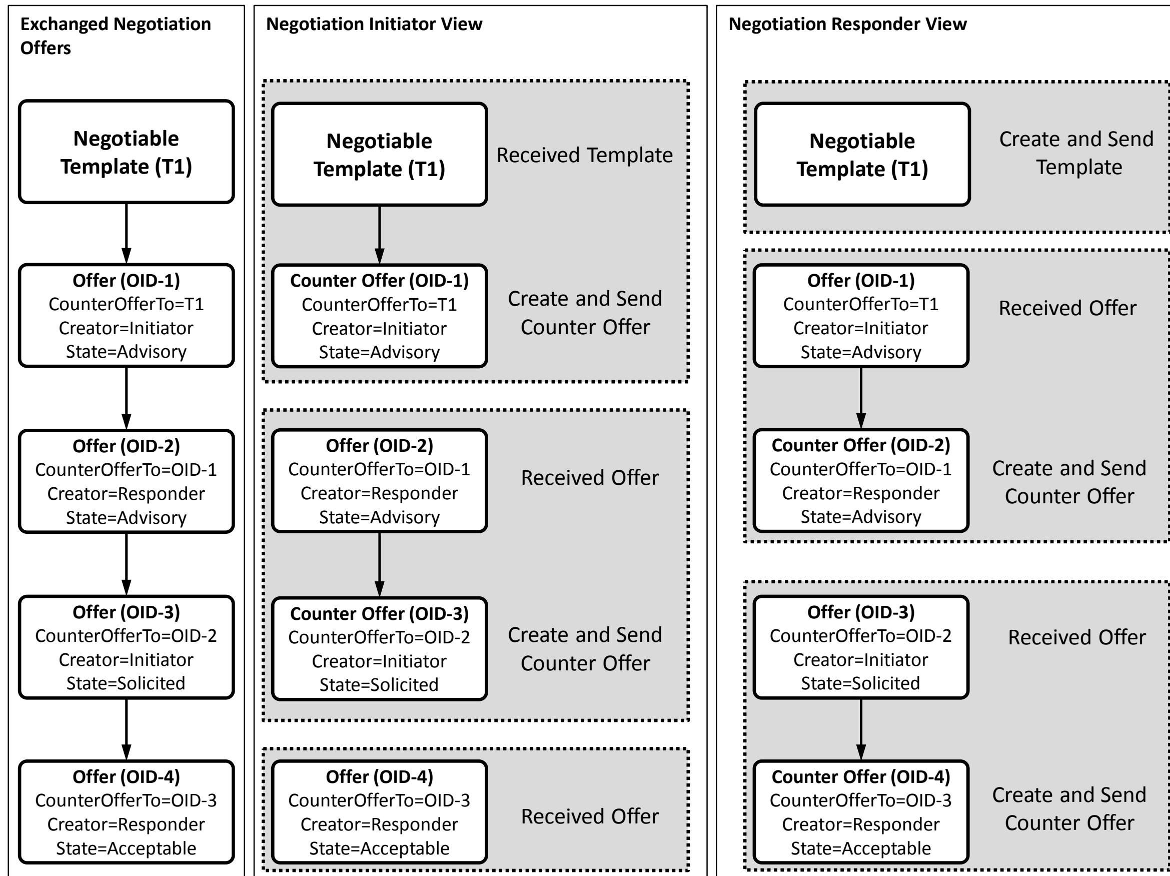


Figure 11: different views on the negotiation process - an offer send by one negotiation participator is a counter offer to a previously received negotiation offer

2.4.2 Layered Architectural Model

The WS-Agreement Negotiation layered model consists of three layers, the negotiation layer, the agreement layer and the service layer. These layers are depicted in Figure 12. There is a clear separation between these layers. The negotiation layer sits on top of the agreement layer. It is therefore decoupled from the agreement layer and the service layer. By that, the negotiation layer may change independently of the agreement layer and can be replaced by another negotiation layer that might be better suited for a specific negotiation scenario.

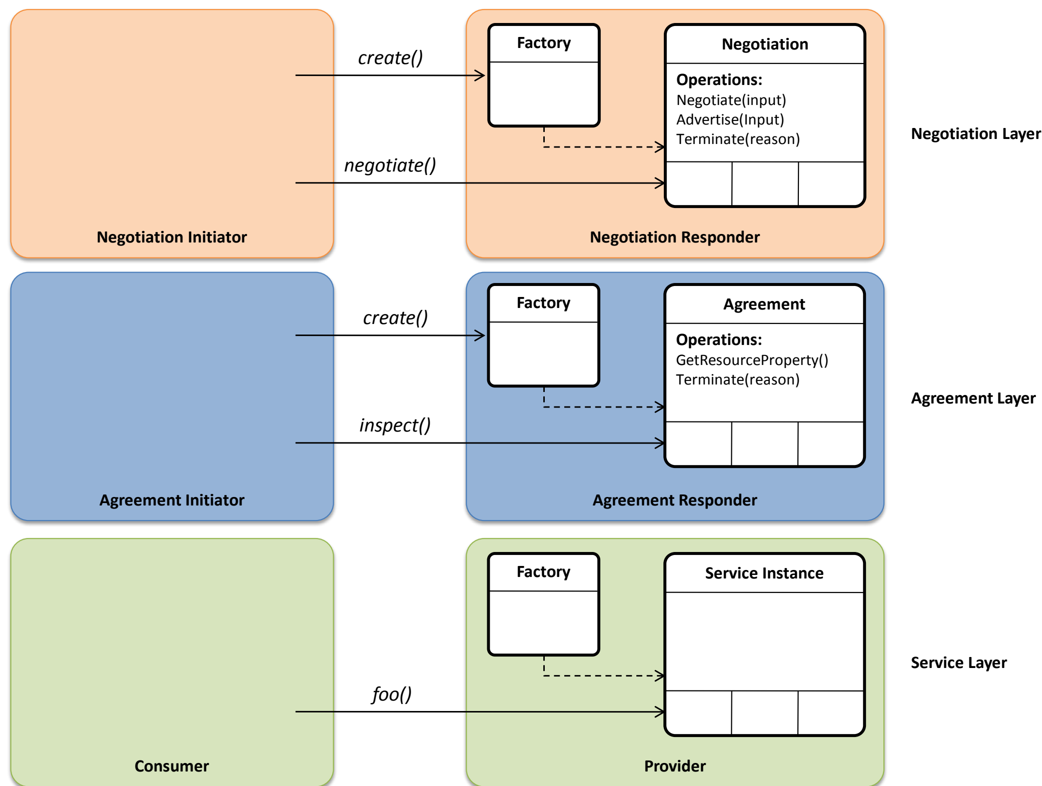


Figure 12: conceptual overview of the layered negotiation model

Negotiation layer

The negotiation layer provides a protocol and a language to negotiate agreement offers and counter offers and to create agreements based on negotiated offers. The negotiation process comprises the exchange of negotiation offers and counter offers. Negotiation offers, as defined in this specification, are non-binding by nature. They do not imply a promise of the agreement responder that it will create an agreement based on a negotiated offer. They only indicate the willingness of the two negotiating parties to subsequently create an agreement. However, it is possible to define languages that can be used in conjunction with this specification in order to realize binding negotiation processes.

Agreements based on negotiated offers are either created by calling the *createAgreement* or *createPendingAgreement* operation on the agreement responder's Agreement Factory port type, which is part of the responder's agreement layer.

Agreement layer

The Agreement layer provides the basic functionality to create and monitor agreements. It comprises the port types defined in the WS-Agreement specification. For further reference refer to the WS-Agreement specification [20].

Service layer

At the service layer the actual service defined by an agreement is provided. This service may or may not be a web service. Moreover, it may consist of multiple services. A resource provisioning service may for example comprise the provisioning of the specified resources

and a monitoring service for the provided resources. The services on the service layer are governed by the agreement layer.

2.5 Negotiation

The negotiation service is a service instance that is used by the negotiation participators to dynamically exchange information in order to reach a common understanding of a valid agreement offer. During the negotiation process the participators exchange negotiation offers in order to indicate their negotiation goals and requirements. A negotiation instance may be limited in its lifetime or the maximum negotiation rounds. These limitations are defined in the negotiation context.

2.5.1 Negotiation Context

The negotiation context defines the roles of the negotiation participators, their obligations, and the nature of the negotiation process. Since negotiation is a bi-lateral process, the roles of each participating party must be clearly defined.

```
<wsag-neg:NegotiationContext>
    <wsag-neg:NegotiationType>
        wsag-neg:NegotiationType
    </wsag-neg:NegotiationType>
    <wsag-neg:ExpirationTime>
        xsd:dateTime
    </wsag-neg:ExpirationTime> ?
    <wsag-neg:NegotiationInitiator>
        xsd:anyType
    </wsag-neg:NegotiationInitiator> ?
    <wsag-neg:NegotiationResponder>
          xsd:anyType
    </wsag-neg:NegotiationResponder> ?
    <wsag-neg:AgreementResponder>
          wsag-neg:NegotiationRoleType
    </wsag-neg:AgreementResponder>
```

```
    <wsag-neg:AgreementFactoryEPR>
          wsa:EndpointReferenceType
    </wsag-neg:AgreementFactoryEPR>
    <xsd:any /> *
</wsag-neg:NegotiationContext>
```

Listing 1: content of a negotiation context

A negotiation instance either refers to the negotiation of new agreements or to the renegotiation of an existing agreement. The type of the negotiation must therefore be defined in the negotiation context. Moreover, the negotiation context defines the roles of the parties participating in the negotiation. The negotiation participators must acknowledge these parameters for the entire negotiation process.

/wsag-neg:NegotiationContext

This is the outermost document tag that defines the context of a negotiation. The negotiation context defines the type of the negotiation and the roles of the negotiation participators.

/wsag-neg:NegotiationContext/wsag-neg:NegotiationType

This REQUIRED element specifies the type of the negotiation process and may contain optional, domain-specific parameters. The negotiation type can either be Negotiation or Renegotiation.

/wsag-neg:NegotiationContext/wsag-neg:ExpirationTime

This OPTIONAL element specifies the lifetime of the negotiation instance. If specified, the negotiation instance is accessible until the specified time. After the negotiation lifetime has expired, this instance is no longer accessible.

/wsag-neg:NegotiationContext/wsag-neg:NegotiationInitiator

This OPTIONAL element identifies the initiator of the negotiation process. The negotiation initiator element can be an URI or an Endpoint Reference that can be used to contact the initiator. It can also be a distinguished name identifying the initiator in a security context.

/wsag-neg:NegotiationContext/wsag-neg:NegotiationResponder

This OPTIONAL element identifies the party that responds to the initiateNegotiation request. The negotiation responder implements the NegotiationFactory port type defined in this specification. This element can be an URI or an Endpoint Reference that can be used to contact the negotiation responder. It can also be a distinguished name identifying the negotiation responder in a security context.

/wsag-neg:NegotiationContext/wsag-neg:AgreementResponder

This REQUIRED element identifies the party in the negotiation process that acts on behalf of the agreement responder. It can either take the value NegotiationInitiator or

NegotiationResponder. The default value is NegotiationResponder. The party identified as agreement responder MUST provide a reference to the AgreementFactory (PendingAgreementFactory) in the negotiation context within the AgreementFactoryEPR element.

/wsag-neg:NegotiationContext/wsag-neg:AgreementFactoryEPR

This REQUIRED element identifies the endpoint reference of the agreement factory that is used to create agreements based on the negotiated agreement offers. After an agreement offer was successfully negotiated, the party identified as agreement initiator MAY create a new agreement with the referenced agreement factory.

/wsag-neg:NegotiationContext/{any}

Additional child elements MAY be specified to provide additional information but MUST NOT contradict the semantics of the parent element; if an element is not recognized, it SHOULD be ignored.

2.5.1.1 Negotiation Type

The negotiation type defines the nature of a negotiation instance. Two types of negotiation exist; negotiation of a new agreements and re-negotiation of an existing agreement. The structure of the negotiation type is depicted in Listing 2.

```
<wsag-neg:NegotiationType>
    {
        <wsag-neg:Negotiation>
                <xsd:any /> *
        </wsag-neg:Negotiation>        |
        <wsag-neg:Renegotiation>
            <wsag-neg:ResponderAgreementEPR>
                wsa:EndpointReferenceType
            </wsag-neg:ResponderAgreementEPR>
            <wsag-neg:InitiatorAgreementEPR>
                wsa:EndpointReferenceType
            </wsag-neg:InitiatorAgreementEPR> ?
            <xsd:any /> *
        </wsag-neg:Renegotiation>
    }
</wsag-neg:NegotiationType>
```

Listing 2: structure and content of the negotiation type

/wsag-neg:NegotiationType

This is the outermost element that encapsulates the negotiation type. It MUST either contain a *Negotiation* or *Renegotiation* element.

/wsag-neg:NegotiationType/wsag-neg:Negotiation

The existence of this element indicates that the negotiation process comprises the negotiation of agreement offers.

/wsag-neg:NegotiationType/wsag-neg:Negotiation/{any}

Additional elements MAY be used to carry critical extensions which control additional negotiation mechanisms. All extensions are considered mandatory, i.e. the responder MUST return a fault if any extension is not understood or the responder is unwilling to support the extension. The meaning of extensions and how to obey them is domain-specific and MUST be understood from the extension content itself.

/wsag-neg:NegotiationType/wsag-neg:Renegotiation

The existence of this element indicates that the negotiation process comprises the renegotiation of an existing agreement. Renegotiation of existing agreements is again a bilateral process between an agreement initiator and an agreement responder. The wsag-neg:Renegotiation element MUST include an endpoint reference to the responder agreement that is renegotiated. In a symmetric layout of the agreement port types the wsag-neg:Renegotiation element MAY also contain an endpoint reference to the initiator agreement. Additionally, the wsag-neg:Renegotiation element MAY contain domain specific data that can be used to control the negotiation process in a domain-specific way.

/wsag-neg:NegotiationType/wsag-neg:Renegotiation/wsag-neg:ResponderAgreementEPR

This REQUIRED element identifies the agreement responder's copy of the agreement that is renegotiated. The service identified by this endpoint reference MUST implement the Agreement port type. Once a renegotiated agreement is created, this agreement instance must change its state to *Completed*.

/wsag-neg:NegotiationType/wsag-neg:Renegotiation/wsag-neg:InitiatorAgreementEPR

This OPTIONAL element identifies the agreement initiator's copy of the agreement that is renegotiated. In a symmetrical deployment of the agreement layer, the agreement initiator and responder host an instance of the agreement. If a renegotiated agreement is created, both agreement instances must change their state to *Completed*. The service identified by this endpoint reference MUST implement the Agreement port type.

/wsag-neg:NegotiationType/wsag-neg:Renegotiation/{any}

Additional elements MAY be used to carry critical extensions, which control augmented renegotiation mechanisms or creation mechanisms for renegotiated agreements. All extensions are considered mandatory, i.e. the agreement responder MUST return a fault if any

extension is not understood or the responder is unwilling to support this extension. The meaning of the extensions and how to obey them is domain-specific and MUST be understood from the extension content itself.

2.5.2 Negotiation Offer

As mentioned before, during a negotiation process information is dynamically exchanged in form of negotiation offers and counter offers. An initial negotiation offer is created on the basis of an agreement template, while counter offers are created on the basis of negotiation offers received by a negotiation participator. The structure of a negotiation offer is basically the same as the structure of an agreement. Agreements are defined in the section *Agreement Structure* of the WS-Agreement specification. However, negotiation offers contain additional elements, namely the *Negotiation Offer Context* and *Negotiation Constraints*.

2.5.2.1 Negotiation Offer Structure

When a negotiation participator receives a negotiation offer, it evaluates the offer and creates zero or more counter offers, which are then sent back to the party that issued the negotiation offer. The basic structure of a negotiation offer is shown in Figure 13.

Figure 13: structure of a negotiation offer

A negotiation offer has basically the same structure as an agreement, but it also contains a Negotiation Offer Id, a Negotiation Context, and a Negotiation Constraints section. It extends the *wsag:AgreementType* and therefore inherits the agreement name, agreement context, and the agreement terms.

Negotiation Constraints define restrictions for structure and values of negotiation counter offers. They must hold true for every counter offer. If this is not the case, the counter offer is rejected. Negotiation Constraints MAY change during the advance of a negotiation process. If, for example, the negotiation initiator chooses one specific service term out of a predefined set (e.g. in a ExactlyOne tag), the negotiation responder may adopt to this choice by changing the negotiation constrains in a counter offer.

Negotiation Constraints are structurally identical to Creation Constraints that are part of an agreement template. Creation Constraints are defined in the section *Agreement Template and Creation Constraints* of the WS-Agreement specification.

The contents of a negotiation offer are of the form:

```
<wsag-neg:NegotiationOffer wsag-neg:OfferId="xs:string">
    <wsag-neg:NegotiationOfferContext>
            wsag-neg:NegotiationOfferContextType
    </wsag-neg:NegotiationOfferContext>
    <wsag:Name>
            xs:string
    </wsag:Name> ?
    <wsag:Context>
            wsag:AgreementContextType
    </wsag:Context>
    <wsag:Terms>
            wsag:TermCompositorType
    </wsag:Terms>
    <wsag-neg:NegotiationConstraints>
            wsag:ConstraintSectionType
    </wsag-neg:NegotiationConstraints>
</wsag-neg:NegotiationOffer>
```

Listing 3: content of a negotiation offer

The following section describes the attributes and tags of a Negotiation Offer:

/wsag-neg:NegotiationOffer

This is the outermost document tag which encapsulates the entire negotiation offer.

/wsag-neg:NegotiationOffer/@wsag-neg:OfferId

The MANDATORY *OfferId* is the identifier of a specific Negotiation Offer. It MUST be unique for both parties in the context of a negotiation.

/wsag-neg:NegotiationOffer/wsag-neg:NegotiationOfferContext

The REQUIRED element Negotiation Offer Context contains the metadata associated with a negotiation offer. The negotiation offer context contains the id of the originating negotiation

offer, its expiration time, and its state. Moreover, the negotiation offer context MAY include domain specific extensions.

/wsag-neg:NegotiationOffer/wsag:Name

This is an OPTIONAL element is the name of the agreement to negotiate. It is described in the section "*Agreement Structure"* of the WS-Agreement specification.

/wsag-neg:NegotiationOffer/wsag:Context

This REQUIRED element of a negotiation offer specifies the context of the agreement to negotiate. The agreement context SHOULD include parties to an agreement. Additionally, it contains various metadata about the agreement such as the duration of the agreement, and optionally, the template name from which the agreement is created. The structure of the agreement context is described in the section *Agreement Context* of the WS-Agreement specification.

/wsag-neg:NegotiationOffer/wsag:Terms

This REQUIRED element specifies the terms of the agreement that is negotiated. Both the structure of and the values of the agreement terms can be subject of the negotiation process. The agreement terms are described in the WS-Agreement specification in the section *Agreement Structure*.

/wsag-neg:NegotiationOffer/wsag-neg:NegotiationConstraints

This REQUIRED element defines constraints on the structure and values that the agreement terms may take in subsequent negotiation offers. Negotiation Constraints with type *Required* MUST hold true for any counter offer. Negotiation Constraints with type *Optional* can be ignored by a negotiation participator. Negotiation constraints are of the type *wsag-neg:NegotiationConstraintSectionType*. A negotiation constraint section MAY contain zero or more negotiation item constraints and zero or more free form constraints.

/wsag-neg:NegotiationConstraints/wsag-neg:Item

This OPTIONAL element defines a negotiation item constraint. It extends the *wsag:Offer-ItemType* which is specified in the section *Creation Constraints* of the WS-Agreement specification. A negotiation item constraint additionally defines two attributes, *Type* and *Importance*.

/wsag-neg:NegotiationConstraints/wsag-neg:Item/wsag-neg:Type

This REQUIRED attribute defines the type of the negotiation item constraint. Valid values are *Required* and *Optional*. If a required negotiation item constraint is violated by a counter offer, this counter offer MUST be rejected. If an optional negotiation item constraint is violated by a counter offer, this item constraint MAY be ignored, depending on the domain specific negotiation strategy. The default value of this attribute is *Required*.

/wsag-neg:NegotiationConstraints/wsag-neg:Item/wsag-neg:Importance

This OPTIONAL attribute defines the importance of a negotiation item constraint. It is intended to be used in conjunction with optional negotiation item constraints. Implementation MAY use this attribute in order to specify the importance of different optional negotiation item constraints. It is therefore possible to implement negotiation strategies that minimize the overall utility of violated optional constraints.

/wsag-neg:NegotiationConstraints/wsag-neg:Constraint

This OPTIONAL element defines a free-form negotiation constraint analog to free-form constrains as specified in the WS-Agreement specification.

2.5.2.2 Negotiation Offer Context

The negotiation offer context contains the metadata of a negotiation offer. It refers to the originating negotiation offer, defines the offer expiration time, and the offer state. Additionally, it may contain domain specific elements in order to provide negotiation extensions, e.g. to realize binding negotiation offers and compensation methods.

```
<wsag-neg:NegotiationOfferContext>

    <wsag-neg:CounterOfferTo>

            xs:string

    </wsag-neg:CounterOfferTo>

    <wsag:ExpirationTime>

            xs:dateTime

    </wsag:ExpirationTime> ?

    <wsag:Creator>

            wsag-neg:NegotiationRoleType

    </wsag:Creator>

    <wsag-neg:State>

            wsag-neg:NegotiationOfferStateType

    </wsag-neg:State>

    <xsd:any /> *

</wsag-neg:NegotiationOfferContext>
```

Listing 4: content of a negotiation offer context

/wsag-neg:NegotiationOfferContext

This is the outermost tag that encapsulates the entire NegotiationOfferContext.

/wsag-neg:NegotiationOfferContext/wsag-neg:CounterOfferTo

The MANDATORY CounterOfferTo identifies the negotiation offer which was used to create this counter offer. When a negotiation offer was used to create this offer, the CounterOfferTo specifies the OfferId of the originating negotiation offer. When an agreement template was used to create this offer, the CounterOfferTo refers to the TemplateId of the originating template.

/wsag-neg:NegotiationOfferContext/wsag-neg:ExpirationTime

This REQUIRED element defines the lifetime of a negotiation offer. A negotiation participator MAY reference a negotiation offer during its lifetime and create counter offers for it.

/wsag-neg:NegotiationOfferContext/wsag-neg:Creator

This REQUIRED element identifies the party that created this negotiation offer. Valid values for this element are *NegotiationInitiator* and *NegotiationResponder*.

/wsag-neg:NegotiationOfferContext/wsag-neg:State

This REQUIRED element contains the state of a negotiation offer. The negotiation offer state indicates whether further negotiation is required. Negotiation offers must be in the ACCEPTABLE state in order to create an agreement based on it. Each negotiation offer state MAY contain domain specific extensions. E.g. if an offer was rejected for some reason, the REJECTED state may contain information on why this offer was rejected. This information can be used to optimize the negotiation process.

/wsag-neg:NegotiationOfferContext/{any}

Additional child elements MAY be specified to provide additional information, but the semantic of these elements MUST NOT contradict the semantics of the parent element; if an element is not recognized, it SHOULD be ignored.

2.5.2.3 Negotiation Offer States

The negotiation of an agreement offer precedes the final agreement creation process. The party that is defined as agreement initiator in the negotiation context is responsible of creating the agreement. A valid negotiated agreement offer SHOULD be in the ACCEPTABLE state when the agreement is created. Figure 14 shows the possible states of negotiation offers along with valid state transitions.

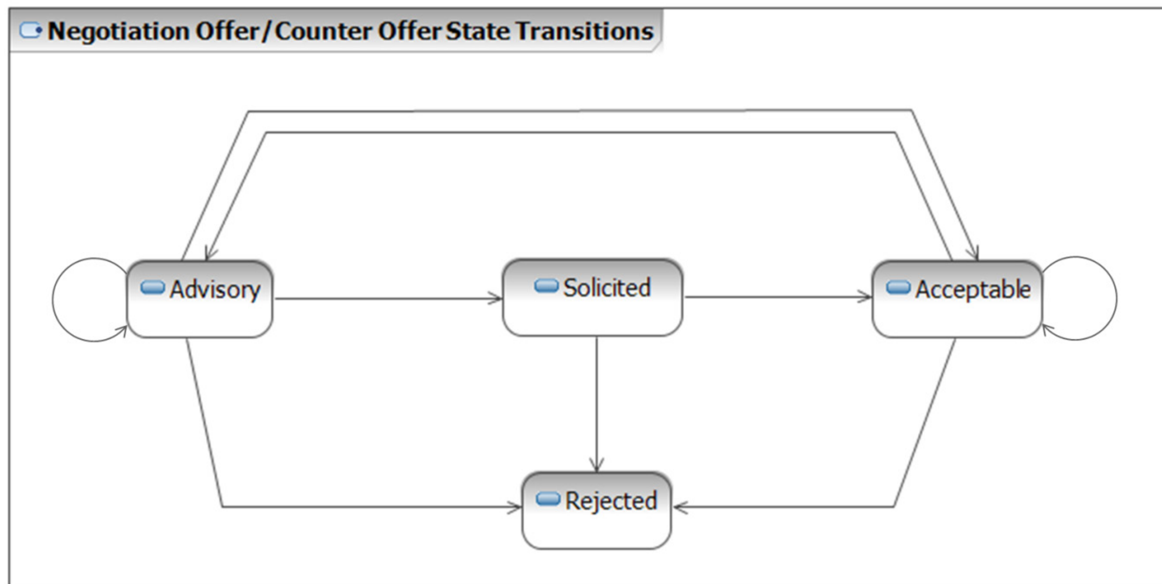


Figure 14: the state machine describes the states of a counter offer in relationship to the state of the offer it refers to

Advisory State

The ADVISORY state identifies negotiation offers which have no further obligations associated. Offers in the ADVISORY state usually contain elements that are currently not specified. Therefore, these offers require further negotiation.

Solicited State

Solicited offers indicate that a negotiation participator wants to converge the negotiation process. The SOLICITED state bears no obligations for an offer, but it requires that counter offers are either in the ACCEPTABLE or the REJECTED state.

Acceptable State

The ACCEPTABLE state indicates that a negotiation participator is willing to accept a negotiation offer as is. All details of a negotiation offer are specified and no further negotiation is required. However, since the negotiated offers are non-binding, there is no guarantee that a subsequent agreement is created. Augmented negotiation protocols may be created based on this specification to address binding negotiations.

Rejected State

If a negotiation offer is rejected, a counter offer is sent back to the inquiring party with the REJECTED state. All terms SHOULD be the same as in the original offer the counter offer refers to. The counter offer MAY contain a domain specific reason why it was rejected. Negotiation offers that are marked as rejected MUST NOT be used to create an agreement. However, they MAY be used to continue the negotiation process by taking into account the reason for rejecting the offer.

Extension of Negotiation Offer States

Each state element MAY contain additional child elements to provide domain specific information. This information can be used to optimize the negotiation process. If this information is not understood, it SHOULD be ignored.

2.5.2.4 Negotiation Offer State Transitions

The state model abstractly describes the possible state transitions that can occur when a counter offer is created for a negotiation offer. This means that the state of each child node in a negotiation tree must be a valid state transition with respect to its parent node's state. Since negotiation offers and counter offers are exchanged between the negation participators over time, this section describes how exactly the state model maps to the exchanged negotiation offers.

The negotiation model allows negotiating multiple negotiation offers at one time. A negotiation initiator may for example create three negotiation offers (*OID-1, OID-2, OID-3*) based on a negotiable template *T1*. In a first negotiation iteration (t=1) these negotiation offers are sent to the negotiation responder in a single negotiate request. The responder creates counter offers for each of the received offers. For the negotiation offer *OID-1* the responder creates two counter offers (*OID-4, OID-5*) which are in the advisory state. The negotiation offer *OID-2* is rejected. The negotiation responder therefore creates a counter offer (*OID-6*) which is in the rejected state. For the negotiation offer *OID-3*, the responder creates one counter offer (*OID-7*) which again is in the advisory state. All states of the counter offers are valid state transitions regarding to the states of the offers they are based on. The counter offers are returned to the negotiation initiator as result of the negotiate call. The negotiation initiator analyses the received counter offers and decides to continue the negotiation process based on the offer *OID-7*. It therefore creates a new negotiation offer (*OID-8*). This time the negotiation offers is in the solicited state, which requires that counter offers are either in the acceptable state or in the rejected state. Negotiation offer OID-8 is then sends in a second negotiation iteration to the negotiation responder, again using the negotiate method. This time the responder decides to accept negotiation offer *OID-8*. It therefore creates a counter offer (*OID-9*) which is in the acceptable state. This process is depicted in Figure 15.

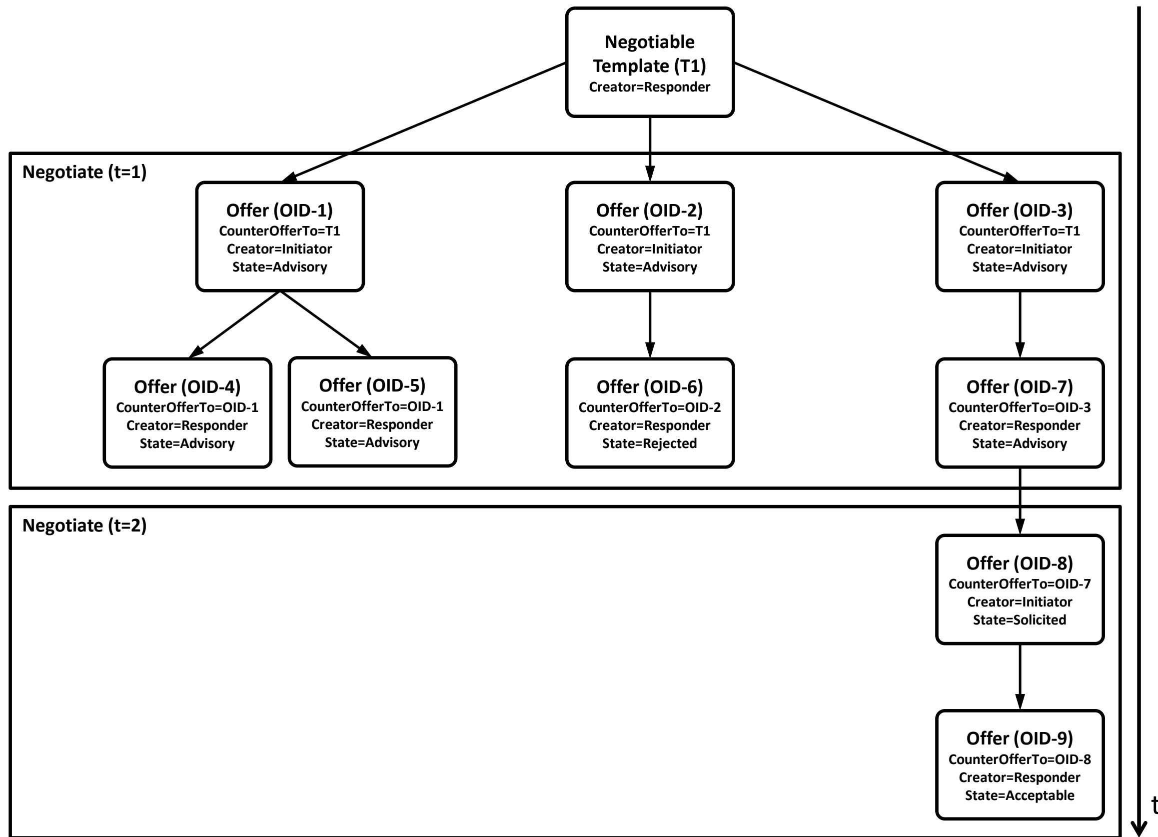


Figure 15: state transitions for parallel negotiation of multiple offers

In a second scenario, the negotiation initiator again creates a negotiation offer *OID-1* based on template *T1*. It sends the negotiation offer in the first negotiation iteration to the responder. The negotiation responder creates two counter offers (*OID-2, OID-3*) for *OID-1* and returns them. The initiator then decides to follow the negotiation process based on offer *OID-3*. It creates a negotiation offer (*OID-4*) and sends it to the responder in the second negotiation iteration. The responder analyses the offer and decides to reject it. It creates a counter offer (*OID-5*) and returns it as result of the second negotiation iteration. Negotiation offer *OID-5* additionally contains a domain specific rejection reason. The negotiation initiator MAY use this information to create a new negotiation offer (*OID-6*), taking the rejection reason into account. The offer *OID-6* MUST NOT be based on the rejected offer *OID-5,* since the negotiation responder already indicated that it is not willing to follow this negotiation branch anymore. Instead *OID-6* is a counter offer to *OID-3*, which is the parent of the rejected offer *OID-4*. The negotiation offer OID-6 is sent in a last iteration to the negotiation responder that finally decides to accept it. This process is illustrated in Figure 16.

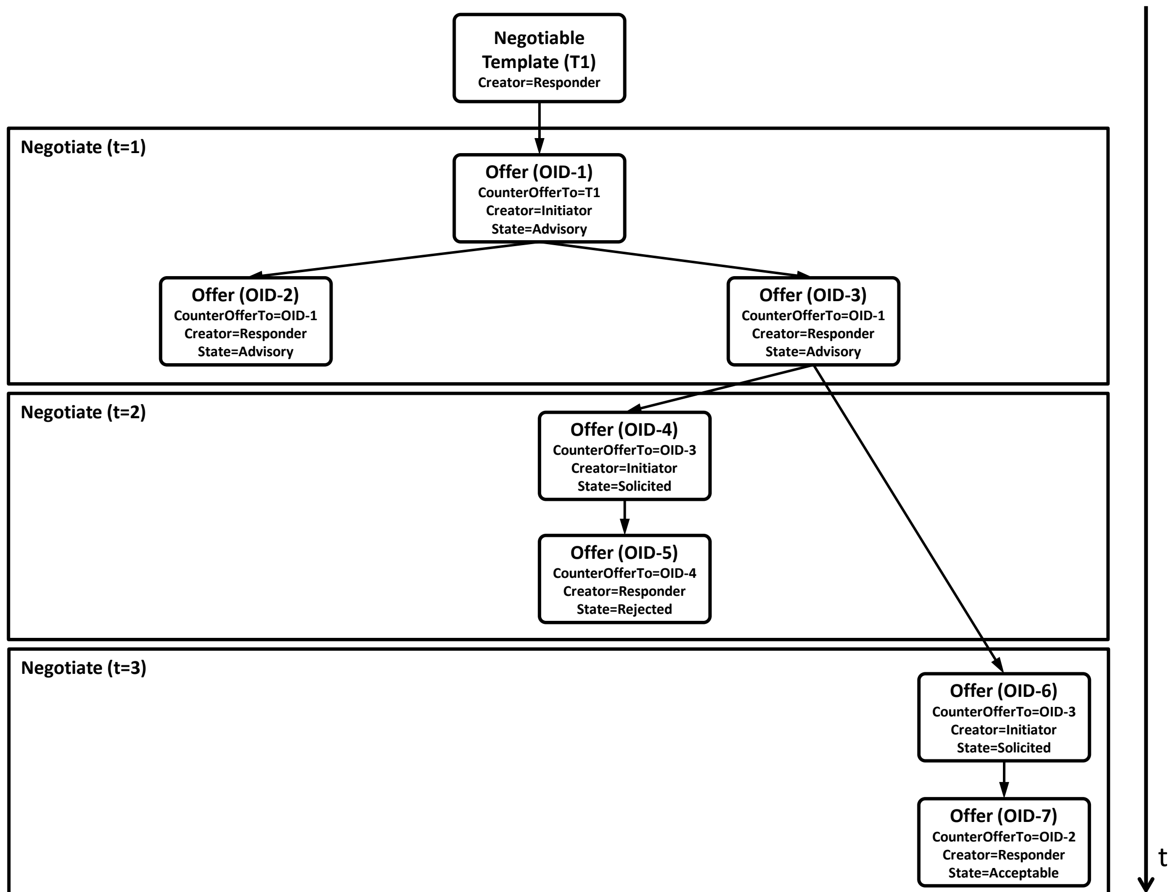


Figure 16: creation of counter offers taking rejection reasons into account

2.6 Creation of Negotiated and Renegotiated Agreements

Negotiation Offers extend the wsag:AgreementType. They can therefore easily be converted into agreement offers. These agreement offers are then used on the agreement layer to create new agreements. Since in a non-binding negotiation scenarios negotiated offers do not bear any obligations for either negotiating party, the creation of agreements based on such a negotiated offer is in principle independent of the negotiation process. The negotiation layer and the agreement layer are therefore completely decoupled and there is no need for additional extensions or control mechanisms for creating new agreements based on negotiated offers. Nevertheless, it is still possible to design augmented negotiation protocols that tightly couple to the negotiation layer and the agreement layer by using the provided extension points.

While this is also true for renegotiated agreements, additional information is required when a renegotiated agreement is created. This information is stored in a Renegotiation Extension document and is passed to the createAgreement (createPendingAgreement) method of an Agreement Factory (PendingAgreementFactory) as Critical Extension. The Renegotiation Extension document contains the endpoint reference of the original agreement that is renegotiated and possibly domain specific extensions. The structure of a Renegotiation Extension document is shown in Listing 6. In case a renegotiated agreement is successfully created, the state of the original agreement(s) MUST change to *Complete*.

2.6.1 Negotiation Extension Document

A negotiation extension document SHOULD be passed to the createAgreement (createPendingAgreement) method of an AgreementFactory (PendingAgreementFactory) when an agreement is created on the base of a negotiated offer. The negotiation extension document SHOULD be passed as critical extension. The following describes the content of a negotiation extension document:

```
<wsag-neg:NegotiationExtension>
    <wsag-neg:ResponderNegotiationEPR>
            wsa:EndpointReferenceType
    </wsag-neg:ResponderNegotiationEPR> ?
    <wsag-neg:InitiatorNegotiationEPR>
            wsa:EndpointReferenceType
    </wsag-neg:InitiatorNegotiationEPR> ?
    <wsag-neg:NegotiationOfferContext>
            wsag-neg:NegotiationOfferContextType
    </wsag-neg:NegotiationOfferContext>
    <xsd:any /> *
</wsag-neg:NegotiationExtension>
```

Listing 5: negotiation extension document to create agreements based on negotiated offers

/wsag-neg:NegotiationExtension

This is the outermost element of a negotiation extension document. This document SHOULD be passed to an agreement factory (pending agreement factory) as a critical extension in the *createAgreement* (*createPendingAgreement*) method.

/wsag-neg:NegotiationExtension/wsag-neg:ResponderNegotiationEPR

This OPTIONAL element specifies the endpoint reference to the negotiation responder's negotiation instance. Implementations MAY use this reference to identify the negotiation process in which an agreement offer was negotiated.

/wsag-neg:NegotiationExtension/wsag-neg:InitiatorNegotiationEPR

This OPTIONAL element specifies the endpoint reference to the negotiation initiator's negotiation instance. Implementations MAY use this reference to identify the negotiation process in which an agreement offer was negotiated.

/wsag-neg:NegotiationExtension/wsag-neg:NegotiationOfferContext

This REQUIRED element specifies the negotiation offer context for this agreement offer. It MUST refer to a valid negotiation offer where this agreement offer is a counter offer to.

/wsag-neg:NegotiationExtension/{any}

This OPTIONAL element contains domain specific extensions that can be used to realize augmented negotiation mechanisms.

2.6.2 Renegotiation Extension Document

The renegotiation extension document MUST be passed to the createAgreement (createPendingAgreement) method of an AgreementFactory (PendingAgreementFactory) as a critical extension when a renegotiated agreement is created. The following describes the content of a renegotiation extension document:

```
<wsag-neg:RenegotiationExtension>
    <wsag-neg:ResponderAgreementEPR>
            wsa:EndpointReferenceType
    </wsag-neg:ResponderAgreementEPR>
    <wsag-neg:InitiatorAgreementEPR>
            wsa:EndpointReferenceType
    </wsag-neg:InitiatorAgreementEPR> ?
    <wsag-neg:ResponderNegotiationEPR>
            wsa:EndpointReferenceType
    </wsag-neg:ResponderNegotiationEPR>
    <wsag-neg:InitiatorNegotiationEPR>
            wsa:EndpointReferenceType
    </wsag-neg:InitiatorNegotiationEPR> ?
    <wsag-neg:NegotiationOfferContext>
            wsag-neg:NegotiationOfferContextType
    </wsag-neg:NegotiationOfferContext>
    <xsd:any /> *
</wsag-neg:RenegotiationExtension>
```

Listing 6: critical extensions to create a renegotiated agreement

/wsag-neg:RenegotiationExtension

This is the outermost element of a Renegotiation Extension document. This document is passed to an agreement factory (pending agreement factory) as a critical extension in a *create-Agreement* call (*createPendingAgreement* call). An agreement factory (pending agreement factory) MUST be able to understand all critical extensions that are contained in a *create-*

Agreement call (*createPendingAgreement* call). If this is not the case, the factory MUST return an error.

/wsag-neg:RenegotiationExtension/wsag-neg:ResponderAgreementEPR

This REQUIRED element specifies the endpoint reference to the original instance of the responder agreement. If an *Agreement Responder* decides to accept an offer for a renegotiated agreement, the state of this agreement MUST change to *Completed*.

/wsag-neg:RenegotiationExtension/wsag-neg:InitiatorAgreementEPR

This OPTIONAL element specifies the endpoint reference to the original instance of the initiator agreement. This element is used in symmetric layouts of the agreement port type. If an *Agreement Responder* decides to accept an offer for a renegotiated agreement, the state of this agreement instance MUST change to *Completed*.

/wsag-neg:RenegotiationExtension/wsag-neg:ResponderNegotiationEPR

This REQUIRED element specifies the endpoint reference to the negotiation responder's negotiation instance. Implementations use this reference to identify the negotiation process in which an agreement offer was negotiated.

/wsag-neg:RenegotiationExtension/wsag-neg:InitiatorNegotiationEPR

This OPTIONAL element specifies the endpoint reference to the negotiation initiator's negotiation instance. Implementations use this reference to identify the negotiation process in which an agreement offer was negotiated.

/wsag-neg:NegotiationExtension/wsag-neg:NegotiationOfferContext

This REQUIRED element specifies the negotiation offer context for this agreement offer. It MUST refer to a valid negotiation offer where this agreement offer is a counter offer to.

/wsag-neg:RenegotiationExtension/{any}

This OPTIONAL element contains domain specific extensions that can be used to realize augmented renegotiation mechanisms.

2.7 Deployments of the Negotiation Port Types

This section describes possible deployments of the Negotiation Factory and the Negotiation port types in detail. These port types can be used in different combinations to support a wide range of signaling scenarios. The examples are not meant to cover all possible combinations of the port types. They illustrate possible signaling scenarios and show how these scenarios are mapped to specific deployments of WS-Agreement Negotiation port types. Furthermore, the interaction of the negotiation layer and the agreement layer is discussed. The detailed specification of the WS-Agreement Negotiation port types can be found in the Appendix.

2.7.1 Simple Client-Server Negotiation

The simple client-server negotiation represents an asymmetric signaling scenario. The server domain implements the Negotiation Factory, Negotiation, Agreement Factory, and Agreement port types. The negotiation process is driven by the client. In the first step the client initiates a new negotiation process by calling the server's *initiateNegotiation* operation. The server returns an endpoint reference to a new negotiation instance. The client uses this EPR for the subsequent negotiation process. In the next step client the queries the negotiable templates from the new created negotiation instance and selects the template it wants to negotiate an SLA for. Moreover, the client creates an initial negotiation offer based on the selected template. This offer is then sent to the negotiation instance by calling the server's *Negotiate* method. The server creates one or more counter offers for the received negotiation offer and sends them back to the client. The client chooses the counter offer that fulfills its requirements best and creates a new agreement with the server by calling its *createAgreement* method. The client sends a NegotiationExtensionDocument along with the createAgreement-request in order to identify the originating negotiation instance and the negotiation offer that resulted in this agreement offer.

In this scenario, the server has a passive role. It is not in control of the negotiation process, i.e. it only reacts to negotiation requests. The negotiation process is depicted in Figure 17.

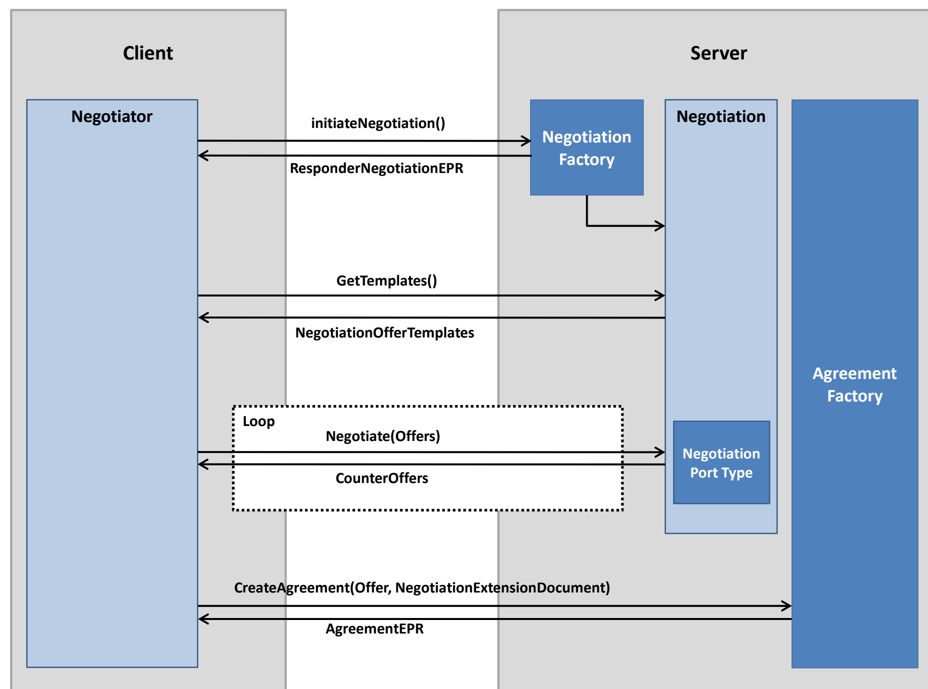


Figure 17: asymmetric deployment of the WS-Negotiation port types

2.7.2 Bilateral Negotiation with Asymmetric Agreement Layer

In a bilateral negotiation both parties actively participate in the negotiation process. For that reason both parties implement the WS-Agreement *NegotiationFactory* and *Negotiation* port types. A bilateral negotiation process is initiated as follows. The negotiation initiator creates a

new negotiation instance. This instance is a web service resource that implements the WS-Agreement Negotiation port type. The negotiation initiator then invokes the *initiate-Negotiation* method of the negotiation responder. The *initiateNegotiation* request includes an endpoint reference to the negotiation instance created beforehand. Moreover, it contains the negotiation context that defines the roles of each party participating in the negotiation process. The negotiation context defines for example which party acts as agreement initiator and which party acts as agreement responder. Once the negotiation instance is created, the negotiation context is fixed and the roles and responsibilities of the negotiation participators do not change anymore.

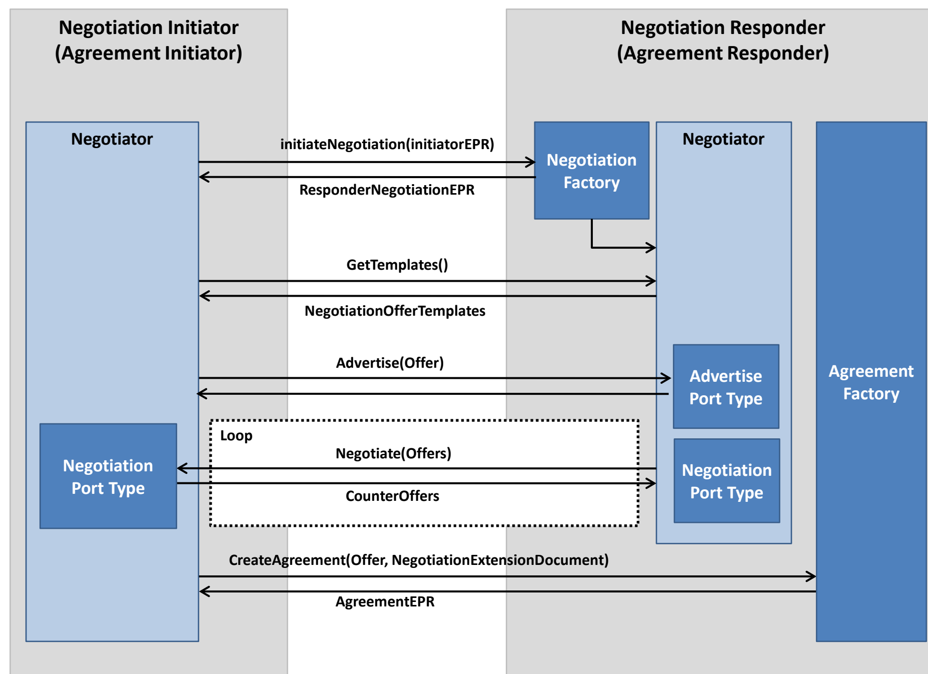


Figure 18: Symmetric deployment of WS-Agreement Negotiation port types, where the negotiation initiator is also the agreement initiator and the negotiation responder is the agreement responder. Both parties have an active role in the negotiation process.

The negotiation scenario depicted in Figure 18 shows an example of a bi-lateral negotiation. In this scenario the negotiation initiator is also the agreement initiator. The negotiation initiator starts the negotiation by initiating a new negotiation process with the responder. Next the initiator queries the negotiable templates from the negotiation responder and creates an initial negotiation offer based on the template it wants to create a SLA for. The initiator then notifies the responder about the initial negotiation offer. This is done by sending the offer to the responder by invoking its *Advertise* method. The negotiation responder now takes an active role in the negotiation process. It creates counter offers for the received negotiation offer and sends them to the initiator by invoking its negotiate method. After several rounds of negotiation the agreement initiator decides to create an agreement based on one of the negotiated offers. It therefore calls the *createAgreement* method of the responder, passing the negotiated agreement offer along with a NegotiationExtensionDocument. The Negotiation-ExtensionDocument is passed as a critical extension. It refers to the negotiation instance that was used to negotiate the agreement offer and contains a reference to the originating negotiation offer.

2.7.3 Re-Negotiation of Existing Agreements

Renegotiation of existing agreements applies the same signaling pattern as negotiation of agreements. If the original agreement initiator matches the initiator of the renegotiated agreement, the roles and obligations of the original agreement also match the roles and obligations of the renegotiated agreement. If the agreement initiator and responder roles are changed, the roles and obligations in the renegotiated agreement must be adopted accordingly. As mentioned before, the roles and the responsibilities of the negotiating parties are specified in the negotiation context as soon a new negotiation is initiated. In a renegotiation process, the negotiation context must also refer to the agreement to renegotiate. It MUST therefore contain an endpoint reference to the original responder agreement instance. In a symmetric deployment of the agreement port type, the negotiation context SHOULD also include a reference to the original initiator agreement. After the initialization of the renegotiation process, both parties negotiate an acceptable agreement offer. In case they succeed negotiating such an offer, the party defined as agreement initiator invokes the *createAgreement* (*createPendingAgreement*) method of the responder. When the renegotiated agreement is created successfully, the original agreements MUST change their states to *Completed*.

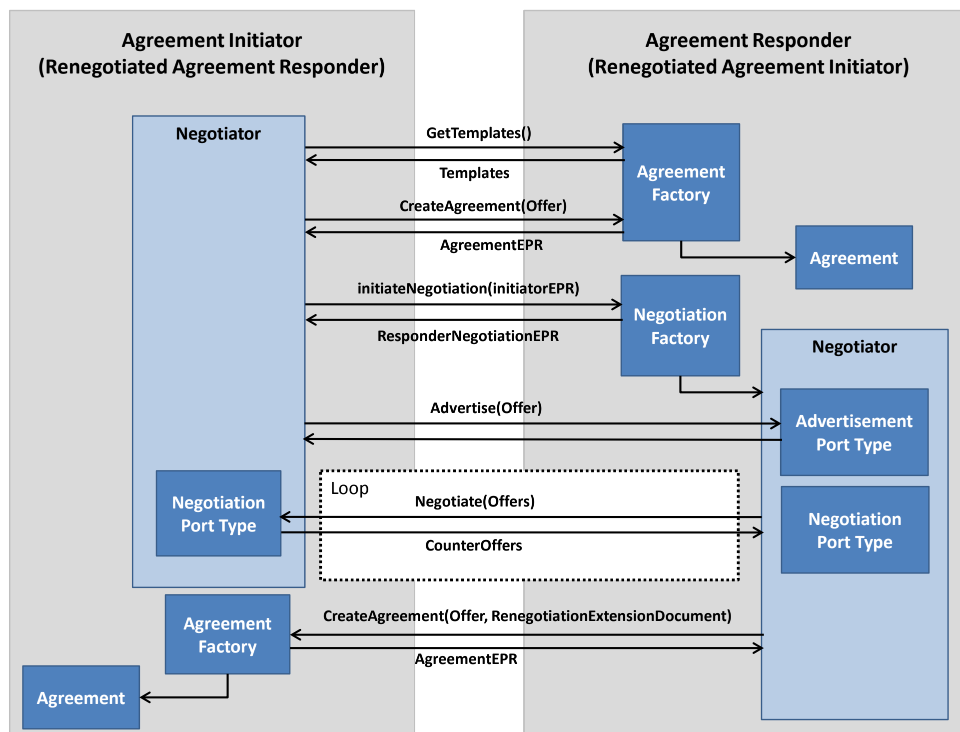


Figure 19: Symmetric signaling on the negotiation and agreement layer. Both parties implement the WS-Agreement Negotiation and WS-Agreement port types. Here, the roles of agreement initiator and responder change for the renegotiated agreement. The responder of the original agreement becomes the initiator of the renegotiated agreement.

As mentioned before, the layout of the agreement layer may either be symmetric or asymmetric. A detailed description of symmetric deployments of the agreement port type is given in the section *Port Types and Operations* of the WS-Agreement specification [20]. Figure 19 shows a symmetric deployment of the negotiation and agreement port types. In this scenario, the initiator of the original agreement becomes the agreement responder for the

renegotiated agreement. The roles of the agreement initiator and responder therefore change in the renegotiated agreement and must be adopted accordingly.

2.8 Summary

In this chapter, the WS-Agreement Negotiation protocol was presented as an approach to negotiate and renegotiate service level agreements. The negotiation protocol is intended to be used in conjunction with WS-Agreement. Existing implementations can seamlessly be extended with negotiation capabilities by using this approach. Due to its flexibility, WS-Agreement Negotiation can be used in a wide set of negotiation scenarios, for example in simple client-server negotiations or in peer-to-peer scenarios. A number of deployment scenarios were described in order to illustrate how the negotiation protocol is used in these scenarios. Moreover, the protocol supports the definition of negotiation constraints. Negotiation participators use negotiation constraints to express requirements on counter offers and may implement validation strategies to assert the adherence to these constraints.

Chapter 3
ARCHITECTURE OF THE SLA FRAMEWORK

Today, service level agreements are an accepted concept in distributed systems such as the Grid. They provide the means to offer services with a defined quality, to monitor the conformance of the agreed service quality, and to define and enforce compensation methods for meeting or violating the agreed service quality. The following examples illustrate how service level agreements can be applied distributed systems.

- A Grid service provider defines an SLA in order to provide its cluster resources to its customers with a defined service level, e.g. with a maximum recovery time in case of a resource failure. Therefore, the service provider has installed a set of monitoring and failover processes to identify resource problems early and react accordingly.

- Another Grid service provider offers a computational service to its customers for running a simulation. The service can be used with two different service levels: a basic service level with a minimal set of computational resources, and a professional service level where more or faster computing resources are allocated for the service. At the professional level, the service provider also checkpoints the application. If a resource fails, the application is migrated to another resource and restarted using the latest checkpoint.

- A third Grid service provider offers a web based application to its customers. The service provider therefore defines a SLA that guarantees the average and maximum application response time for up to 100, up to 1.000, or up to 10.000 users. A consumer can create a SLA with the service provider indicating the required service level. The service provider can dynamically allocate or de-allocate resources for the web application depending on the current load of the system.

WS-Agreement is a standard for creating and monitoring service level agreements in distributed environments. Since it is designed to be domain independent, it can be used in a wide range of application scenarios and is therefore a perfect candidate for realizing generic SLA management systems. The WSAG4J framework [5] presented in this chapter is such a generic SLA management system for the Java programming language [34]. The management of service level agreements based on WS-Agreement comprises a number of reoccurring tasks, which are implemented and automated by the WSAG4J system to a different degree. Domain independent tasks such as template publishing, offer validation, agreement creation and monitoring, and guarantee evaluation are completely automated by the framework. For domain dependent tasks such as service instantiation and monitoring a programming model is provided in order to add this functionality to the WSGA4J system. The framework can be used to realize SLA aware service provisioning systems for specific application domains and it provides a full implementation of the WS-Agreement protocol and language. It has been successfully applied in a wide set of application scenarios, ranging from SLA aware provisioning of computational, network, and license resources up to SLA aware provisioning of scientific applications.

This chapter describes the architecture and capabilities of the WSAG4J framework in detail, and depicts basic concepts of the framework. It also shows how the different capabilities are implemented. In the beginning of this chapter, an overview of the WSAG4J architecture is given and the basic framework components, their responsibilities and interactions are described. This comprises a description of how the WSAG4J engine is integrated with the web service layer. Next, the agreement factory component is described. In the WSAG4J engine this is the responsible component for creating new agreements. It validates the consistency of agreement offers and instantiates the services for a new agreement. In WSAG4J new agreements are created based on templates. These templates can contain a set of rules that describe how a valid agreement offer is structured and which values are valid for it. The agreement offer validation is a domain independent process that ensures that an offer is created according to these rules. This process helps to protect the agreement responder from accepting agreement offers that are created in an illegal way. This is of particular significance for application scenarios where service consumers are charged for a service usage. The agreement offer validation may become a very complex process, which is completely automated by the WSAG4J framework. The framework therefore significantly decreases the complexity of building WS-Agreement based systems. This chapter describes the offer validation process in detail, starting from the definition of agreement creation constraints, over the validation model, up to the implementation in the framework. In contrast to the offer validation, the instantiation of services belonging to an agreement is a domain specific process. WSAG4J defines a programming model in order to add agreement specific service instantiation strategies to the system. This programming model is also presented in detail. The last part of this chapter describes the agreement monitoring process. Agreement monitoring comprises the monitoring of the services associated with an agreement. The service monitoring produces the required data to assess the quality of the service provided. Based on these monitoring results the guarantees of an agreement are evaluated and the resulting penalties and rewards are accounted. This chapter describes in detail how agreement guarantees are defined in the WSAG4J framework and how they are evaluated and accounted later on.

3.1 Related Work

A number of WS-Agreement based implementations already exist today. [35] and [22] provide an overview of the most important implementations. One of the first adopters of WS-Agreement was the Cremona framework of IBM, which is a software architecture and library for creating and monitoring SLAs based on WS-Agreement. Cremona was first described in [36] and was published as part of the IBM Emerging Technology Toolkit (ETTK) implementation for Web Services.

AgentScape [37], a software platform for distributed agents, adopted WS-Agreement for resource negotiation. The resource negotiation model enables agents to dynamically acquire computational resources for execution [38]. Resources are provided to agents by autonomous entities called hosts. Hosts are structured in virtual domains, which are in turn controlled by a domain coordinator. Agents can negotiate with the coordinators from multiple domains and finally create an agreement with the domain coordinator that makes the best offer.

The European AssessGrid [39] project introduces risk-awareness to Grid actors, such as end-users, brokers and resource providers [40]. The probability of failure is thereby an important concept in the resource provisioning process. SLA-aware resource providers offer their services with a defined probability of failure. The reliability of such a statement is another aspect addressed by the project. The AssessGrid Negotiation Manager implements the components of the architecture based on the Globus Toolkit 4.x [3]. It comprises a resource provider component and a broker component. The resource provider supports the negotiation of general SLA parameters like number of nodes, memory, deadline for job completion, and guaranteed probability of failure, as well as price negotiation. Moreover, in case of a resource failure it is capable to acquire resources from another resource provider and to migrate the failed job using a checkpoint-restart mechanism. The broker component supports meta-scheduling of workflows and time, cost, and risk optimized assignment of SLA.

The CATNETS project investigates decentralized resource allocation mechanisms in computational networks [41]. Software agents buy and sell services and resources based on the Catalaxy paradigm. The agents interact with the service and resource providers at the corresponding service or resource markets. The contractual relationships are expressed via WS-Agreement. CATNETS defines a bidding language and a protocol that are used by the agents to bid for services or resources. Resource and service auctions finally result in an agreement. The CATNETS WS-Agreement implementation is integrated into the Triana workflow engine in order to visualize Agreement Templates and Offers in a negotiation process.

WS-Agreement was further used in a variety of other research projects. The BREIN project [42] uses SLAs in order to support reliable Grids for businesses. The goal of the BEinGRID project [43] was to foster the acceptance of Grid technologies for business. It included a number of business experiments, which required support for SLA negotiation, evaluation, and accounting. BEinGRID therefore developed a WS-Agreement-based SLA layer. The Job Submission Service (JSS) [44] is a resource broker that helps users to minimize the turn-around time of a job. JSS uses WS-Agreement for the advance reservation of computational resources [45].

In the following we present an approach to a generic WS-Agreement based SLA management framework. This framework is unique in the sense that it is a stand-alone WS-Agreement based SLA management system which is independent of a particular resource provisioning system. It provides a number of generic, domain-independent SLA management mechanisms that can easily adapted to SLA aware service provisioning systems.

3.2 Overview of the WSAG4J Framework Architecture

The WSAG4J framework is designed as a generic SLA layer that can be used in a wide set of scenarios. Therefore, it needs to be as extensible and modular as possible. Extensibility and modularity are key concepts, which have influenced the framework design from the very beginning. The framework comprises the following modules:

- *The API Module*
  This module contains interface definitions and implementations that are shared by the different modules of the framework.

- *The Client Module*
  This module is an implementation of the client API defined in the API module. It is implemented for accessing the WSAG4J web service stack.

- *The SLA Engine Module*
  This module is the core of the WSAG4J framework. It provides a generic implementation of a WS-Agreement based SLA engine. It implements standard functionality for processing agreement offers, creating agreements, monitoring the agreements runtime states, and evaluating and accounting agreement guarantees. Agreement acceptance policies and business logic for instantiating and monitoring SLA aware services can easily plugged in.

- The Web Service Module
  This module implements the remote frontend for the WSAG4J engine. It implements the WS-Agreement port types and delegates the calls to the WSAG4J engine, if necessary. The web service module is implemented on the base of the Apache Muse [46] framework. This framework provides the WSRF container required by the WS-Agreement specification.

- *The Server Distribution*
  This module comprises the WSAG4J server including the web service stack, the SLA engine and all required configurations. It is a packaged web application archive, which can easily be deployed in a wide set of application servers.

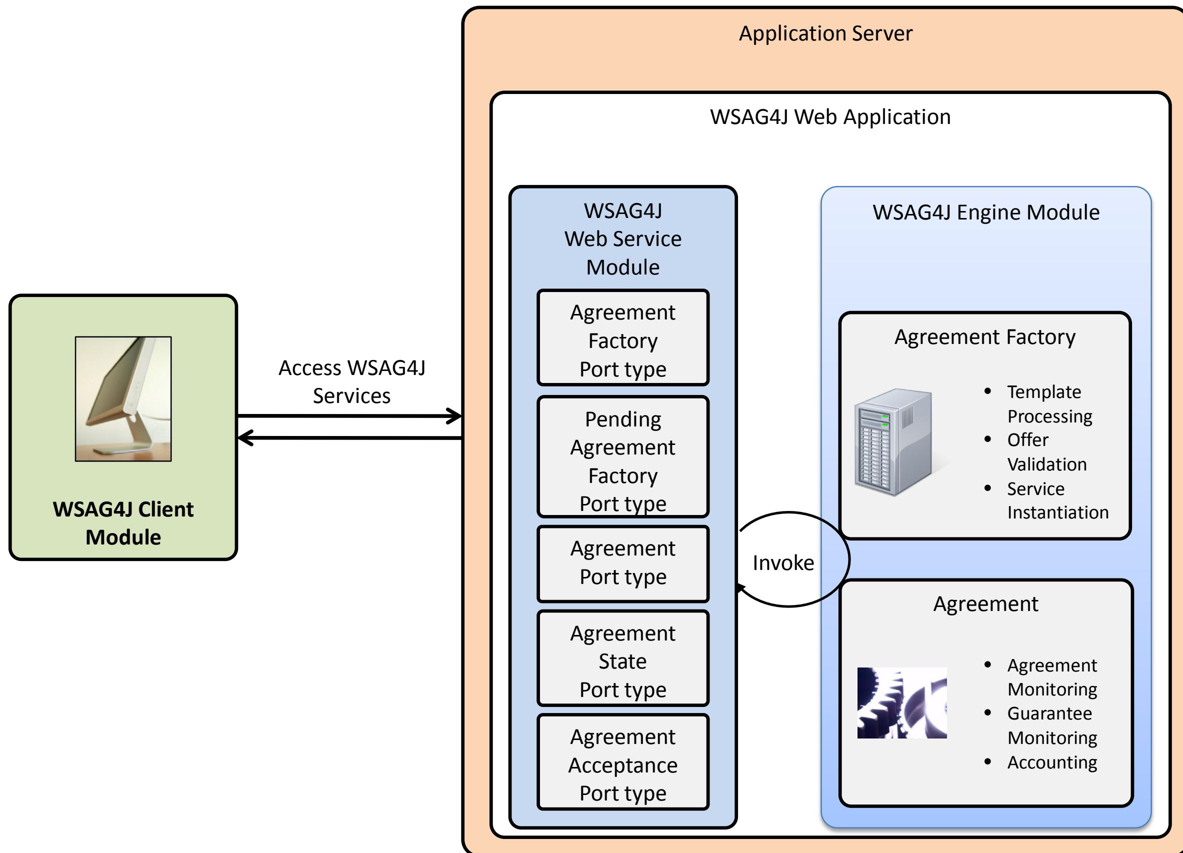


Figure 20: overview of the WSAG4J components

Figure 20 shows the basic modules of the WSAG4J framework and their deployment. The API module is not shown separately since it is essentially used by all other modules. Since the WSAG4J framework was designed in a very modular way, complete modules can easily be exchanged. The SLA engine can for example be used with a different WSRF stacks such as Globus or UNICORE. In that case, only the WSAG4J engine and the dependent modules (e.g. the API module and additional types) are required for the integration. It is also possible to use the WSAG4J engine without a remote stack at all, e.g. in a desktop application. Vice versa, the web application module and the web service module may be used with other WS-Agreement engine implementations. In this case these implementations need to implement the WSAG4J Server API.

3.3 Integration of the WSAG4J Engine into the WSRF Layer

A WSAG4J engine is usually accessed via the WS-Agreement protocol. This protocol is coupled with the Web Service Resource Framework (WSRF). The agreement factory and agreement instances are modeled as WSRF resource. A WSRF resource is comparable to an object in an object oriented programming language, but it can be accessed remotely via the SOAP protocol. Several implementations of the WSRF specification are available, for example the Apache Muse framework [46], Unicore WSRF Lite [47], the Globus Toolkit 4.x [3], WSRF::Lite [48], and the WebSphere Application Server V6 and V7 [49]. The WSAG4J web service module implementation is based on the Apache Muse [46] framework. The web

service module implements the basic port types defined by the WS-Agreement specification. These port types are:

1. Agreement Factory,
2. Pending Agreement Factory,
3. Agreement,
4. Agreement State,
5. and Agreement Acceptance.

As mentioned before, the web service module is one specific frontend of the WSAG4J engine. It accesses the WSAG4J engine via the agreement factory interface that is implemented by WSAG4J engine and allows creating agreements in a synchronous way. The result of the agreement creation process is an agreement object, which implement the agreement interface. The engine may return different agreement implementations, i.e. in order to support different agreement monitoring strategies. Figure 21 depicts the external interfaces of the WSAG4J engine.

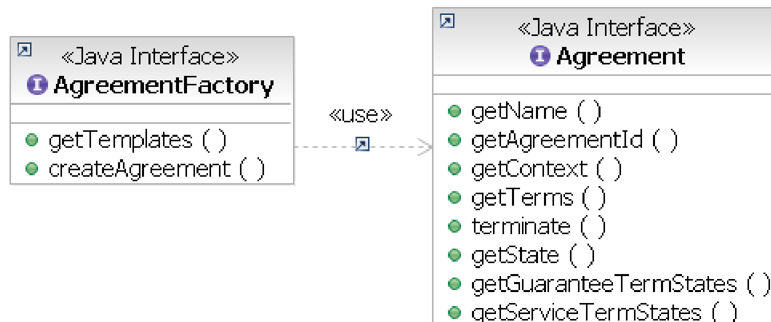


Figure 21: interfaces of the WSAG4J engine

The WSAG4J engine can be accessed via the WS-Agreement Agreement Factory port type as well as via the Pending Agreement Factory port type. The basic difference between these two port types is the usage scenario. The Agreement Factory port type is expected to immediately return a result whether or not an agreement offer was accepted. For scenarios in which the decision process of accepting or rejecting an agreement takes a long time, the synchronous creation of an agreement might not be feasible. For example the agreement creation process in distributed systems might fail due to timeouts of the network connection. The allocation of network resources might also be undesirable for a long lasting decision processes. Therefore, the Pending Agreement Factory allows creating agreements in an asynchronous way. A system that implements the Pending Agreement Factory port type returns an agreement instance immediately after the invocation, but this agreement instance is in the Pending state. It stays in this state until the decision to accept or reject the agreement offer has been made. An agreement initiator can optionally provide an endpoint reference to an Agreement Acceptance resource, which is notified of the acceptance or rejection of the agreement. If no acceptance endpoint is provided, no notification takes place. In this case, the agreement initiator can determine the acceptance of an agreement by polling the agreement state.

The WSAG4J framework implements the Agreement Factory port type and the Pending Agreement Factory port type via a so called *AgreementFactoryCapability*. This capability

maps web service calls to a Java implementation. It implements methods for retrieving agreement templates, creating agreements and pending agreements. It is the main integration point of WSAG4J engine and web service module. Web service calls for retrieving templates and creating agreements are delegated to the engine component. When an agreement is created via the *createAgreement()* method, the engine is invoked in a synchronous way. In case the agreement creation process is successful, i.e. a new agreement instance is returned by the engine, the *AgreementFactoryCapability* creates a new WSRF agreement resource and registers the agreement object with it. All calls to the agreement WSRF resource are delegated to the agreement object. The agreement factory capability then returns the endpoint reference of the agreement WSRF resource to the user. This process is shown in Figure 22.

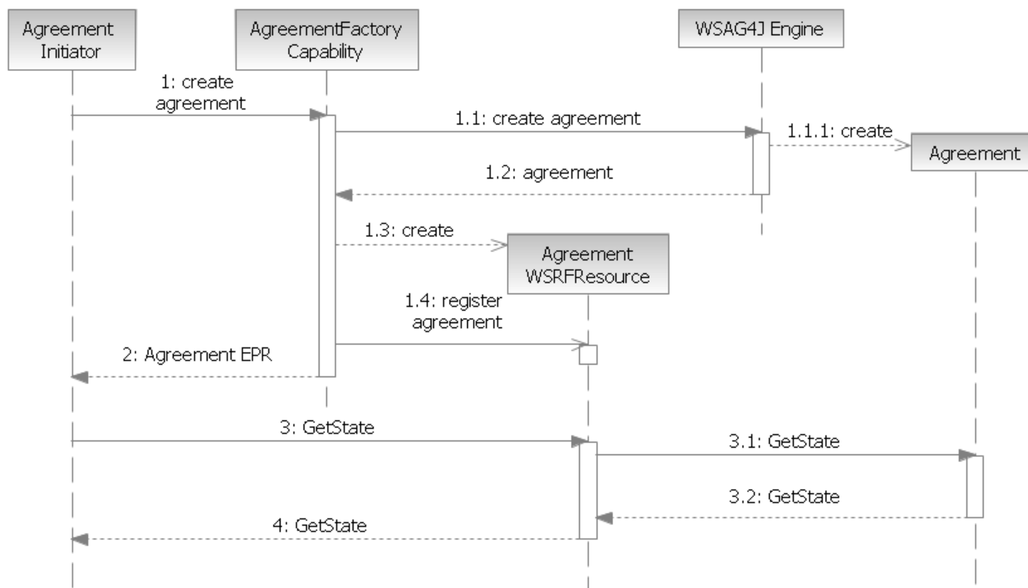


Figure 22: creation of an agreement via the agreement factory port type

Note that WSAG4J Engine and Agreement are components that are implemented as part of the WSAG4J engine module. The Agreement Factory Capability and the Agreement WSRF Resource are part of the WSAG4J web service module.

The agreement creation process is slightly different when the *createPendingAgreement* method of the *PendingAgreementFactory* port type is invoked. In this case a *PendingAgreement* instance is created. The pending agreement instance starts the agreement creation in a separate process and optionally notifies an agreement initiator on the acceptance or rejection of the agreement offer. This process is shown in Figure 23 and described below.

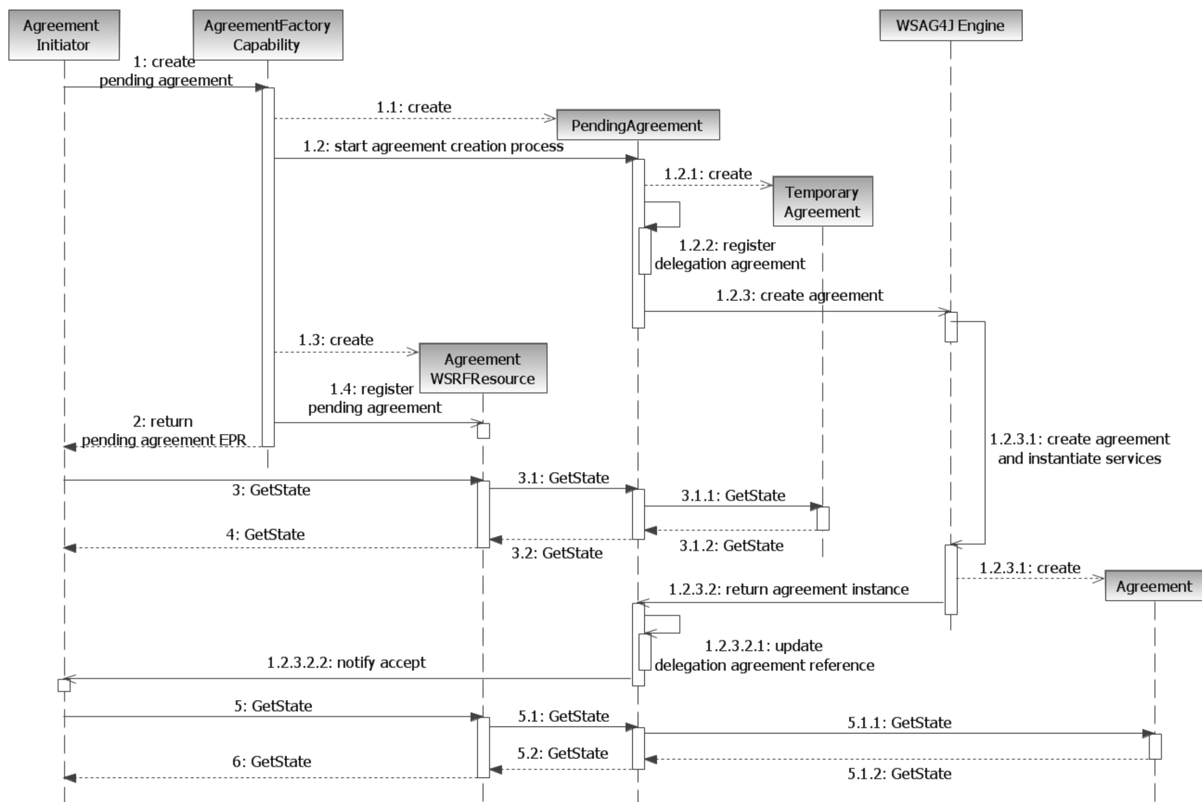


Figure 23: creation of an agreement via the pending agreement factory port type

When the createPendingAgreement method of the WSAG4J agreement factory service is invoked, the agreement factory capability creates a new *PendingAgreement* instance. The pending agreement instance is initialized by the factory capability with the agreement offer, a reference to the WSAG4J engine, and the agreement acceptance endpoint if provided. Next, the factory capability creates a new agreement WSRF resource and registers the pending agreement instance with this resource. Now, the factory capability instructs the pending agreement instance to start the agreement creation with the WSAG4J engine in a separate process. The factory capability then returns the endpoint reference of the agreement WS resource to the agreement initiator. The implementation of the *PendingAgreement* follows the state pattern [29]. Depending on its state, the pending agreement will alter its behavior, for example agreement monitoring is only done when the agreement is in the observed state. The pending agreement forwards all WS-Agreement specific calls to a delegation agreement instance, which represents the current state of the agreement. When a pending agreement is created, a delegation agreement is initialized with a temporary agreement instance based on the agreement offer. The temporary agreement instance represents the agreement in pending state. The agreement instance remains in the pending state until the WSAG4J engine returned from the agreement creation process. If the agreement creation was successful, the temporary agreement is replaced by the agreement instance returned by the WSAG4J engine. If an acceptance endpoint is supplied, the agreement initiator is notified of the agreement acceptance. If the agreement creation failed, the state of the temporary agreement instance is changed to *Rejected*. If an agreement acceptance endpoint is supplied, the agreement initiator is notified of the rejection. Note that the web service module uses the capabilities of the WSAG4J engine to create and monitor agreements. Protocol specific functionality such as

notifications and creation of the pending agreement are completely implemented in the web service module. The WSAG4J engine is described in the next section.

3.4 Architecture of the Agreement Factory

The agreement factory component is a central component of the WSAG4J framework. It is the primary interaction point for applications to create new agreements. Moreover, it is responsible for the agreement offer validation and service instantiation. Therefore, it is subsequently also called WSAG4J engine. An agreement factory has the following responsibilities:

Publish agreement templates

The agreement factory publishes the supported agreement templates via its *getTemplates()* method. In the WSGA4J framework each template represents a class of SLAs that is supported by the engine.

Validate incoming offers

The agreement factory supports the validation of agreement offers. Offer validation is performed based on the creation constraints defined in the template that was used to create the agreement offer. The agreement factory uses the *Agreement Offer Validator* to ensure the compliance of an offer with the template's creation constraints.

Instantiate agreements and agreed services

The agreement factory creates agreements for valid agreement offers and instantiates the services associated with the agreement. In a generic SLA framework custom SLA acceptance policies and service instantiation strategies must be supported. WSAG4J supports individual SLA acceptance policies and service instantiation strategies for each class of SLAs, which means for each individual template.

3.4.1 WSAG4J Agreement Factory Components

Figure 24 shows the basic components of the WSAG4J agreement factory. The WSAG4J engine implements the agreement factory interface. It encapsulates its subsystems and therefore acts as a facade. A facade is an object-oriented design pattern that provides "…a unified interface to a set of interfaces in a subsystem. A facade defines a higher level interface that makes the subsystem easier to use. [29]" The WSAG4J engine therefore provides a high level interaction point, which can be easily accessed through a web service interface. It provides a method to access the agreement templates that are registered with this engine and a method to create agreements based on incoming offers.

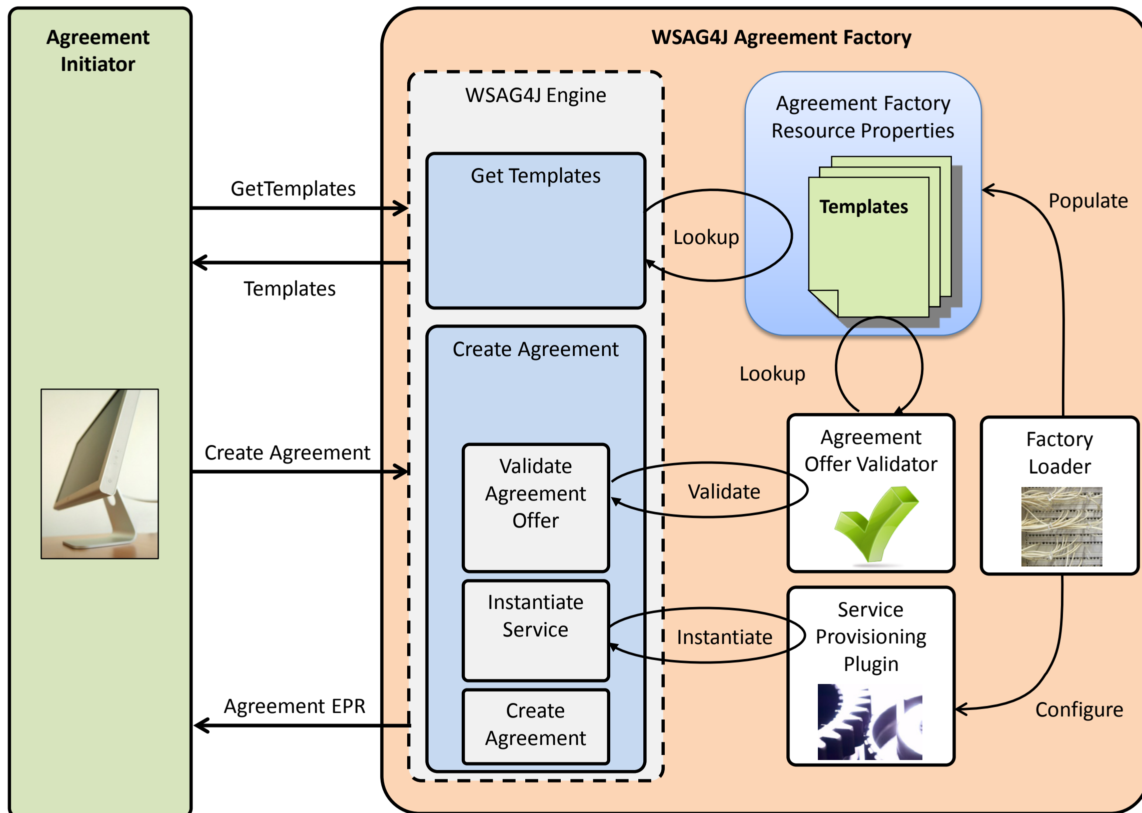


Figure 24: basic components of the WSAG4J agreement factory

The most important components of the WSAG4J engine are the agreement factory loader and the agreement offer validator. The agreement factory loader is responsible for bootstrapping the WSAG4J engine. This process consists of the following steps:

1. Load the agreement factory configuration
2. Load and process the configured templates
3. Configure and instantiate the required plug-ins for each template

A WSAG4J agreement factory configuration consists of a set of agreement factory plug-ins. Each agreement factory plug-in consists of a *GetTemplateAction*, a *NegotiationAction*, and a *CreateAgreementAction*. The *GetTemplateAction* implements the logic how to load and process the agreement template for a specific factory plug-in. The agreement template is uniquely identified within the WSAG4J engine by its name and id. These two attributes also build the compound key of an agreement factory plug-in. Each agreement template in the WSAG4J engine can therefore be resolved to one specific agreement factory plug-in. The *NegotiationAction* of an agreement factory plug-in implements the business logic to create counter offers for incoming negotiation offers. Accordingly, the agreement factory plug-in's *CreateAgreementAction* implements the agreement acceptance policy and business logic to create new agreements based on the associated template. Figure 25 shows the external and the internal view on a WSAG4J agreement factory. Externally the factory exposes its capabilities in form of agreement templates. Internally each agreement template maps to one agreement factory plug-in.

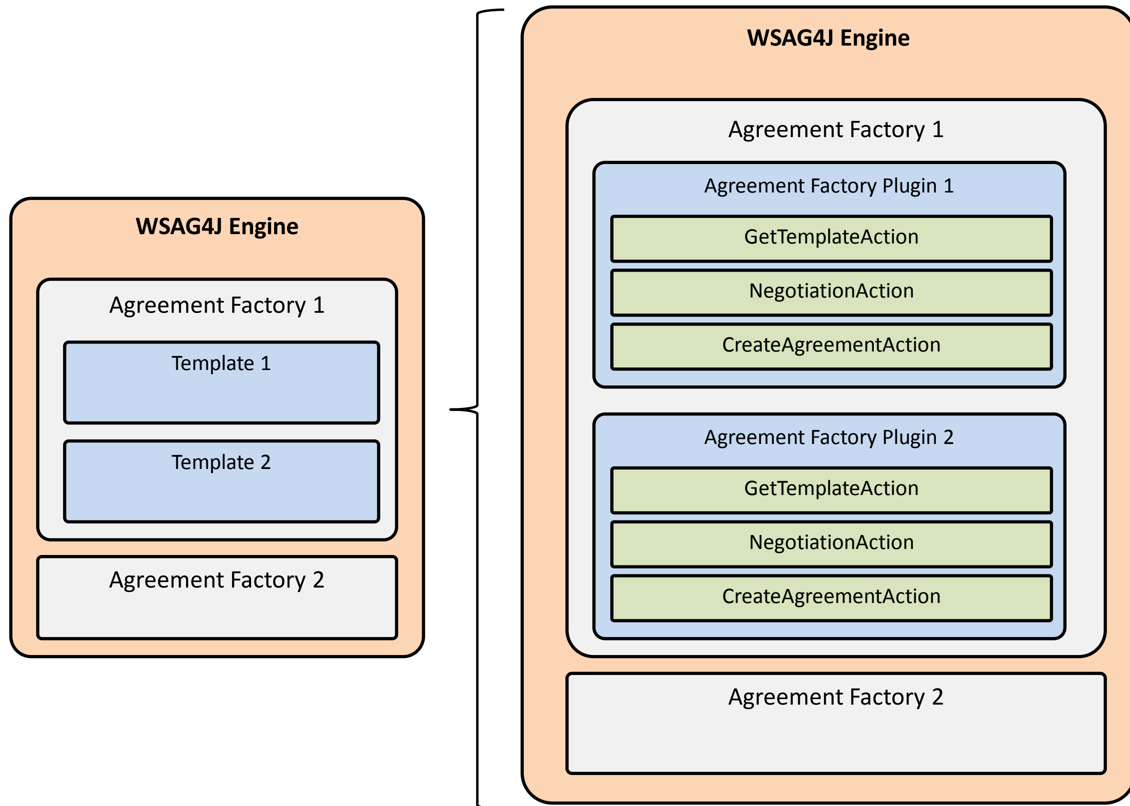


Figure 25: mapping of agreement templates to WSAG4J agreement factory actions

During the WSAG4J engine initialization process the agreement factory loader initializes all agreement factory plug-ins. This includes the initialization of the agreement templates. During this initialization process the agreement template is loaded from a template store. In a second optional step the template is processed. During the template processing dynamical data can be included into a template. After this step the template initialization process is completed. Besides the template action (*GetTemplateAction*) each factory plug-in comprises two additional actions. The first one is the negotiation action (*NegotiationAction*), which implements strategies for agreement negotiation. The second action (*CreateAgreementAction*) implements the business logic for accepting or rejecting agreement offers and to instantiate the associated services for accepted agreements. These actions are also configured by the agreement factory loader during the bootstrapping process. Once all agreement factory plug-ins have been initialized by the factory loader, the loader updates the agreement factory internal resource properties document with the factory action templates. This completes the WSAG4J engine initialization process. External entities can query the agreement factory for available templates and negotiate or create agreements based on these templates.

How agreements are created by the WSAG4J agreement factory is described next. This process is depicted Figure 26 in detail.

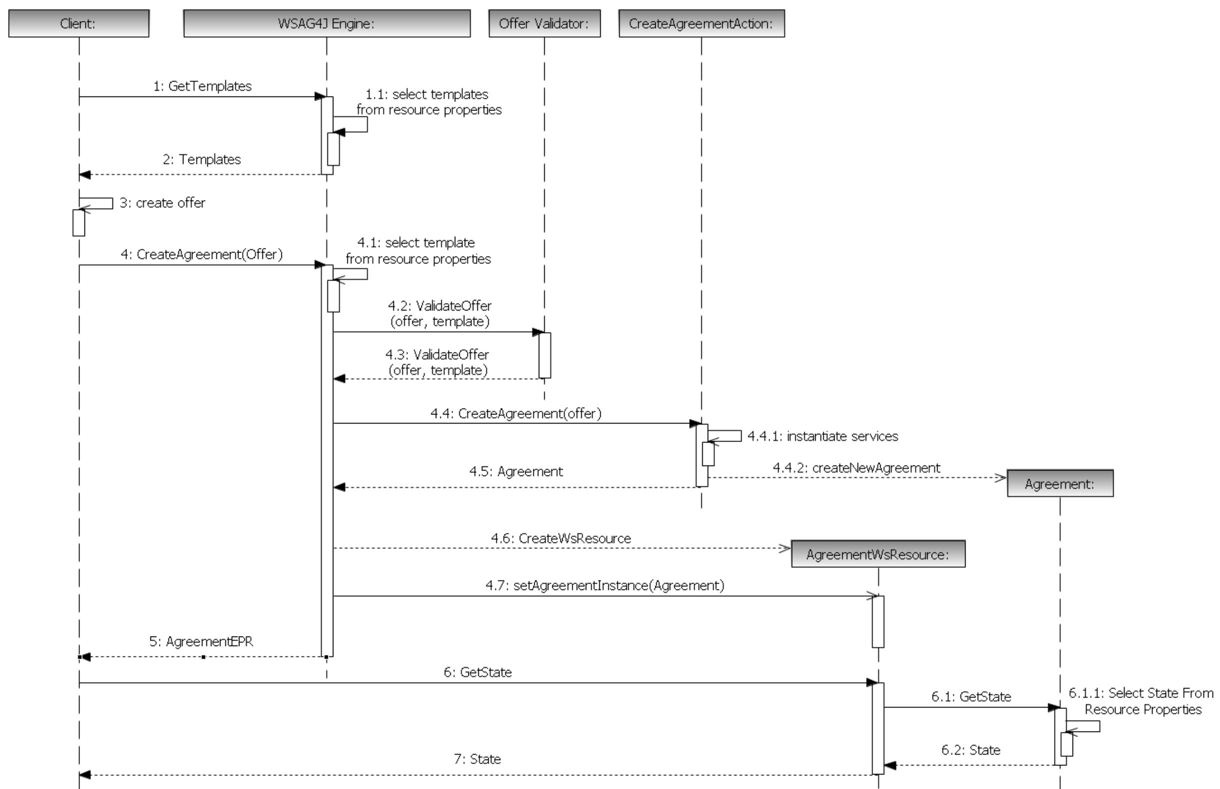


Figure 26: detailed description of the agreement instantiation process

In the WSAG4J framework a client interacts with the WSAG4J engine via the agreement factory interface. In order to create a new agreement with a WSAG4J agreement factory, a client first queries the available templates. This is a required step since the WSAG4J framework only accepts agreement offers that are based on known agreement templates. Now the client creates a new offer based on one of the templates. The agreement offer specifies the template name and version it is based on. Then the client invokes the *createAgreement()* method of the agreement factory, passing the offer as a parameter. When *createAgreement()* is invoked, the WSAG4J agreement factory first checks the validity the offer. The validation is performed by the *Agreement Offer Validator.* This process consists of two steps: 1. the template that was used to create the offer is looked up by the agreement factory and 2. the offer is validated for conformance with the template by the Agreement Offer Validator. Only if this validation process succeeds, the offer is processed further. In case the offer validation is successful, the agreement factory plug-in that is associated with the originating template is resolved and the associated *CreateAgreementAction* is invoked. The *CreateAgreementAction* implements the business logic for instantiating the required services for the SLA. Moreover, this action instantiates a new agreement object. The agreement object is then returned by the WSAG4J agreement factory to the calling entity, e.g. the Agreement Factory Capability implemented in the WSAG4J web service module. This entity can use the agreement object e.g. to monitor the state of the agreement or to terminate an agreement instance.

3.4.2 Agreement Offer Validation

As mentioned before, an agreement template can contain a section of creation constraints. The agreement responder uses creation constraints to define the structure and possible values of valid agreement offers that are based on a particular template. Creation constraints provide an

automated way to assert that agreement offers are created matching the agreement responder's expectations. This mechanism helps to protect systems from illegal manipulation of input values. This is especially important for systems that create and manage service level agreements, which are essentially electronic contracts. Since SLAs contain a description of the service to deliver and optionally business values such as pricing, the illegal modification of agreement offers must be prevented. This is an important requirement for dynamic SLA provisioning in distributed environments. The WSAG4J engine supports this requirement by implementing a well defined and robust offer validation process. Only offers that are valid with respect to its template creation constraints are accepted by the framework.

WS-Agreement distinguishes between two types of creation constraints, *Item Constraints* and *Free Form Constraints*. Item constraints represent the default offer validation model in WS-Agreement, while free form constraints are an extension point for using other offer validation models in conjunction with WS-Agreement. WSAG4J implements a generic offer validation process based on the WS-Agreement item constraint model. An item constraint is defined by an offer item element in an agreement template's creation constraints section. Each item constraint refers to a set of elements in an agreement offer. These elements are subsequently called *Offer Items* in order to distinguish between the item constraint and the restricted elements. This section describes how the compliance of agreement offers is validated with respect to the creation constraints defined in a template. Additionally the limitations of the offer item model in WS-Agreement are discussed and it is shown how to overcome these limitations.

An offer item constraint has a name that can be used to easily identify it in an agreement template. This can for example be useful for an agreement initiator to find a specific item constraint that specifies value ranges for a specific property. For that purpose, the structure of the agreement template and the names of the offer items must be known to the agreement initiator before creating an agreement offer. For automatic constraint validation, the item name is usually not needed.

Besides the name, the offer item constraint has a location element. The location contains a machine processable expression to select document fragments (offer items) from an agreement offer. These offer items are the elements of an agreement offer that are restricted by the offer item constraint. The location can for example contain an XPath expression [31]. An offer item constraint can refer to zero to n offer items. Each offer item must satisfy the constraint specified in the *Item Constraint* element. Only if all offer items are valid according to the specified item constraints, the validation of the offer item succeeds. An agreement offer is only compliant to an agreement template when the validation process succeeds for all *Creation Constraints* in the template.

3.4.2.1 Item Constraint Model

The WS-Agreement specification defines the content of an Item Constraint as a choice between the xs:simpleRestrictionModel group and the xs:typeDefParticle group. Both groups are defined as part of the XML Schema specification. The xs:simpleRestrictionModel is defined in [25] and the xs:typeDefParticle is defined in [24]. Both model groups are fundamental pillars of the XML Schema grammar.

Value Constraints

The simple restriction model of XML Schema allows defining restrictions on the value spaces of simple types. In XML Schema simple types are atomic data types such as xsd:string, xsd:decimal, xsd:integer, xsd:float, xsd:boolean, xsd:date, xsd:time, xsd:QName, or xsd:anyURI. This means that offer item constraints based on the simple restriction model must refer to offer items that are typed as simple types. The simple restriction model can contain a definition of a simple type and/or a definition of a group of facets.

A simple type definition is either a restriction of an existing simple type, a union of simple types or a list of simple type values. Restrictions are rather simple. They are defined on the basis of an existing simple type. Additionally, restrictions can include a set of constraining facets. Constraining facets are defined in section *Constraining Facets* in [25] as "…optional properties that can be applied to a data type to constrain its value space". Constraining facets are xs:length, xs:minLength, xs:maxLength, xs:pattern, xs:enumeration, xs:whiteSpace, xs:maxInclusive, xs:maxExclusive, xs:minExclusive, xs:minInclusive, xs:maxInclusive, xs:totalDigits, and xs:fractionDigits. Depending on the base type of a restriction only a subset of the constraining facets is applicable. For example the xs:length facet can be specified for restrictions with the base type xs:string, but the xs:minInclusive facet would be invalid for these restriction. The example in Listing 7 shows an offer item constraint that refers to offer items that must contain an integer value from 1 to 100.

```
<wsag:Item wsag:Name="SimpleTypeRestriction">
  <wsag:Location>…</wsag:Location>
  <wsag:ItemConstraint>
    <xs:simpleType xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:restriction base="xs:int" >
        <xs:minInclusive value="1" />
        <xs:maxInclusive value="100" />
      </xs:restriction>
    </xs:simpleType>
  </wsag:ItemConstraint>
</wsag:Item>
```

Listing 7: restriction of the value space of an integer

Besides restrictions, the simple restriction model also allows to define lists and unions. [25] provides the following definitions for atomic data types, lists and unions:

- Atomic data types are those having values which are regarded by this specification as being indivisible.
- List data types are those having values each of which consists of a finite-length (possibly empty) sequence of values of an atomic data type.
- Union data types are those whose value spaces and lexical spaces are the union of the value spaces and lexical spaces of one or more other data types.

In an XML document a list is a set of atomic values that are separated by white spaces. The content of a list can be restricted by providing an enumeration of valid values. Moreover, the list can have associated constraint facets that restrict for example its length, which is the number of elements contained in the list.

Structural Constraints

WS-Agreement uses the xs:typeDefParticle group in order to define structural constraints on offer items. While the simple restriction model focuses on the definition of value spaces of simple types, structural constraints define how the content of offer items is structured. They specify which child elements can be part of a certain offer item, the order in which these child elements must occur, the cardinality of each child element, or how child elements are grouped. The statements that can be used in a structural *Item Constraint* are described below.

All statement

The *All* statement restricts the structure of an XML document fragment in a way that all elements in this statement must be present in the document fragment. The order of the elements in the document fragment is not important. The *All* statement in the xs:typeDefParticle group is defined in the XML Schema definition as an xs:all element.

Sequence statement

The *Sequence* statement requires that all elements that are defined in a sequence must be part of the offer item. In contrast to the *All* statement, the order of the elements is significant. The *Sequence* statement in the xs:typeDefParticle group is defined in the XML Schema definition as an xs:sequence element.

Choice statement

The *Choice* statement of an *Item Constraint* requires that one of the elements defined in the choice must be part of the referenced offer items. The *Choice* statement in the xs:typeDefParticle group is defined by the XML Schema definition as an xs:choice element.

Group statement

The *Group* statement references a group definition in an external, domain specific schema. Groups allow specifying how specific elements in a document fragment are grouped together. Such groups may consist of *All* statements, *Sequence* statements, and/or a *Choice* statements. In order to validate an *Item Constraint* that uses group references, the schema with the group

definition must be available at validation time. The definition of a group is not part of the item constraint. The *Group* statement in the xs:typeDefParticle group is defined in the XML Schema definition as an xs:groupRef element.

Each of the above mentioned statements can include an occurrence attribute that defines how often the statement may occur in the referenced offer items. Structural *Item Constraints* can easily become very complex. The following example illustrates the usage of structural constraints in agreement templates. In this example a service provider offers a set of predefined computational services to its customers. Customers can choose exactly one of these predefined services from the agreement template when an agreement offer is created. Additionally, the provider defines a performance guarantee in the template. This guarantee is applied to the service chosen by the customer. The service provider expects that a valid agreement offer has exactly the same structure as the one provided in the template. Moreover, the service provider requires that the service consumer selects the service it is interested in from the agreement template. The service consumer does this by removing all other service description terms from the *ExactlyOne* element when creating the offer. Valid agreement offers therefore contain only one service description term within the *ExactlyOne* tag. These structural requirements are enforced by the service provider by defining a corresponding creation constraint in the template. The *Item Constraint* shown in Figure 27 implements such a creation constraint. It refers to the *Terms* element in an agreement offer and enforces that this element contains exactly one *All* statement. The *All* statement must in turn include one *ExactlyOne* statement and one guarantee term. Moreover, the *ExactlyOne* statement must include one service description term. The service description term is the one the agreement initiator has selected for execution. The *Item Constraint* also enforces that this service description term contains a JSDL job definition document which describes the service to execute. An XML representation of such a structural constraint can be found in the Appendix in section Agreement Template Example.

Besides the restriction of the agreement term compositor structure, the structure of service and guarantee terms should also be restricted. Moreover, values that can be specified in agreement offers should be restricted by using value constraints as described before.

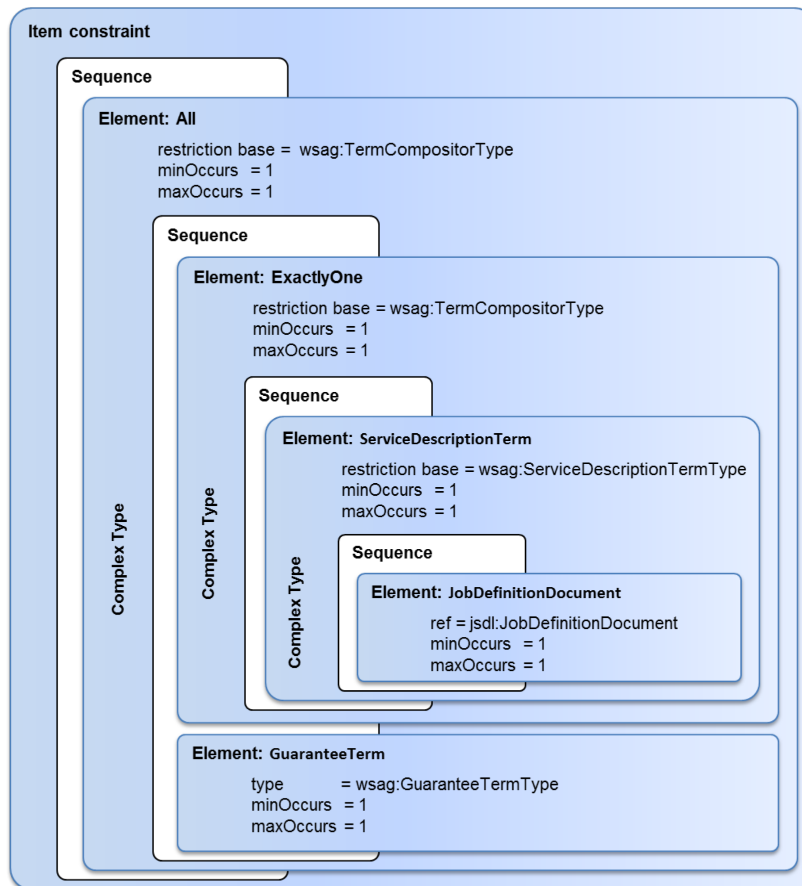


Figure 27: example of a structural creation constraint

Discussion of Creation Constraints

It is general considered a best practice to use standardized description languages to define service description terms in an agreement. By using such standards the interoperability and acceptance of systems is improved as they define a clear way to express common problems. Therefore, standards are often designed to be flexible and multi-purpose in order to be applicable in a wide range of scenarios. The demand for defining multipurpose standards has also a disadvantage. They define lots of elements that cover a wide range of application scenario. Since the application scenarios may differ, a lot of these elements are defined as optional. This makes the schema definitions are often very fuzzy. Besides that, these standards often provide a lot of extension points to include domain specific information that is not covered by the standard itself. This makes it very difficult for applications to validate whether a document includes all required information or not. Moreover, it is very hard to ensure that all information that is provided in a document is really processed. This is of particular importance for Service Level Agreements. Since SLAs as discussed here are a kind of electronic contracts that can be negotiated and created in an automatic way, agreement responders need a method to define acceptable representations of service description using well known standards.

The Job Submission Description Language [50] is such a generic standard in the area of high performance computing. It is used to describe computational jobs for submission to compute

resources in Grid computing. For the reasons described before, JSDL defines most of the elements in its data structures as optional. JSDL is often used with WS-Agreement in order to describe computational services. JSDL documents are therefore included in the corresponding service description terms of a template. These JSDL documents can be restricted by using the *Item Constraints* presented before. It is for example possible to define a constraint that enforces that a JSDL job description document contains an application, data staging, and resources element. *Item Constraints* therefore provide a mechanism that can be used together with common standards in order to profile how exactly the standard is applied an agreement offer by defining the structure and valid values for them. Moreover, the *Item Constraints* can also be used to restrict the structure of the overall agreement. They can also support the agreement initiator to find out acceptable values for service descriptions. On the other hand, they protect the agreement responder from accepting offers that are created in an illegal way. With the help of creation constraints the input parameters of SLA aware service provisioning systems can now be defined very strictly. This also helps to increase the stability of these systems. Agreement templates that are fully restricted by creation constraints allow agreement initiators to find out exactly which parameters they may specify or change in an agreement offer. Agreement responder in turn can specify agreement templates in a way that each aspect specified in a valid offer is really processed. This helps agreement initiator and responder come to a common understanding of the service that is provided in the context of a SLA.

3.4.2.2 Limitations of the Offer Item model

As described in the previous section, the WS-Agreement Creation Constraint model already provides a very powerful way to ensure the validity of agreement offer. However, the work on agreement offer validation presented in this thesis has shown two essential limitations of the constraint model. These limitations are:

- Ensuring the cardinality of offer items is not included in the current model
- Specific XML structures are hard to process with Item Constraints

The WSAG4J framework resolves these limitations as described below:

Ensuring the cardinality of selection results

An item constraint refers to a set of offer items via its location. Due to unforeseen manipulation of an agreement offer or due to design issues of the creation constraints the selection result might not meet the expectations during design time, e.g. a item constraint is not validated when the offer item selection process does not return a result. Offer item constraints that are processed with the WSAG4J framework can therefore include a special constraint annotation document. The constraint annotation document is added in the extension point of an offer item constraint. The constraint document allows specifying the expected multiplicity of offer items returned by the selection process. The offer validation process only succeeds in case of the number of referenced offer items match the number of expected offer items and the validation of each offer item is successful.

Validation of hard to process structures

Specific constructs in agreement offers are hard to validate with a grammar based language such as XML Schema that is used for the offer item constraints. A common example is the following: when an agreement initiator specifies one specific attribute (e.g. amount) it must also specify a second attribute (e.g. unit) accordingly. Another example is when the initiator should select one service term that contains a JSDL application out of a set of service terms (e.g. in an *ExaclyOne* statement). The offer item constraint should then validate that the application name and version match the values provided in the template. This is illustrated in Figure 28.

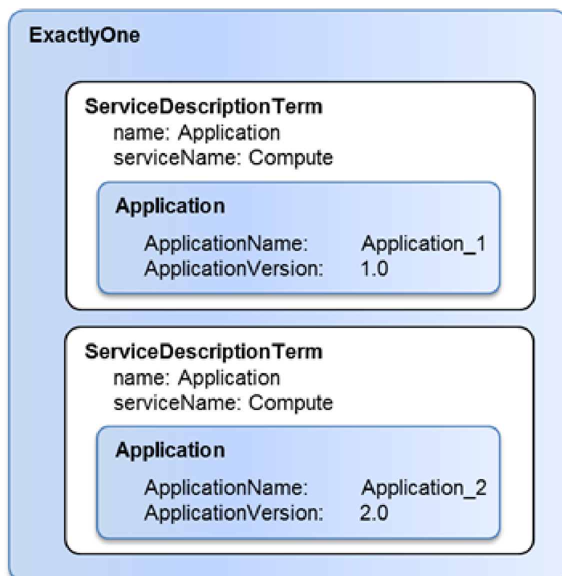


Figure 28: choice of different service terms

The example shows a list of service description terms in an *ExactlyOne* statement. Valid offers must contain only one service description term with the name *Application* and the service name *Compute*. This is enforced by a structural constraint. An additional creation constrain should now ensure that the service description term in an offer specifies application name and version according the agreement template, e.g. if the service description term contains "*Application_1*" as application name then the application version must be "*1.0*". This can't easily be done with the WS-Agreement offer item constraints. However, WS-Agreement also allows specifying so called *Free Form Constraints*. This is an extension mechanism to include additional offer validation models. One solution that addresses the validation problems described above is to introduce a new set of offer item constraints based on the Schematron language [51]. Schematron is a rule based validation language. Instead of defining a grammar that is used for a document validation, Schematron defines a set of assertions that check the presence or absence of patterns in an XML document. These patterns can for example define that the content of one element is controlled by the content of one of its siblings, e.g. if application name is *Application_1* the application version must be *1.0*. Creation constraints based on Schematron are not a replacement for the offer item constraints described in the previous section. They rather complement these offer item constraints. The WSAG4J framework does not support Schematron constraints by now, but it is planned to extend the framework accordingly in a future version.

3.4.3 Constraint Validation in the WSAG4J Engine

Validation of agreement offers is a fundamental functionality of the WSAG4J framework. For each incoming offer the WSAG4J Engine first looks up the template that was used to create the offer. In the next step it validates the offer items for each offer item constraint. Only if the offer validation succeeded the agreement creation process is started. As seen in the previous

sections, the offer item constraints can easily become very complex. Since a generic WS-Agreement engine as WSAG4J must in principle support any possible *Item Constraints*, such an engine must be capable of handling the WS-Agreement item constraint model, namely the xs:simpleRestrictionModel and the xs:typeDefParticle. A simplistic approach to the offer validation would basically require to interpret the content of an *Item Constraint* and to check the referenced offer items accordingly. This would result in writing a new schema-validating XML parser. Since this approach is rather ineffective another solution was required.

The fundamental idea behind the *WSAG4J Agreement Offer Validator* is based on the fact that there is a variety of validating XML parsers publicly available. These parsers already implement XML Schema validation, and therefore also the models that are used to define offer item constraints. If it is possible to take advantage of one of these parsers for the agreement offer validation, offer item constraints could be supported in a very generic way. So the remaining question is how to take advantage of the validation capabilities of such a parser. The validation problem is divided in two sub problems:

1. Validation of constraints based on the xs:simpleRestrictionModel group
2. Validation of constraints based on the xs:typeDefParticle group

The WSAG4J Agreement Offer Validator solves these validation problems in a very efficient way. During the validation process the offer validator dynamically creates a XML Schema based on the contents of an *Item Constraint*. Therefore, the validator first analyses the content of the item constraint. If the item constraint is a restriction based on the xs:typeDefParticle group, it must contain an all statement, a sequence statement, a choice statement, or a group reference. In this case a validation schema for a complex type restriction is generated. If the *Item Constraint* does not contain one of the above mentioned elements, a schema for simple type validation is generated. Generating a validation schema based on the content of the *Item Constraints* is easily possible since the xs:simpleRestrictionModel and xs:typeDefParticle are also used by XML Schema to define simple type definitions and complex type definitions. Therefore, it is in principle possible to create a XML Schema definition for an *Item Constraint*, the validation problem transforms to the following sub problems:

1. Validation of the correct definition of an Item Constraint
2. Creation of a schema for simple type validation
3. Creation of a schema for complex type validation
4. Validation of the offer items against the creation constraint schemas

The following section shows the overall offer validation process as it is implemented in the WSAG4J agreement offer validator. Then it describes in detail how the above mentioned problems are solved in the WSAG4J solution.

3.4.3.1 Validation of an Item Constraint Definition

This point is rather simple, but nevertheless required. An *Item Constraint* may either contain elements of the xs:simpleRestrictionModel group or the xs:typeDefParticle group. A mixture of elements of both groups is not allowed. This is important since the generated constraint validation schema must either contain a simple type definition (xs:simpleRestrictionModel) or

a complex type definition (xs:typeDefParticle). The validation of the correct definition of the *Item Constraints* of a template is done when the templates are loaded. During this process all agreement template documents are validated against the WS-Agreement schema. This schema defines the content of an *Item Constraint* as a choice between the xs:simpleRestrictionModel group or the xs:typeDefParticle group. The validation process already ensures that an *Item Constraint* is defined syntactically correct.

3.4.3.2 Schema Creation for Simple Type Validation

In case an offer item constraint is based on the xs:simpleRestrictionModel, the Agreement Offer Validator creates a validation schema with a simple type definition. All offer items that are referenced by an item constraint are dynamically validated against this schema. The following example describes in detail how such a validation schema is generated. The *Item Constraint* in that example defines a simple restriction of an integer value. The restriction ensures that the values of referenced offer item are in the range of 1 to 100. The target namespace of the generated validation schema can be domain specific. Since the validation schema contains only a simple type definition, no additional information for the schema element itself is required, e.g. the attributes elementFormDefault or attributeFormDefault of the schema element don't have an impact on the validation process. Next, the top level simple type for the validation is generated. This type definition must specify a name, e.g. *GeneratedConstraintValidationType*. Moreover, the type must contain a restriction element. The content of the restriction element is defined by the XML Schema specification as xs:simpleRestrictionModel. This means that the content of the *Item Constraint* element can now simply be copied into the restriction element. Figure 29 shows the item constraint, the resulting validation schema, and how they relate to each other.

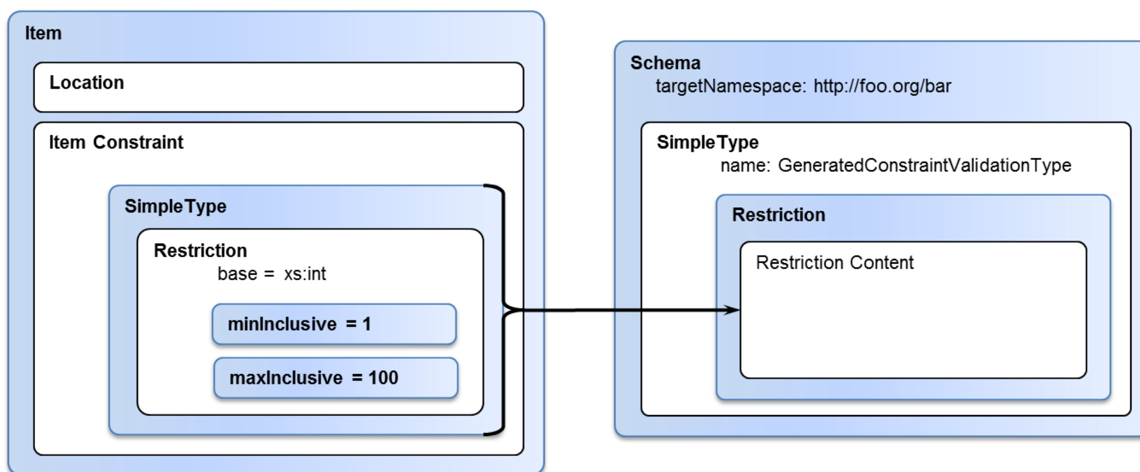


Figure 29: creation of a validation schema for simple type validation

At the first glance it may look like the top level simple type definition in the validation schema is redundant, but this is not the case. According to the simple restriction model, the *Item Constraint* does not require the definition of a simple type. It could rather only include facets, such as minInclusive or maxInclusive. Then the top level simple type restriction would only include these facets and no nested simple type. In that specific case the restriction element of the top level simple type definition must additionally specify the type of the referenced offer item as restriction base. Therefore, the offer validator needs the capability to

dynamically determine the schema type of the referenced offer items. XML parsers generate this kind of information (e.g. which element has which schema type) as a result of a schema validation process. It is called Post Schema Validation Infoset (PSVI). Moreover, a number of schema-aware XML processors such as XmlBeans [52] or Xerces-J [53] expose this information to applications. Given that the schema type of an offer item can be determined at runtime the validation schema can be generated for all of the described scenarios. The capability to dynamically determine the schema type of an offer item is also of particular importance for the generation of complex type validation schemas. This process is described now.

3.4.3.3 Schema Creation for Complex Type Validation

Structural item constraints are the second class of offer item constraints. These constraints are based on the xs:typeDefParticle group. The process of generating a validation schema for a structural constraint is in the following described with the help of the simple example. It defines a simple *Item Constraint* that restricts the structure of the *Terms* element in an agreement offer. The XPath expression in the item constrain location element is set accordingly. The item constraint restricts the content of the *Terms* element. The *Terms* element must contain exactly one *All* element of the type wsag:TermCompositorType. The *All* element must in turn contain exactly one ServiceDescriptionTerm element of the type wsag:ServiceDescriptionTermType.

The validation schema generation for structural constraints differs slightly from the process used for value constraints. In the first step, a new validation schema is created. In contrast to a value constraint validation schema, the target namespace of the structural constraint validation schema must match the namespace of the type of the referenced offer item. This is due to the fact that the validation schema contains a restriction of the offer item type definition. For such restrictions to be valid, the namespace of the restricted base type (type of the offer item) must match the namespace of the restriction type (generated constraint validation type). Moreover, the elementFormDefault value of the validation schema must match the corresponding value of the offer item type definition. This means if the type of the offer item is specified to be fully qualified, then the elementFormDefault property of the generated schema must also be set to “qualified”. In the next step, the schema definition of the offer item type is included into the validation schema. In the given example the offer item defined as an instance of the type wsag:TermCompositorType. Therefore, the WS-Agreement type system must be included into the validation schema. Depending on the implementation of the Agreement Offer Validator this is done for example by generating an appropriate include statement in the validation schema. Now the constraint validation type is generated. This time the type definition is a complex type with the name *GeneratedConstraintValidationType*. The validation type has a complex content that restricts the offer item type. The restriction base of the *GeneratedConstraintValidationType* must be set accordingly. In the example the restriction base is the wsag:TermTreeType, which is the according data type of the *Terms* element in an agreement offer. Here again, the Agreement Offer Validator must be capable to determine the schema type of an offer item dynamically. This is done in a similar way as described in the last section. Now, the content of the *Item Constraint* element are copied into the restriction element. The resulting validation schema is now complete and can be used for

the offer item validation. Figure 30 illustrates the validation schema creation process for the structural constraint and shows the relation of the item constraint and the generated schema.

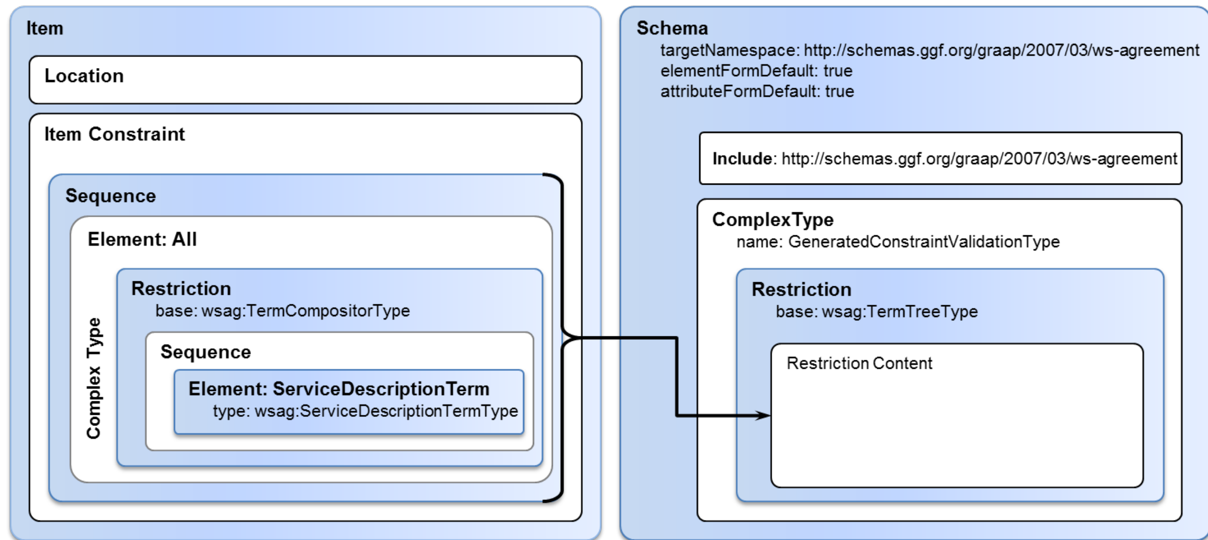


Figure 30: creation of a validation schema for structural constraints

3.4.3.4 Validation of the offer items with the creation constraint schemas

Given that an Agreement Offer Validator created the validation schemas as described in the previous sections, it now needs to validate the offer items with the generated schemas. This process is quite straightforward. In a first step, the validator must change the type of the offer item to the *GeneratedConstraintValidationType* defined in the validation schema. In a second step the offer item is re-validated. Changing the type of the offer item is done by using the xsi:type statement. The type specified in the xsi:type statement must either be the type of the offer item or any type derived from it. In our validation scenario the qualified name of the before created *GeneratedConstraintValidationType* is specified. Depending on the implementation of the offer validator, the offer item may additionally need to reference the generated constraint validation schema via the xsi:schemaLocation element. Now the offer item is re-validated. One easy way to achieve this is to serialize the offer item and parse it again with a validating parser. Again, this is specific to the implementation of the agreement offer validator.

The outcome of the offer item validation assesses if one particular offer item is valid according to its item constraint. The validation of one particular creation constraint is only successful if the validation process for all referenced offer items succeeds. An agreement offer is only valid if the validation of all creation constraint contained in the originating template succeeds. If a valid agreement offer was received, the WSAG4J agreement factory calls the appropriate *CreateAgreementAction* that was configured for the originating template. The CreateAgreementAction implements the agreement acceptance policy and the service provisioning logic for this specific class of agreements.

If no template was used to create an agreement offer, it is in principle still possible to determine whether an offer can be accepted or not. Therefore, the agreement offer is checked against the creation constraints of each template that is supported by the WSAG4J engine. If the offer is valid for one of the templates, the according service instantiation logic is invoked.

However, this approach requires that the deployed templates define structure and value constraints in a rigorous way. The creation constraints must be comprehensive for all deployed templates. This means that the offer validation must only succeed when all information that is required to create an agreement is really specified in the agreement offer and all information in the agreement offer is really processed.

The preceding section described the agreement offer validation process, as it is implemented in the WSAG4J framework. Agreement offer validation precedes the agreement creation process. Only valid offers are further processed by the WSAG4J engine, invalid offers are directly rejected. The framework automatically asserts the compliance of incoming agreement offers with respect to the creation constraints defined in the agreement template. WS-Agreement creation constraints can be defined at agreement template design time. They are automatically enforced by WSAG4J during runtime. This reduces the complexity of the input validation for SLA management systems significantly and therefore fosters the acceptance of such systems. While the preceding section described the creation of agreements, the following section describes the monitoring of existing agreement instances.

3.5 Agreement Monitoring System Architecture

As described before, the WSAG4J framework provides the required mechanisms to validate incoming agreement offers based on the creation constraints defined in an agreement template. Therefore, the structure and values of agreement offers can be defined in detail. This mechanism prevents the processing of invalid agreement offers by the service provisioning plug-in (*CreateAgreementAction*) in the WSAG4J framework. The service provisioning plug-in implements two important aspects of an agreement instantiation process: (1) the decision policy whether or not to accept an agreement offer, and (2) the agreement specific service provisioning strategy. Both aspects are domain specific. On the one hand different classes of agreements may require different agreements acceptance policies. On the other hand the provisioning of the agreed services usually varies for different classes of agreements. As a consequence, these two aspects are subject of domain specific implementation in the WSAG4J framework.

When a service provisioning plug-in is successfully invoked by the WSAG4J engine, it returns a new agreement instance. The agreement instance comprises the description of the agreed services and guarantees that are defined in the agreement offer. During the lifetime of the agreement, these guarantees must be monitored and in case of fulfillment and/or violation of the agreed guarantees the according rewards and/or penalties must be enforced. The WSAG4J framework provides a generic mechanism to monitor the compliance of guarantees while an agreement is active. This is done by the *Agreement Monitor* that is part of a *WSAG4J Monitored Agreement* instance. Figure 31 illustrates the overall architecture of the Monitored Agreement.

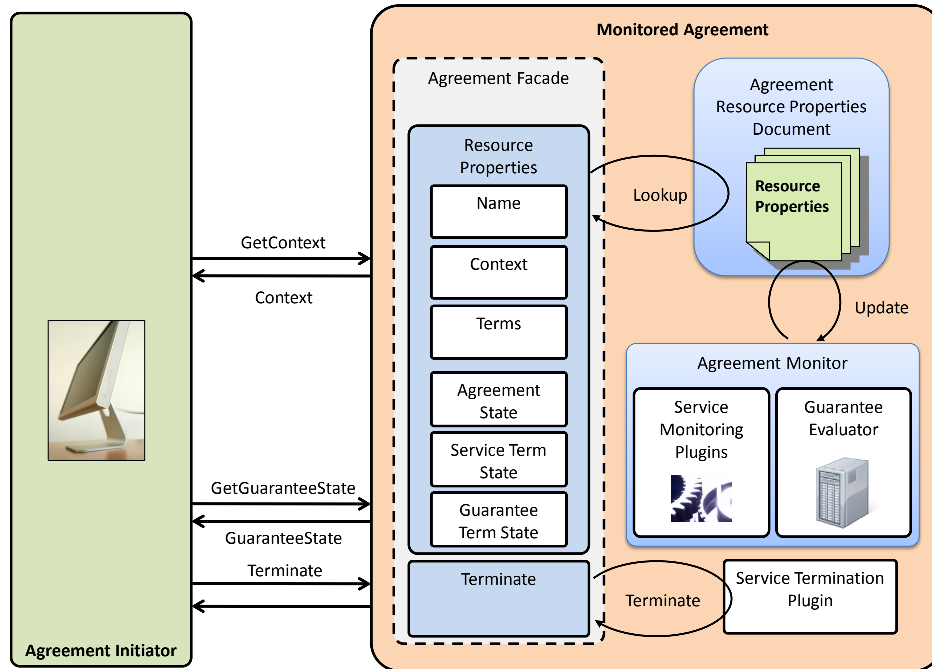


Figure 31: components of a monitored agreement instance

The Monitored Agreement is a specific implementation of the *Agreement* interface that is defined in the WSAG4J API module. It implements a generic method to evaluate the guarantees of an agreement based on the agreement service term states. A service term state exposes the current state of a service, which is defined in the context of an agreement by a *Service Description Term*. These states are retrieved by the service monitoring system and are updated in predefined intervals. The monitored agreement acts as a facade in order to provide easy access to the agreement properties and to encapsulate the monitoring and accounting subsystems. It also provides the required functionality to call the agreement termination strategy via the *Terminate* method. The agreement termination strategy can be configured during the agreement creation process.

3.5.1 Processing Agreement Resource Properties

One of the basic paradigms of WS-Agreement is the object oriented design. The agreement factory creates new agreement instances. These agreement instances contain data and functionality. An agreement instance implements a method to terminate the agreement if permitted. The properties of an agreement instance are exposed as WSRF resource properties. The standard agreement resource properties are defined in the agreement port type. They are depicted in Figure 32 as static properties. These static properties are specified in an agreement offer and do not change once the agreement is created.

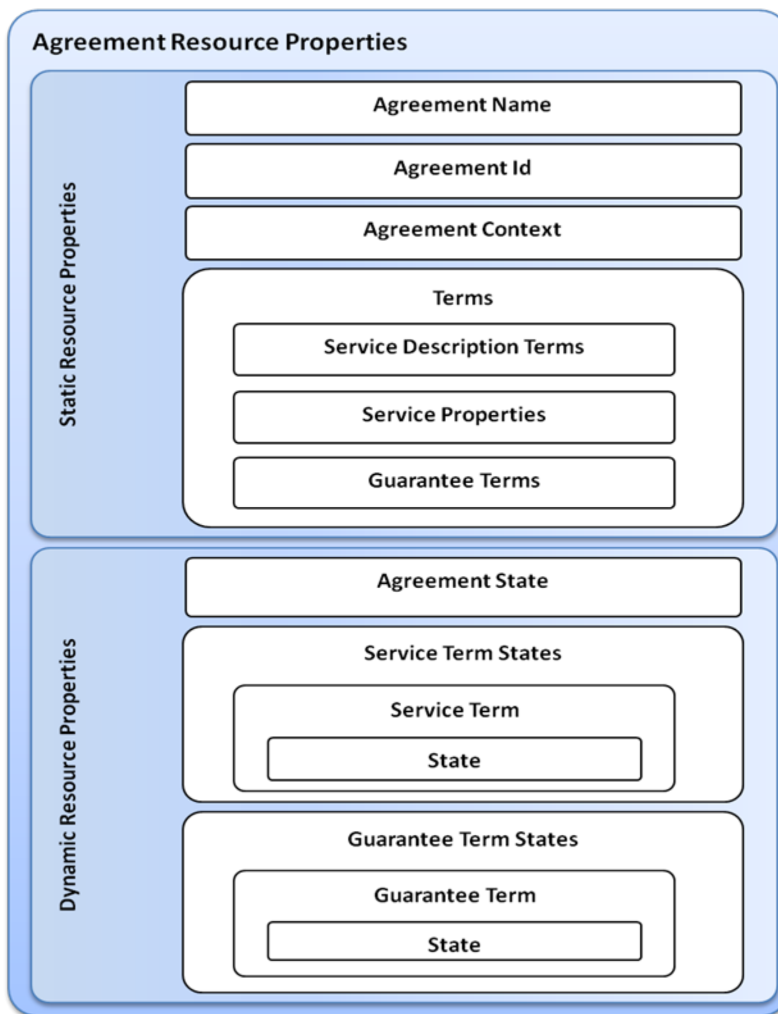


Figure 32: resource properties of an agreement instance

The agreement resource properties are defined in the WS-Agreement schema as separate XML document elements. Each agreement property is uniquely identifies by a qualified name (QName). In the WSAG4J framework each agreement instance stores the agreement properties in a separate resource properties document. External entities can query the agreement properties via the WSRF GetRe-sourceProperty method. This method takes the QName of the requested property as an input parameter. The WSAG4J web service module selects the according element from an agreement resource properties document and returns the result as an XML document to the requesting entity. Querying resource properties of an agreement therefore does not imply additional processing. This is important in order to ensure that accessing the agreement properties scales in terms of parallel requests. It is also of special importance for the monitoring of agreements.

Agreement monitoring is a continuous process that must be done as long as the agreement is active. An agreement is considered to be active as long it is in the *Observed* state. Agreement monitoring comprises the monitoring of the provided services and their associated service levels, the evaluation of the guarantees of an agreement, and the accounting of penalties and rewards, if applicable. WS-Agreement therefore defines a set of additional resource properties that are used to monitor the status of agreements at runtime. These resource properties are *Agreement State, Service Term States,* and the *Guarantee Term States.* They are stored as additional elements in the agreement resource properties document. The state properties are dynamic properties that change during the lifetime of an agreement.

Since the dynamic properties are stored in the agreement's resource properties document it is possible to access these properties in a scalable way. Moreover, the process of producing the dynamic properties is decoupled from accessing them. This is especially important, since the

production of these dynamic properties can potentially be a long lasting, time consuming process. As an example, consider a service consumer who creates an agreement with a service provider in order to execute a set of jobs. The service provider accepts the agreement and submits the jobs to its resources. The service provider then exposes the progress of each job via the appropriate service term state of the agreement. Therefore, the job execution systems notify an agreement monitoring service as soon as the state of a job changes. The agreement monitor service updates the agreement resource properties with the appropriate state information for each computational job. On the one hand, this monitoring strategy ensures that an agreement resource property is always up to date. On the other hand the production of agreement monitoring data is decoupled from accessing this data. This ensures that accessing agreement properties scales in terms of parallel requests.

In contrast to the given example, an agreement implementation may query the state of a job online when a dynamic resource property is accessed. This strategy would perform much worse, since for each request for a dynamic resource property would imply that a backend system is invoked.

3.5.2 Monitoring of the Agreement States

The last section illustrated why it is important to decouple the agreement monitoring process from accessing the agreement state properties. This section will now focus on how exactly the dynamic properties of an agreement instance are updated in the WSAG4J framework. In WS-Agreement, three types of dynamic resource properties are defined: (1) the agreement state, (2) the service term states, and (3) the guarantee term states.

Agreement State

The agreement state property represents the overall state of an agreement. The agreement can have one of the following states: *Pending, Observed, Rejected, Complete, Terminated, PendingAndTerminating,* and *ObservedAndTerminating.* The *Pending* state and its sub-state refer to agreements where the agreement responder did not yet decided to accept or reject the agreement. Agreements in these states are not active yet, and therefore no monitoring of the agreements is required. The *Observed* state and its sub-state refer to agreements where the agreement responder already accepted the agreement. Agreements in these states are active and need to be monitored. Agreements in the *Rejected, Terminated,* or *Complete* states are not active anymore. These agreements are not monitored anymore. The WSAG4J framework changes the state of an active agreement automatically to *Complete* if all aspects of an agreement are completed, e.g. all service term states of the agreement have reached the state *Completed*.

Service Term States

A monitored agreement exposes a service term state for each service description term that is defined in the agreement terms. A service term state defines the domain-independent state of a specific service description term. It can be in one of the following states: *NotReady*, *Ready*, or *Completed*. If a service description term is in the state *NotReady* the associated service is not yet set up; if the state is *Ready* the described service is set up and ready to use; and if the service term state is Completed, the service provisioning has ended and the service cannot be

used anymore. The service term state property is only applicable if an agreement is either in the *Observed* or *ObservedAndTerminating* state, which means that the agreement is active and monitored. Besides the abstract, domain independent state, a service term state can also include domain specific state information. This domain specific state information comprises for example monitoring information of a service execution, such as the minimal, maximal, and average response time of a provided service. In the WSAG4J framework the service term states are produced by a set of *Monitoring Handlers* that are registered with a *Monitored Agreement* instance. The *Agreement Monitor*, which is part of each *Monitored Agreement* instance, invokes these handlers in pre-defined intervals, e.g. once a minute. The purpose of these handlers is to provide the domain specific logic to update the agreement service term states. Once the *Agreement Monitor* has invoked all *Monitoring Handlers*, it updates the agreement's resource properties document with the new service term states.

Guarantee Term States

An agreement instance also contains one guarantee term state for each guarantee in an agreement offer. The guarantee term states can be one of the following: *NotDetermined*, *Fulfilled*, or *Violated*. The initial state of a guarantee is *NotDetermined*. A guarantee remains in this state until an assessment on the fulfillment or violation of the guarantee can be made. When a guarantee can be evaluated, its state changes either to *Fulfilled* or *Violated*. Depending on the guarantee state, a penalty or a reward may be issued. Penalties and rewards are defined by the associated guarantee term in an agreement. In the WSAG4J framework the evaluation of the guarantee term states is done by the *Guarantee Evaluator* component. The *Guarantee Evaluator* is invoked by the *Agreement Monitor* after the service term states where updated. The *Guarantee Evaluator* processes the updated agreement resource properties document and computes the state of each guarantee term. Additionally, the *Guarantee Evaluator* issues the penalties and/or rewards that are defined for a guarantee. Therefore it is possible to configure an appropriate accounting system for the SLA framework.

3.5.2.1 The WSAG4J Agreement Monitoring Process

The WSAG4J framework implements a schedule based monitoring process for agreements. This monitoring process is implemented by the *Monitored Agreement*. A monitored agreement implements the basic functionality to update the agreement state, the service term states, and the guarantee term states. The monitoring process is organized in predefined intervals, the so called *Monitoring Intervals*. These intervals are defined by the monitoring schedule. The *Agreement Monitor*, which is part of a monitored agreement instance, initiates the monitoring cycles based on the monitoring schedule.

A monitoring cycle comprises the following activities:

- *Retrieve and update the state for each service term in the agreement*
- *Compute and update the Agreement State based on the Service Term States*
- *Compute and update the state of each Guarantee Term*

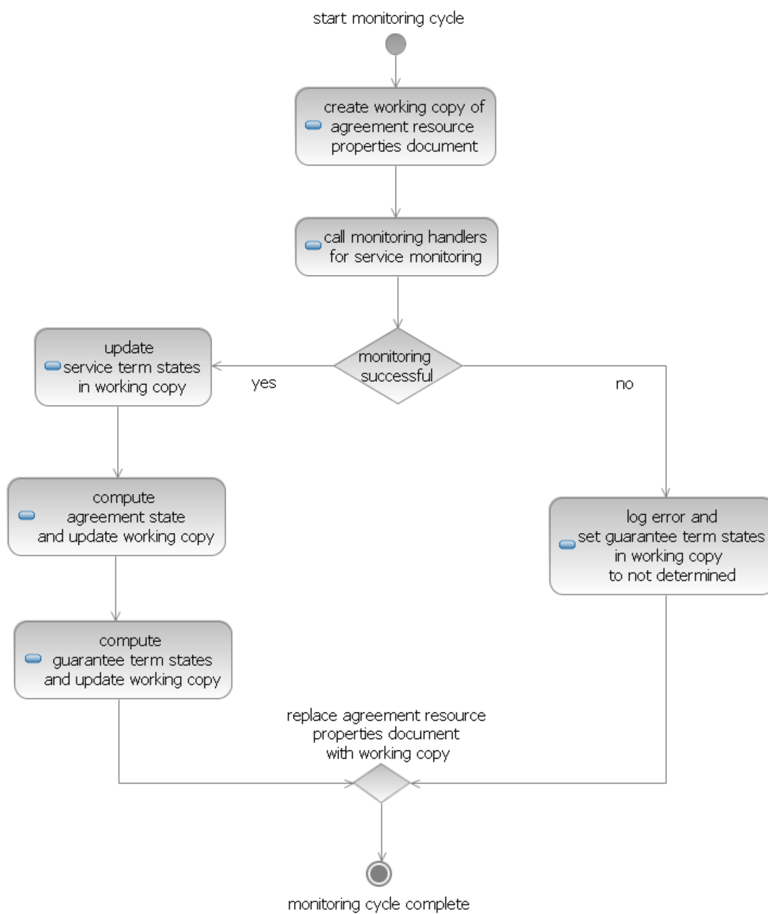


Figure 33: activities of a monitored agreement in a monitoring cycle

A monitored agreement is typically created in the context of a *CreateAgreementAction*. The CreateAgreementAction is responsible for initiating the service provision process for accepted agreement offers. Moreover, it returns an agreement instance that is capable of monitoring the agreed services and the associated guarantees. The Monitored Agreement of the WSAG4J framework is the default implementation of such an agreement. It can be configured with a termination handler that implements the business logic that is invoked when an agreement instance is terminated. A termination handler can for example shut down the provided service when the agreement is terminated. Moreover, the monitored agreement can be configured with a set of monitoring handlers. The monitoring handlers implement the business logic to retrieve the current state of the provided services and to update the agreement service term states accordingly. The monitoring handlers are invoked by the Agreement Monitor based on a monitoring schedule. The monitoring schedule can be configured for each monitored agreement instance individually by specifying a Cron expression. If the service term states were updated by the monitoring handlers successfully, the monitored agreement computes the overall state of the agreement and the state of the agreement guarantees. This process is depicted in Figure 33. It is important to note that all monitoring activities in a monitoring cycle are performed on a working copy of the agreement resource properties document. Therefore, when a monitoring cycle is started, a copy of the agreement resource properties document is created. This copy is then updated by the Agreement Monitor and the monitoring handlers. Once the monitoring cycle is completed, the agreement resource properties document is replaced with the updated copy. This ensures that external entities such as the agreement initiator can query the agreement properties even during a monitoring cycle and still have a consistent view on the agreement state, the service term states, and the associated guarantee term states.

3.5.2.2 Computation of Service Properties

In WS-Agreement the *Service Level Objective* of a guarantee is defined as an assertion over a set of measurable service properties. These measurable properties are the service properties.

They are defined in an agreement by the *Service Property* statement. This statement contains a set of variable definitions, where each variable has a name and a location. The variable name is the unique identifier of a service property. The variable location is a machine processable expression, e.g. an XPath expression. This expression is resolved during the guarantee processing to the value of the variable. Service properties are defined in the scope of a specific service. Therefore, different *Service Property* definitions can be provided for different services in the agreement.

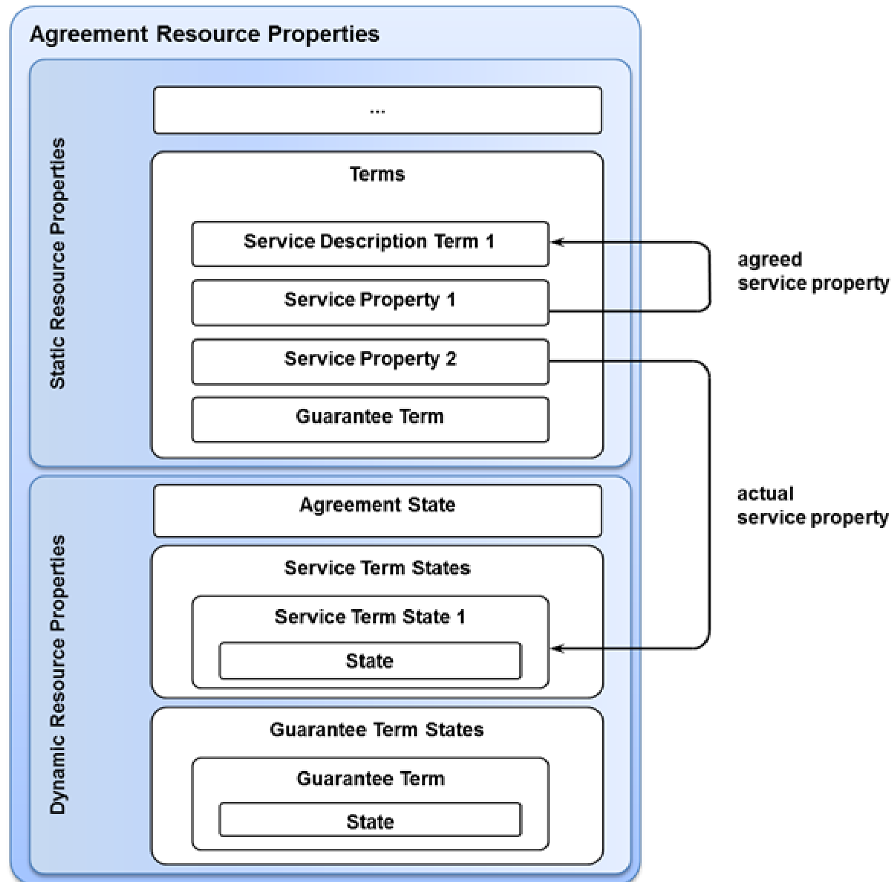


Figure 34: definition of agreement service properties

In general we can distinguish between those variables that define *Reference Values*, and those variables that define *Actual Values*. The reference value defines a measurable service level that should be achieved during provisioning, while the actual value is the currently achieved service level. Variables that define reference values are in the following called *Agreed Service Properties* and variables that define actual values are in the following called *Actual Service Properties*. Agreed service properties refer to values that are defined in the static agreement resource properties. These properties were defined in the agreement offer when the agreement was created. Actual service properties refer to values in the dynamic agreement resource properties, e.g. to a service term state. These properties are generated by the agreement monitoring and may include domain-specific monitoring data, e.g. the response time of an agreed service.

The computation of the service properties takes place after monitoring the agreement state and the service term states. Therefore, the agreement resource properties document is already

updated with the actual states. Now, the variables defined in the service properties can be resolved to their actual values within a monitoring cycle. Figure 34 illustrates this process. The variables can now be used to evaluate expressions in a guarantee term definition, e.g. the service level objective of a guarantee. These expressions can be computed automatically in order to determine the state of a guarantee. This is described in the next section.

3.5.2.3 Processing of Guarantee Term States

A guarantee term specifies a *Service Level Objective* (SLO). In short, the SLO defines an assertion that is defined over a set of service properties. WS-Agreement does not define an expression language for these assertions. The specification assumes that such expression languages will be defined outside of the WS-Agreement specification. The WSAG4J framework uses Java Expression Language (JEXL) that is implemented by the Apache JEXL framework [54]. JEXL already supports a wide set of expressions. This expression language is used in the WSAG4J framework to define service level objectives or qualifying conditions for guarantees in an agreement template. The following example illustrates the evaluation of a simple guarantee. Given that the service properties *Req_CPU_Speed* and *Act_CPU_Speed* are defined in an agreement, both properties are resolved by the WSAG4J *Guarantee Evaluator* before a guarantee is evaluated. In the following we assume that an agreement contains a guarantee term with the following *Service Level Objective*:

```
Act_CPU_Speed >= Req_CPU_Speed
```

The service level objective states that a guarantee is fulfilled when the CPU speed of the provided resource is at least as high as the requested CPU speed. The *Guarantee Evaluator* now populates the context of the JEXL interpreter with the names and values of the variables resolved during the processing of the service properties. Then it invokes the interpreter in order to evaluate the expression. The JEXL interpreter will evaluate the expression to a Boolean result. If a variable cannot be resolved, the interpreter throws an exception. Depending on the evaluation result, the *Guarantee Evaluator* either resolves the guarantee state to *Fulfilled* or *Violated*.

The same mechanism is applied for the evaluation of qualifying conditions. A qualifying condition can optionally be defined for a guarantee term in order to specify restriction under which a guarantee is evaluated. From a technical point of view the evaluation process of a qualifying condition and a service level objective are the same. If a qualifying condition is part of a guarantee and evaluates to *TRUE*, then the service level objective is evaluated and the guarantee term state is changed to *Fulfilled* or *Violated*. If a qualifying condition is part of a guarantee and evaluates to *False*, the service level objective is not evaluated and the guarantee term state is *NotDetermined*.

3.5.3 Guarantee Evaluation Example

The following example illustrates how a guarantee is evaluated by the WSAG4J framework. It defines a simple template for executing a computational job. The template contains an agreement context, the terms of the agreement, and a section with creation constraints. The context defines the metadata of the agreement. It specifies the parties of an agreement, the role of each party, and the name and the identifier of the template. The agreement terms

provide a default definition of the service that will be provided, a set of service properties, and one guarantee. The guarantee defines a reward and a penalty for meeting the defined service level objective, respectively for violating it. Finally, the template includes a set of creation constraints that restrict the structure and the content of valid agreement offers. In particular these creation constraints ensure that a customer can only change the number of resources that are provided, the number of CPUs of each individual resource, and a lower bound for the individual CPU speed. The contents of the service properties and guarantee terms are predefined in the template. The creation constraints prevent consumers to change these elements, since they are required to evaluate the fulfillment of the defined guarantees at runtime. The XML representation of the template used in this example can be found in the Appendix in section Agreement Template Example.

The service description of the computational service is realized by a JSDL job definition document. The job definition specifies the application to be executed and the resources that are available for the application. The required resources are specified in the resource section of the JSDL document. The service should be executed in an environment that comprises 16 nodes, where each node has 2 CPU's with a CPU speed of at least 2 GHz. Next, the template defines a set of service properties. The service properties are a set of variables that are used later on to define the *Service Level Objective* of a service guarantee. In order to reduce the complexity, the example template defines only three variables. The first variable resolves the current state of the service provisioning (*SERVICE_STATE*), the second variable resolves the agreed minimal CPU speed (*REQ_CPU_SPEED*), and the last variable resolves the CPU speed of the actually provided resources (*ACT_CPU_SPEED*). All variable definitions include a location element which specifies how variables values are resolved in the agreement resource properties document at agreement monitoring time. The example template uses XQuery [55] to resolve the variable values. XQuery is the XML query language supported by the WSAG4J framework for resolving service properties. It supports the definition of XML namespace prefixes as part of the XQuery expression, which is important when processing XML documents with fully qualified elements. Figure 35 shows the structure of the agreement template used in the example. Notice that only the *REQ_CPU_SPEED* variable can be resolved in the template. This is a user defined variable that is specified by setting the requested CPU speed in the service description. The *SERVICE_STATE* and *ACT_CPU_SPEED* variables can't be resolved in the agreement template, neither in an agreement offer. These variables refer to the service term state element of the computation service. The service term state element is part of the agreement's resource property document. It contains state information of the compute service at service provisioning time, for example the number of allocated resources for the service and the exact CPU speed of the allocated resources. This state information is produced by the agreement monitoring system once an agreement is in the *Observed* state.

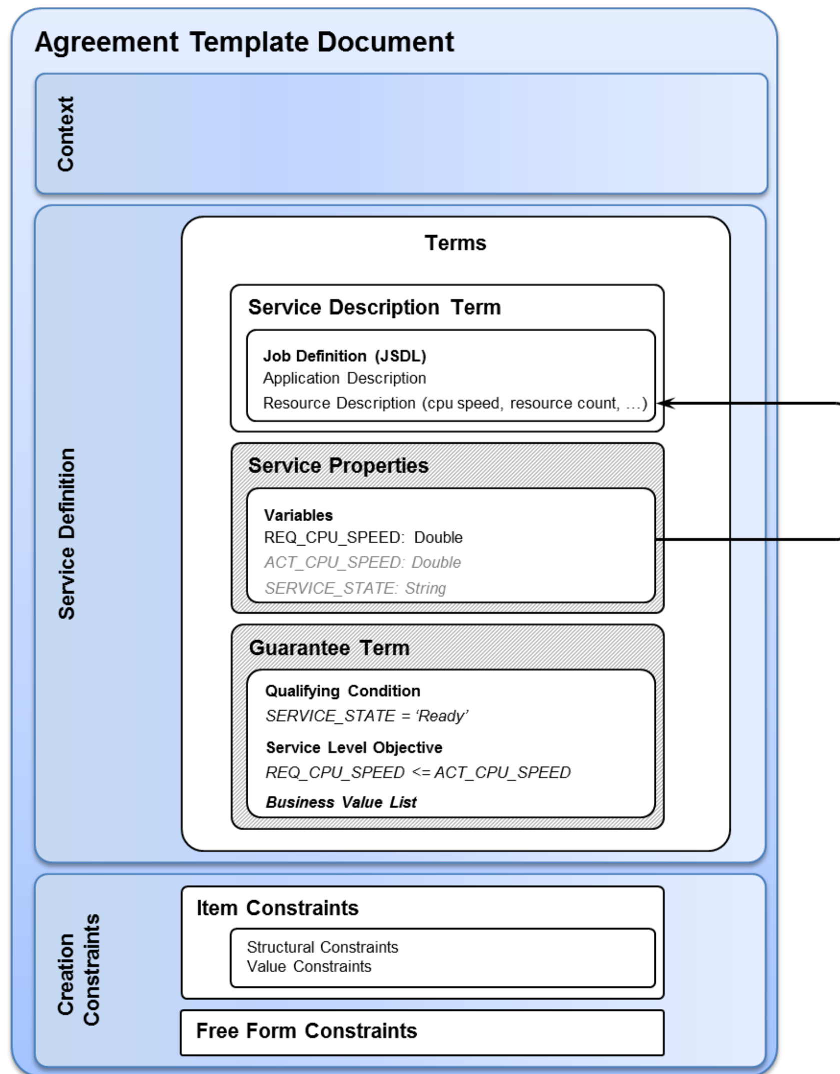


Figure 35: example template structure

The service properties represent a set of measurable properties that are used to define the guarantees of the agreement. In particular these variables are used in logical expressions to define the *Qualifying Condition* respectively the *Service Level Objective* of guarantees. The example template defines one guarantee with a qualifying condition and a service level objective. The qualifying condition must be fulfilled before the guarantee is evaluated. In the example, the qualifying condition is linked with the execution state of the provided service. Only if the computational service is in the state *Ready*, the guarantee is evaluated. If the service is not set up yet (*NotReady*) or already shut down (*Completed*) the guarantee is not evaluated. In case the qualifying condition is met and the service is active, the service level objective of the *CPU_SPEED_GUARANTEE* is evaluated. The guarantee itself defines a reward that is issued in case the service level objective is met, and a penalty in case the service level objective is violated.

Based on the example template, a new agreement is created. Due to the strict creation constraints only the number of resources, the individual CPU count, and the individual CPU speed can be changed by the agreement initiator. All other values must remain unchanged in a valid agreement offer. Each agreement instance has an associated resource properties

document that encapsulates the static agreement properties that are defined in the agreement offer, and the dynamic agreement properties that are generated by the agreement monitoring. WSAG4J generates one service term state for each service term defined in an agreement. In the example, one service term state is generated for the computational service.

Some of the service properties defined in an agreement template may refer to XML structures that are only available in the agreement resource properties document. In the example template, the *SERVICE_STATE* and *ACT_CPU_SPEED* variables are representatives for this kind of variable definitions. *SERVICE_STATE* resolves to the actual state of the computational service from the corresponding service term state. Another example is the *ACT_CPU_SPEED* variable. This variable refers to a domain specific extension in the corresponding service term state. The actual content of the service term state is retrieved from a service monitoring system at service provisioning time. The WSAG4J framework does this by calling a set of registered monitoring handlers. The monitoring handlers are integrated with the service provisioning system and implement the domain specific logic to retrieve the current state of the provided service. After retrieving the required information the monitoring handlers update the service term states in the agreement resource properties document. In the example the agreement resource properties document contains one service term state that corresponds to the computational service description term. The service term state contains one *State* element that exposes the overall state of the service and one JSDL job definition document that exposes the detailed state of the service. The job definition document contains a description of the computational service and a description of the actual allocated resources.

After the agreement resource properties document was populated with the actual service term states, the WSAG4J Guarantee Evaluator computes states of the guarantees. It therefore selects all guarantees defined in the agreement and iterates over the guarantee's service scope. For each service that is covered by a particular guarantee the evaluator resolves the corresponding service properties. Next, the evaluator resolves the value of each variable defined in the service properties. This is done by executing the XQuery expression specified for each variable on the updated agreement resource properties document. Next, the guarantee evaluator creates a new expression context and populates the context with the variable names and resolved values. Now, the qualifying condition of the guarantee is processed. If no qualifying condition has been defined for a guarantee, this step is skipped. Otherwise the evaluator creates a new JEXL expression based on the value of the qualifying condition. This expression is then evaluated with the before created expression context. The expression evaluation muist result in a Boolean value. In case the evaluation result is *TRUE*, the processing of the guarantee term state is continued, otherwise the guarantee term state is set to *NotDetermined*. If the qualifying condition is fulfilled, the evaluation of the service level objective is performed. This is essentially done in the same way as for the qualifying condition, but now the expression defined in the *CustomServiceLevel* of the service level objective is evaluated. If the expression evaluates to *TRUE*, the guarantee term state is set to *Fulfilled*, otherwise it is set to *Violated*. This completes the processing of a guarantee term state. The guarantee term states are computed each time at the end of each monitoring cycle. Figure 36 illustrates how static and dynamic resource properties are resolved in the described example. Note how the service property variables are used in order to define the qualifying

condition and service level objective in the guarantee term. An XML representation of the agreement properties document used in this example is provided in the Appendix in section Agreement Properties Document.

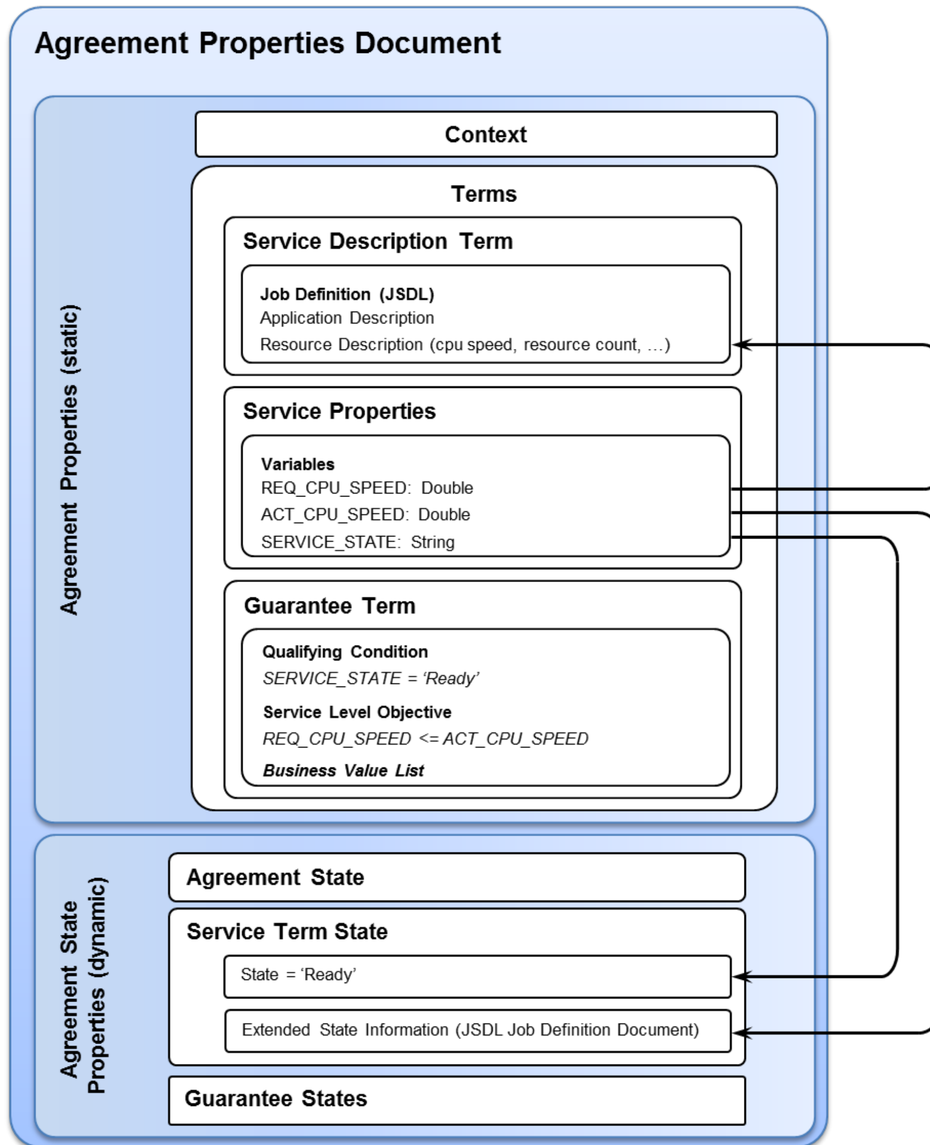


Figure 36: example of resolving static and dynamic service properties for guarantee evaluation

The computation of the guarantee term states does not include the evaluation of penalties and rewards. This is typically a separate process that is performed by the *Guarantee Monitor* after the computation of the guarantee term states. The *Guarantee Monitor* issues penalties and reward with an accounting system if a guarantee is fulfilled or violated in the assessment interval defined in the guarantee. Therefore, the *Guarantee Monitor* needs to keep track of all guarantee states of an agreement. Then, the *Guarantee Monitor* checks whether to issue penalties or rewards. This check is based for example on the assessment intervals defined in the penalty and reward sections of a guarantee. Assessment intervals define a time period for which a guarantee must hold (be violated) before a reward (penalty) is issued. Currently, the WSAG4J framework only contains a basic implementation of a Guarantee Monitor. This implementation is capable of handling assessment intervals that are equal to the monitoring

interval. A more generic implementation of the *Guarantee Monitor* that is capable of handling assessment intervals in a more flexible way can easily be added.

The preceding section described the SLA monitoring support of the WSAG4J framework. It was shown how service properties and guarantees are defined in WSAG4J, how service properties are monitored, how guarantees are evaluated based on these monitoring values, and how penalties and rewards are accounted. WSAG4J defines a programming model to add domain specific monitoring handler for retrieving the actual values of a service property. The guarantee evaluation in turn is implemented domain independently.

3.6 Summary

In this chapter we presented the WSAG4J system as a generic SLA management system. It was shown how the system can be used to provide SLA aware services in distributed environments. WSAG4J builds upon the WS-Agreement standard for creating and monitoring service level agreements. It provides a complete implementation of this standard. The framework has a modular design and can therefore easily be used with other applications, for example with different web service stacks. The framework provides a simple programming model for implementing WS-Agreement based services. Common tasks, such as service publishing, agreement offer validation, agreement creation, service monitoring, agreement monitoring, guarantees evaluation and the accounting are implemented in a standardized way and most of them are completely automated. The framework therefore reduces the complexity of the implementation of concrete SLA management systems. Moreover, the framework fosters the clear design of SLAs. For example, creation constraints are automatically enforced when they are contained in an agreement template. SLA management systems that use the framework can take advantage of the existing constraint enforcement mechanism. This also fosters the use of creation constraints. Accordingly, the guarantees of an agreement can be automatically evaluated by the framework. An agreement template must therefore follow a set of simple design rules described in this chapter. In general it can be stated that guarantees are assertions over measurable service properties. Additionally, guarantees may contain penalties and rewards, which are enforced by the WSAG4J engine during the agreement runtime. When an agreement is created, the agreement initiator and the agreement responder agree on the properties of the service that should be provided. During the service provisioning time, the properties of the actually delivered service are monitored. Based on these monitoring values the state of the agreement guarantees is assessed by the WSAG4J engine and the rewards and penalties are accounted. The conditions when a guarantee is fulfilled or violated are defined as part of the guarantee at the template design time. Again, this fosters the clear design of SLAs.

Finally, it can be stated that the WSAG4J framework is an essential building block for realizing SLA management systems in a distributed environment. It implements the required functionality to dynamically create service level agreements synchronously by using the AgreementFactory port type, or asynchronously by using the PendingAgreementFactory port type. WSAG4J-based systems can easily be deployed in service-oriented architectures and accessed remotely using the WS-Agreement protocol. Therefore, the framework seamlessly integrates into today’s Grid architectures.

Chapter 4
ORCHESTRATION OF SERVICE LEVEL AGREEMENTS

After the presentation of a generic SLA management framework in the Chapter 3, this chapter describes how service level agreements are used by an orchestration service in order to instrument resources in Grid environments. In section 4.1, related work in the area of resource orchestration in distributed systems is presented. Section 4.2 gives an overview of orchestration scenarios. These scenarios emphasize the motivation for an orchestration service and describe how infrastructure services can be coordinated in order to provide higher level services with added functionality. In section 4.3, we present a generic architecture for an orchestration service that uses SLAs for service planning and coordination. This generic architecture was implemented and validated in a concrete orchestration service during several German and European research projects. Section 4.4 gives an overview of these projects and describes the most important milestones. In section 4.5, we present the Phosphorus Orchestration Service. This service is a prototypical implementation of the orchestration service architecture presented here. Based on this implementation the orchestration service architecture is evaluated in section 4.6. The identified architectural issues were addressed in the enhanced version of the orchestration service architecture, which is presented in section 4.7.

4.1 Related Work

Grid schedulers and resource orchestration were investigated in research for several years. In [56] the authors present an overview of common Grid scheduling use cases, such as scheduling of complex workflow, component based applications, and co-allocation of resources, as well as projects that implemented concrete Grid resource management solutions. Moreover, the authors then identified a list of reoccurring patterns that can be found in Grid resource management and scheduling.

In [57] Schopf describes the basic actions of Grid scheduling. A Grid scheduling workflow in general consists of ten actions, which can be grouped in three phases; the resource discovery phase, the resource selection phase, and the job execution phase. Resource discovery includes the filtering of known resources according to the user's access rights and the minimal requirements of an application. During the system selection phase, dynamic information is gathered from the resources and based on that information a system is selected for execution. Finally, the job execution phase comprises actions such as advance reservation, job submission, preparation tasks, job monitoring, job completion, and cleanup tasks.

In [58] Foster et. al describe a generic architecture in order to support resource reservation and co-allocation for the Globus Toolkit. This architecture distinguishes between resource reservation and allocation and treats them as first class entities. They can be separately

created, managed, and monitored. Moreover, reserved resources can be allocated and used on demand, which provides applications with great flexibility in using these resources.

In the past a set of dedicated Grid scheduling solutions was developed. KOALA [59] is a Globus-based Grid scheduler that supports co-allocation of Grid resources, in particular co-allocation of computational and data resources. It implements different job placement policies for efficient scheduling of component-based applications, for example the close-to-file policy and the worst-fit policy. The close-to-file policy aims to minimize delays in job start-times by minimizing the file stage-in times for a job. The worst-fit strategy aims to balance idle processors in the Grid by allocating resources from the site with the largest number of idle processor. Moreover, KOALA supports the co-allocation of computational and data resources even in environments that do not support resource reservations. The scheduler therefore implements a resource claiming heuristic in order to maximize the success-rate of a co-allocation and to minimize the over-provisioning of computational resources. The KOALA architecture and experimental results are described in [60].

The GridWay Metascheduler [61] is a Globus-based workload management system that suppports job management and resource brokering on the Grid. It assists users in executing single Grid jobs, job arrays, or complex jobs. Job scheduling and execution is transparently done on behalf of the end user. During job execution GridWay dynamically adapts to changing conditions, for example resource failures or application performance degradation. GridWay therefore provides fault recovery mechanisms, dynamic scheduling, migration on-request and opportunistic migration [62]. GridWay implements a decentralized, modular architecture, which promotes loose coupling of applications and the underlying local management systems[63].

The above mentioned Grid scheduling solutions aim to support and improve Grid resource utilization, fair-share, or optimization of job runtimes. They address well-known Grid resource management problems such as the management of distributed data, heterogeneous resources, or the discovery and selection of adequate resources for Grid jobs. Traditional Grid scheduling solutions therefore complement the work presented in this thesis. They implement the methods required by computational resource providers to offer reliable, efficient, and computational services using Grid technology. Furthermore, Grid Scheduler and Resource Broker may also acquire resources from other resource providers using the SLA management technologies described in this thesis. Moreover, typical Grid resource management problems such as data management or resource co-allocation will still remain for SLA-aware resource orchestration services and the lessons learned from Grid Scheduling therefore provide valuable input these emerging services.

4.2 Orchestration Scenarios

Orchestration services are an important component in the architecture of a Grid system. They use and combine fundamental Grid services in order to provide higher level services with an additional functionality. Often they serve as a user entry point to a Grid. Orchestration services act as a facade [29] in order to hide the complexity of the Grid infrastructure from the end user and to ease the access to such an infrastructure. The following three examples describe orchestration scenarios that occur in Grid environments. In these scenarios the

orchestration service uses SLAs to allocate basic Grid services, for example computational or network services.

4.2.1 Co-allocation of Different Resource Types

A typical orchestration problem is the co-allocation problem that comprises the provisioning of different resources at the same time. This problem occurs in a wide set of resource provision scenarios, for example in the area of application service provisioning. A service provider offers an application to its customers as a service. This application simulates pollution transport in ground water. Multiple users can connect to the application service, start a simulation, display the simulation results at their workstations, and collaboratively analyze the results. The application is offered by the service provider with different service levels, for example a guaranteed frame rate of the visualization for 5, 10, or 20 users. The application itself consists of three modules. The first module is the simulation service. Users interact with the simulation service in order to provide input data for the simulation or to start and stop the simulation process. The second module is the collaboration service. Users connect to this service to collaboratively analyze the simulation results, for example to zoom into the model and discuss details. The third module, the visualization service, creates a video stream based on the user interactions that is then displayed at the work stations of the users. The different application modules communicate over a network connection. In order perform efficiently, the network connection must provide a certain quality of service, for example the latency between the different application modules must not be more than 20 ms and the guaranteed bandwidth must be at least 5 GBit/s.

When a customer creates a new SLA for this application, the service provider dynamically allocates the required resources from the different resource providers. It negotiates therefore with compute and network resource providers in order to find a common time at which the required resources are available. Once an applicable time frame has been found, the application service provider creates a SLA with each resource provider. The computational SLAs define for example a guaranteed uptime for the computational resources, while the network SLA defines the guaranteed latency and bandwidth for the network connection. As soon as the allocated resources are available, the application service provider dynamically deploys the application modules to the resources and starts the different services. Once all services are started successfully, the customers can use the application.

4.2.2 Load Balancing of Web-Applications

In another scenario an application service provider has developed a web-application for customer relationship management (CRM). The service provider offers its CRM application in a Software as a Service (SaaS) model. The CRM application is hosted by the service provider and customers can access the application online. The service provider defines a SLA for the application service provisioning. The SLA defines the average and maximum response times for the application at different service levels, for example an average response time of 25 ms for 10, 50, 100 or 1000 parallel users. The application service provider's core competence is the development of the CRM software. The service provider therefore allocates the required resource for the application service provisioning from a specialized computational resource provider. The resource allocation is backed up with a SLA. It is well

known to the application service provider that during the core working hours there is usually a high load on the CRM system, while during the night and on weekend the load is usually low. The SLA with the computational resource provider therefore defines two sets of resources, one for the normal working hours and one for the night time. As an example the SLA defines that 16 compute nodes must be available from Monday to Friday between 8 am and 6 pm, otherwise 4 compute nodes must be available. In order to provide the application with the defined service level, the application service provider must be capable to cope with performance peaks during day and night time. For that reason the application service provider has an orchestration service in place that monitors the current load of a CRM instance that reacts automatically on load changes. When the load for some reason exceeds the expected level, the orchestration service allocates additional resources from the computational resource provider by negotiating a new SLA. If this is not possible due to capability limits, the orchestration service allocates the required resources from a different resource provider. It then attaches the new resources dynamically to the CRM instance in order to meet the defined service level objectives. When the system load goes back to normal load, the orchestration service dynamically detaches additional resources from the CMR instance and terminates the respective resource SLA. The application service provider therefore is capable to dynamically handle performance peaks without over-provisioning of computational resources. It uses its resources more efficiently and therefore is more competitive on the market.

4.3 A Generic Orchestration Service Architecture

As mentioned before, orchestration services are important components of Grid infrastructures. They compose basic Grid services in order to provide higher-level services with enhanced functionality. Additionally, they dynamically negotiate, create and monitor SLAs for the basic Grid services in order to guarantee a service quality for the orchestrated services. SLA aware orchestration services therefore do not only support functional service properties, they also support guarantees for non-functional service properties. The orchestration service architecture presented here consists of three layers. The bottom layer is the service layer. At this layer the basic services are provided that are coordinated by the orchestration service. These basic services comprise for example the provisioning of computational resources, network resources, or licenses. The service providers implement a SLA layer in order to offer their services in a SLA-aware way. On top of the service layer sits the orchestration layer. This layer is responsible for planning and coordination of the services provided at the service layer and for the provisioning of the higher-level services. Orchestration services negotiate, create and monitor SLAs with providers at the service layer. The upper layer is the consumer layer. This layer comprises individuals such as scientists, or other services such as another orchestration service.

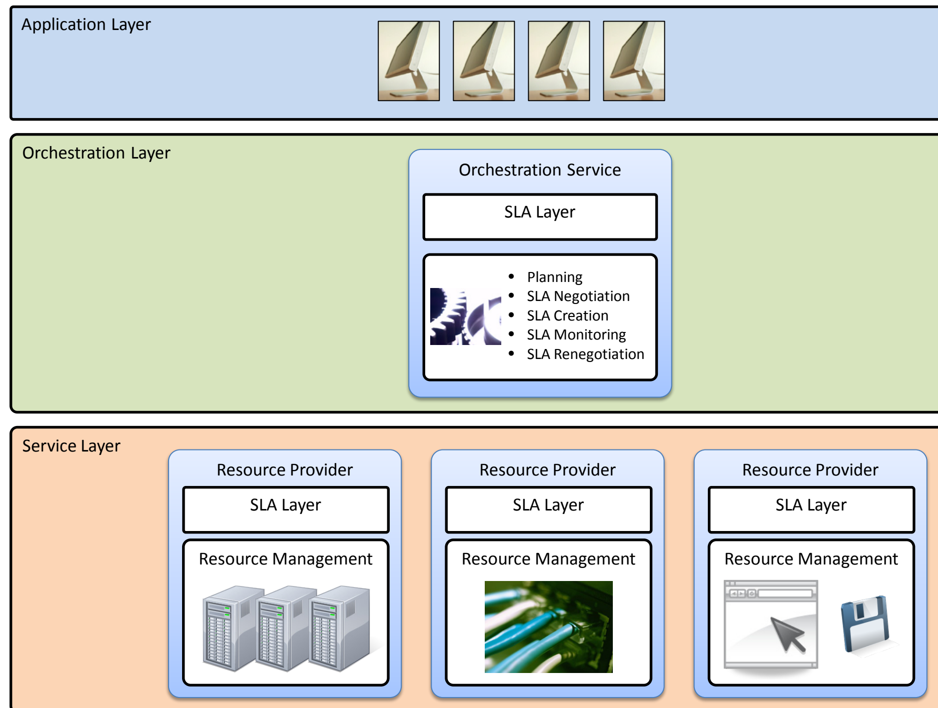


Figure 37: layered architecture of the orchestration service

Orchestration services can be used at different levels of distributed system architectures, i.e. in order to implement service provisioning paradigms such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS) or Software as a Service (SaaS). Depending on the application area, the orchestration services differ in functionality. In the following the IaaS, SaaS and PaaS provisioning paradigms are described in detail:

Infrastructure as a Service (IaaS)

Infrastructure as a Service describes a paradigm in resource provisioning where a company or an institution no longer buys the infrastructure it requires to run its application and services, but instead rents it from an IaaS provider. Typical representatives for this resource provisioning paradigm are providers of computational, storage or network capabilities. The IaaS paradigm comprises the outsourcing of IT infrastructure to dedicated service providers such as specialized data centers. IaaS providers offer their resources to their consumers as managed services with a defined quality of service, such as availability, failure recovery time, etc. Consumers can allocate resources on demand and therefore dynamically scale up and down their IT infrastructure depending on their requirements. Orchestration services can be used at IaaS layer in order to allocate resources on demand, to select resources automatically based on the consumer's requirements, or to increase the fault tolerance of the system.

Platform as a Service (PaaS)

The Platform as a Service paradigm does not only address the provisioning of a plain computing infrastructure, but also the provisioning of the required software stack for hosting the applications of a customer. The software stack can for example comprise application servers, database installations, etc. In Grid computing different software stacks are used, i.e. Globus [3] or Unicore [4]. Grid consumers should be able to allocate Grid platforms on

demand and use it as if it was a real one. Orchestration services can be used in PaaS scenarios i.e. to allocate the required resources (e.g. own resources or from an IaaS provider) and to deploy and configure the PaaS service (e.g. the Grid software stack).

Software as a Service (SaaS)

The *Software as a Service* paradigm describes scenarios where service providers offer complete software solutions to their customers. A typical example for SaaS is the provisioning of web-applications. These applications are often offered with well defined service levels, for example defined maximum and average response times for a specific number of users or with defined reaction times for resolving bugs in the software. In a SaaS scenario orchestration services are used for example to dynamically scale up or scale down the underlying computing infrastructure. This can be done based on the current and expected system load. The required resources are for example dynamically allocated from either a PaaS provider or an IaaS provider.

4.4 Evolution of the Orchestration Service

The orchestration service presented in this thesis was developed in a variety of German and European research projects. The initial version of the orchestration service was realized in the German VIOLA project [64]. This service was called VIOLA Meta-Scheduling Service (MSS). It was capable of co-allocating arbitrary types of resources [65] [66] in order to execute meta-computing applications on multiple Grid sites. The VIOLA MSS used SLAs to specify execution guarantees for meta-computing applications, for example the required resources per Grid site and network QoS between Grid sites. The VIOLA project started in June 2004 and ended in April 2007. The VIOLA MSS can be regarded as the ancestor of the WSAG4J framework. The initial version of the WSAG4J framework was published in 2006 as part of the VIOLA MSS. The MSS used the capabilities of the Grid middleware for resource co-allocation, job submission and management. It can therefore be regarded as a Grid scheduler that provides dynamic SLAs in a Software as a Service (SaaS) scenario.

In the year 2006 the IANOS project was initiated as a collaboration of four research institutions in Germany and Switzerland, namely the École Polytechnique Fédérale de Lausanne (EPFL), the Fraunhofer Institute SCAI (FhG SCAI), the Swiss National Supercomputing Center (CSCS), and the Technical University of Dortmund (TUD). The goal of the IANOS project was to integrate research work conducted by the EPFL in the area of application-oriented scheduling with the VIOLA MSS. The architecture of the MSS was therefore extended in order to automatically select the best-suited resources for an application execution. The MSS uses the IANOS Resource Broker for the resource selection. The IANOS Resource Broker identifies the best suited resources for an application from a domain based on data collected from historical application runs [67] [68]. For the resource selection the resource broker takes into account criteria such as CPU costs, license fees, or energy consumption [69] [70]. Moreover, the IANOS Resource Broker is capable of estimating the runtime of an application [71] on a concrete resource, based on the problem size that the application must solve. The IANOS system implements a user centric approach for Grid resource management that hides the complexity of the Grid from the user. The IANOS system

offers computational services in form of dynamic service level agreements. It can therefore be regarded as one particular implementation of a Software as a Service provider.

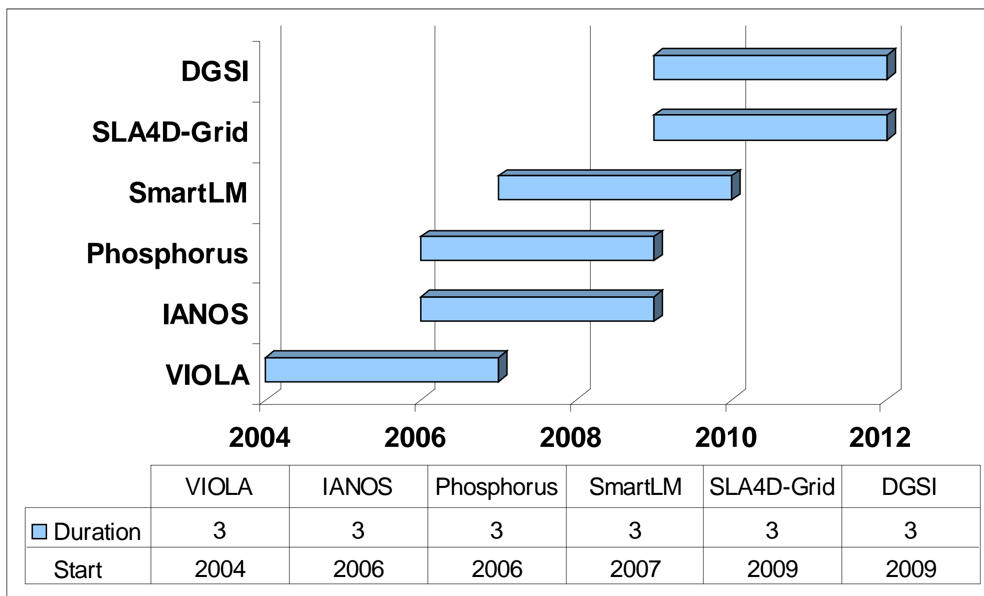
The research conducted in the VIOLA project was continued in the European Phosphorus project. The goal of the Phosphorus orchestration service was to support the execution of virtual screening workflows [72] by coordinating and allocating resources from different domains by using service level agreements. Similar to the VIOLA project, resources comprise computational and network resources. The orchestration service architecture anticipates service level agreements at different levels; 1) at the resource layer for the provisioning of computational and network resources, and 2) at the application layer for the provisioning of the virtual screening service. The WSAG4J framework was extended in this project in order to support SLA negotiation, dynamic agreement offer validation, service monitoring, and automatic guarantee evaluation. Additionally, the computational and network resource management systems were extended with an SLA layer. The SLA layers were used by the orchestration service to execute the virtual screening workflow. The Phosphorus orchestration service was successfully demonstrated at different occasions, for example during the Super Computing 2008 conference and the Terena Networking Conference 2009. The developments are further used in different German and European research projects, for example the European SmartLM project or the German SLA4D-Grid and DGSI projects.

The SmartLM project [73] is another European research project that adopted the WSAG4J framework and the orchestration service presented in this thesis. The goal of the SmartLM project is provide a Grid-friendly software licensing solution for location independent application execution. Service level agreements are used in SmartLM in order to create contracts on license usage and for dynamic license allocation and provisioning. The WSAG4J framework is used for the implementation of the SmartLM SLA service. Moreover, the Phosphorus orchestration service was adapted for the co-allocation of computational and license resources.

Currently, work is continued in the German SLA4D-Grid [74] and DGSI [75] projects. The goal of the SLA4D-Grid Project is to design and realize a Service Level Agreement layer for the Germany's national Grid infrastructure D-Grid. The SLA layer offers access to D-Grid resources under given guarantees, quality-of-service requirements and pre-defined business conditions. SLAs can be automatically negotiated, created, and their compliance monitored. The SLA4Grid project targets a productive Grid infrastructure. During this project the WSAG4J engine is integrated into the D-Grid middleware stacks, overcoming the limitations resulting from the overlay model implemented by the WSAG4U framework. Moreover, the fundamental SLAs for computational resource provisioning will be designed in this project. These SLAs will be supported in the different D-Grid middleware stack, fostering interoperability between these systems.

The goal of the DGSI project is to design and implement a SLA based Grid scheduling interoperability layer that enables users of one community to access and use resources from other D-Grid communities to perform their work. Existing scheduling solutions of the D-Grid communities should still be usable with the DGSI system. Furthermore, the SLA layer will define guarantees for the offered services in order to ensure the quality of a provided service,

for example in terms of guaranteed resources availability. The basic functionality of a SLA layer is already provided by the WSAG4J framework, current projects such as SLA4D-Grid and DGSI can concentrate on the definition of SLAs, comprising of service definitions, definitions of service properties and guarantees. Here, the advantages of a generic SLA layer as provided by the WSAG4J become obvious. Figure 38 shows an overview of the most important research projects and collaborations that influenced the work presented in this thesis.



	VIOLA	IANOS	Phosphorus	SmartLM	SLA4D-Grid	DGSI
Duration	3	3	3	3	3	3
Start	2004	2006	2006	2007	2009	2009

Figure 38: usage of the WSAG4J framework and the orchestration service in different research projects

The WSAG4J framework is considered as one major outcome of the work presented in this thesis. Besides the projects mentioned before, the WSAG4J framework was also successfully employed in a variety other research projects, for example in the European SLA@SOI project [76] [77], or in the German THESEUS project [78] [79] for the TEXO use case. Additionally, the framework was used in a different bachelor, master, and PhD thesis.

In the following sections, the Phosphorus Orchestration Service is described. This service is the second fundamental outcome of this thesis. It is a prototypical implementation of the generic orchestration service architecture. Based on this implementation the generic architecture is evaluated in order to identify its strengths and weaknesses. Afterwards, a revised architecture is presented that overcomes the identified weaknesses.

4.5 The Phosphorus Orchestration Service

Within the European Phosphorus project [6] an orchestration service for executing molecular docking applications on Grid resources was developed [80]. Docking applications are data-intensive applications with high requirements on computational resources and data management. The Phosphorus project itself was focused on the development of next generation networks that enable Grid applications to use distributed computational resources together with on-demand network services and dedicated end-to-end network QoS. The Phosphorus orchestration service is one component in the Phosphorus software architecture. It abstracts the complexity of the underlying infrastructure from users and enables them to utilize the available resources in an easy, seamless and secure way. Figure 39 shows the Phosphorus testbed with collaboration partners from the USA.

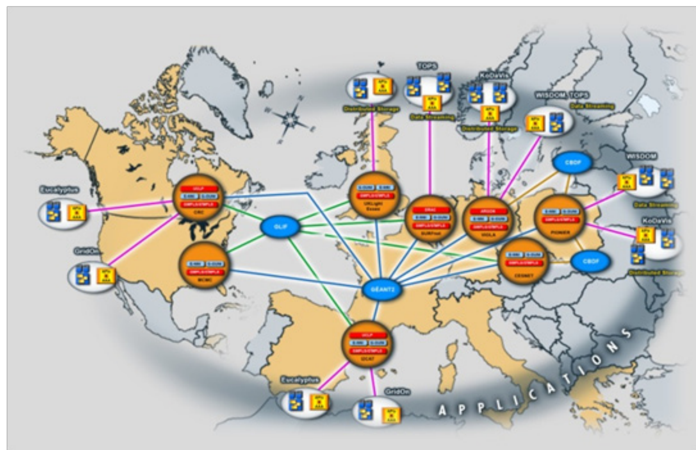


Figure 39: the Phosphorus testbed

The orchestration service used resources from different Phosphorus sites, for example from universities and research institutions in Germany, the U.K., and Poland. The following goals and requirements were defined for the orchestration service:

1. *Data Management:* Since the molecular docking application is a data-intensive application, efficient data management is required in order to efficiently stage-in the simulation input data and to stage-out the simulation results. The orchestration service should use the extended network management capabilities for the data management by allocating dedicated network QoS for the file transfers.

2. *Load Distribution:* The orchestration service must be capable of distributing a user's workload to the different resource providers in the Phosphorus testbed. The workload is distributed to the resource providers according to their capabilities to deliver the simulation results at the earliest possible date. Resource providers that have faster resources therefore get more load than those with slower machines.

3. *SLA-enabled Resource Provisioning:* Service Level Agreements are adopted as a basic resource provisioning paradigm for the resource providers and the orchestration service. The required resources for a job execution are allocated from different, independent domains (resource providers), where the resources of each domain are offered controlled by SLAs. The orchestration service must be capable to create SLAs with different resource providers and to control and monitor the resource allocations via the SLA layer.

4. *Failure Management:* Since resources in a Grid can fail, the orchestration service implements the basic functionality to cope with errors that may occur during the application execution, for example via job re-submission.

5. *Functional Evaluation of the SLA framework:* The WSAG4J framework is adopted for the realization of the different SLA layers. The framework is evaluated in order to verify functional completeness, applicability, composability, and standard conformance.

The following topics were considered to be outside of the scope of the Phosphorus Orchestration Service implementation:

1. *Implementation and Evaluation of Grid Scheduling Algorithms:* Different scheduling algorithms can be applied for example in order to minimize the turnaround time for a job, to maximize utilizations of the Grid systems, or to minimize the costs of a job. This thesis focuses on the generic architecture of orchestration services rather than on the implemented scheduling algorithms.

2. *Implementation and Evaluation of SLA-aware Resource Management Strategies:* Service Level Agreements comprise the provisioning of services with a guaranteed quality of service. Concrete service provisioning implementations can comprise dynamic resource management strategies in order to provide a guaranteed quality of service. A lot of work has been done already in this area, for example in the field of SLA-aware computational service provisioning. Therefore, this topic is out of scope of this thesis.

3. *Definition of Guarantees and Key Performance Indicators (KPI):* The orchestration service presented here is a proof of concept implementation and research prototype. Its goal is to validate the feasibility of SLAs for dynamic, automated resource orchestration. This prototype should help to identify the limitations of the current orchestration service architecture and provide the required input in order to overcome these limitations. Guarantees and KPIs are concepts that can easily be added to existing SLA layers. Therefore, this topic is out of scope of this thesis.

The implementation of the orchestration service architecture requires SLA-aware resource provisioning systems. Since today's Grid middleware systems do not support service level agreements out of the box, an appropriate SLA layer must be implemented for this purpose. This is addressed in section 5.3.1. The architecture and implementation of the orchestration service itself is discussed in section 5.3.2.

4.5.1 Resource Layer Architecture

The Phosphorus Orchestration Service is a concrete implementation of the orchestration service architecture. The orchestration service executes applications in a SaaS scenario. It allocates the required resources for the application by dynamically creating service level agreements with different resource providers. The resource providers offer their resources in an IaaS scenario. In order to dynamically create, monitor, and orchestrate SLAs, the computational and network resource management systems must be integrated with a SLA layer. The WSAG4J framework is used as a base implementation for the SLA layer. The integration of the resource management systems with the SLA layer comprises the following points:

1. Integration of the resource management system with the SLA layer
2. Design of relevant SLAs for the different resources (computational and network)
3. Implementation of appropriate SLA negotiation strategies for co-allocation scenarios

Before the SLA layer can be integrated with the underlying resource management systems, the decision for the integration model must be made. In general two integration models could be applied, the *integrated model* or the *overlay model*. Both models have different advantages and disadvantages.

Overlay Model

The overlay model constitutes a loosely coupled integration with the resource management layer. The overlay model uses the public interfaces (remote interfaces) to access the functionality of the resource management layer. In distributed systems, these public interfaces are often realized in the form of web services. Dedicated client implementations are usually available in order to access the resource management layer. The main advantage of the loosely coupled integration is the technology independence of the integrated systems.

Integrated Model

The integrated model comprises the native integration of the SLA layer into the resource management layer. This means that the SLA layer is directly integrated into the resource management software stack and therefore becomes an integrated part of the resource management system. Consequently, the integrated model constitutes a tight coupling of the resource management and the SLA layer.

Comparison of Overlay and Integrated Model

The decision on the integration model for the resource management layer is a vital architectural decision. Depending on the system requirement, one or the other integration approach is more suitable. The major advantages and disadvantages of the two integration approaches are listed in Table 1.

Advantages	Disadvantages
- Loosely coupled - Low integration effort - High flexibility due to technology independence to the target system - Systems can evolve independently (technology and architecture) - Low risk of change for new released of the target system, since public APIs can usually be considered stable	- Only functionality offered via public APIs can be used - Specific functionality (e.g. security) must potentially be re-implemented - Degraded performance due to remote access - Medium to high administration effort, since an additional system has to be maintained

Table 1: advantages and disadvantages of the overlay model

Advantages	Disadvantages
- Tightly coupled - Extended functionality by using internal APIs - Takes direct advantage of the systems base functionality and future extensions - Better performance due to tight integration - Low administration effort, since only one system has to be maintained	- Potentially high integration effort - Integration is coupled to one specific system, additional systems require separate integrations - Prone to internal API changes - Technological dependencies to the implementation of the target system must be resolved

Table 2: advantages and disadvantages of the integrated model

In the Phosphorus project, the overlay model was chosen as the formal integration approach. This decision was made for the following reasons:

1. *Integration of multiple resource management systems:* The integration effort had to be limited to an acceptable extend. The native integration of the SLA layer was therefore out of scope.

2. *Architectural and technological independence:* The used resource management systems, respectively the used Grid systems, were implemented using different architectures and technologies. Since the overlay approach promotes loose coupling of the systems, the same architecture can be applied for each integration scenario. Moreover, the technologically inevitable overlap between the systems and the involved technological dependencies are limited. This is due to the fact that only lightweight API clients are used to access the functionality of the Grid systems.

3. *Independence of development cycles:* The underlying Grid systems used in the project were at different stages of development. While some systems were quite mature with well-defined release cycles, other systems were still in early development phases and therefore likely to change.

4. *Performance:* Even though there is performance degradation due to the loose coupling of the systems and the implied remote communication, this issue was not regarded as critical since it can be solved in a later stage by migrating to the integrated model.

Even though the overlay model was chosen as the architectural pattern for the system integration, it is possible to migrate the SLA layer implementations to the integrated model at a later time.

4.5.1.1 SLA Layer for Compute Resources

Grid systems, such as Globus [3], Unicore [4], or gLite [81], provide the software stacks for building complex Grid systems. They offer secure and seamless access to computing resources, including file transfers, job submission and monitoring, and they can be used with a wide range of local resource management systems, for example Torque [82], Platform LSF [83], Oracle Grid Engine [84], etc. In the Phosphorus project the Unicore 6 system was used as a reference implementation of a Grid system. Since Unicore does not support service level agreements by default, the WS-Agreement for Unicore 6 system (WSAG4U) [85] was

developed. WSAG4U is a generic SLA layer for Unicore 6 based on the WSAG4J framework. The fundamental design goals were:

1. Implementation of a WS-Agreement based SLA layer for computational resources
2. To support reservation of computational resources
3. To support negotiation of execution times for resource reservations
4. Computational services are controlled by a SLA and are provided autonomously
5. Integration into Unicore 6 system using standard mechanisms

WSAG4U is a concrete implementation of a WS-Agreement based SLA layer for Unicore 6. Each Unicore 6 site is encapsulated by an Agreement Factory. A Unicore 6 Agreement Factory exposes a set of SLAs to reserve resources and to execute jobs at this Grid site. Each SLA is defined by an agreement template and each template is associated with a domain specific implementation to instantiate and to monitor the services for this SLA. The domain specific implementation uses the Unicore Client API in order to access the Unicore Atomic Services. Figure 40 shows an overview of the basic WSAG4U architecture.

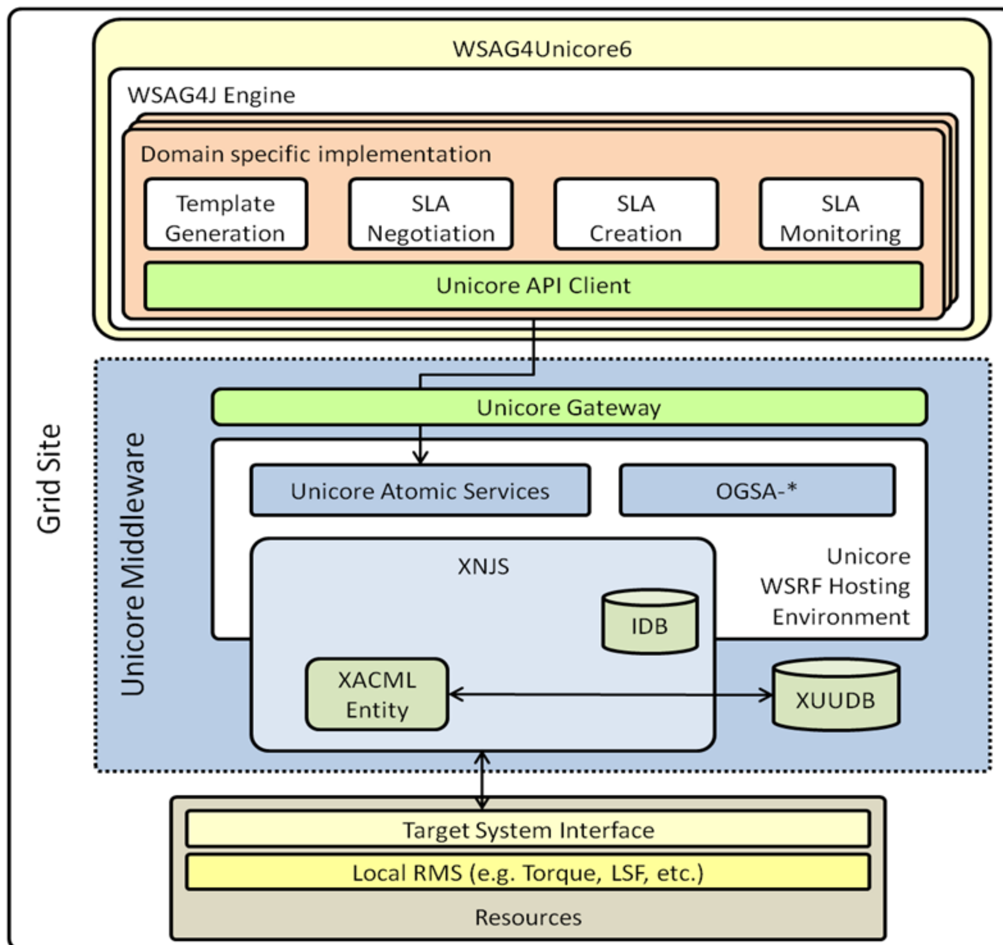


Figure 40: WS-Agreement for Unicore 6 (WSAG4U) architecture

A WSAG4U instance can be configured with a Unicore 6 service registry. The system instantiates a new WSAG4U Agreement Factory for each Unicore 6 site registered within that service registry. WSAG4U can therefore easily be used as a SLA-based integration platform for existing Unicore 6 installations.

In the following, a short overview of the SLAs supported by the WSAG4U system is given.

Computing SLA

The *Computing SLA* encapsulates the submission of a computational job to a Unicore 6 site. It specifies the application to be executed and the required resources. The application and the resources are described using the Job Submission Description Language (JSDL). The agreement template contains a list of application service description terms. Each service description term (SDT) represents one application that is installed at the Unicore 6 site. The agreement initiator must choose one application from the template for execution. The application SDTs are therefore structured in an ExactlyOne tag. Moreover, the template contains one resource SDT. This SDT represents the resources that will be allocated for the application. The application is executed autonomously by the Grid site. As soon as the SLA is created, an according job is submitted to the Unicore 6 system. The input files and output files for this computational job are specified in the agreement offer as part of the application service description. The service consumer must transfer the input files to the service provider before the SLA is created. After the computational job is finished the consumer can retrieve the results. The consumer can monitor the progress of the job at the SLA layer.

Advance Reservation Computing SLA

The *Advance Reservation Computing SLA* has a similar structure as the *Computing SLA*. Additionally, a consumer can specify a start time and an end time for the computational job. The time constraint is specified in a separate service description term. The definition of the time constraint follows the *Time Constraints Profile* [86] defined by the GRAAP group of the OGF. The Advance Reservation Computing SLA is not a reservation by itself. It is rather a computational job with a guaranteed start time. This specific SLA directly couples a reservation of computational resources with a computational job (the selected application). The computational service (the execution of the computational job) can

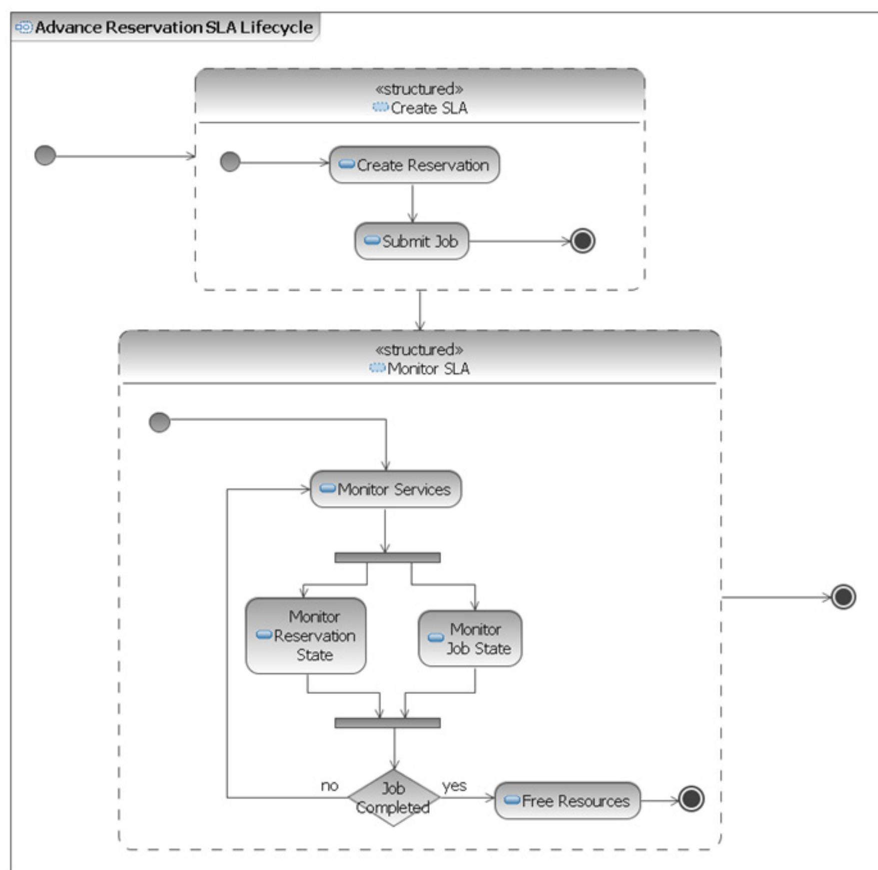


Figure 41: detailed lifecycle of an advance reservation computing SLA

therefore be provided autonomously without user interaction. The detailed lifecycle of an Advance Reservation Computing SLA is depicted in Figure 41. When a new SLA is created, the computational resources are reserved for the specified time frame. If the reservation succeeds, the specified application is submitted as a Grid job for the reserved resources. As soon as the reserved resources are available, the specified application is started on these resources. When the application execution is finished, the reserved resources are de-allocated. The state of the SLA changes to *Completed*.

StageIn and StageOut SLAs

The stage-in and stage-out SLAs provide the required capabilities to copy data to a Grid site (stage-in) or retrieve data from a Grid site (stage-out). The file transfers are done on a best effort base. The data is stored on a shared storage, which is accessible within the whole Unicore site. The description of the file staging service was realized by using the appropriate JSDL data staging elements. The file source in a stage SLA must point to a valid file endpoint of a Unicore installation. The same applies for the target endpoint in a stage-out SLA.

Service Monitoring

WS-Agreement supports service monitoring via the Service Term State properties of an agreement. By default these service term states contain only limited information related to the service provisioning lifecycle, namely the service lifecycle states *NotReady*, *Ready*, or *Complete*. In order to expose the outcome of a Grid job this information is insufficient. The WSAG4U system therefore includes an extra execution state document within the service term states. The execution state document specifies the state of the Unicore 6 job that was instantiated for the agreement. Additionally, the execution state document contains a JSDL job definition document that represents the Unicore 6 job. The following example shows the service term state for the application service description term of a *Compute SLA* respectively of an *Advance Reservation Compute SLA*.

```
<wsag:ServiceTermState wsag:termName="APPLICATION_SDT" xmlns:wsag="...">
  <wsag:State>Complete</wsag:State>
  <wsag4u:ExecutionState>
    <wsag4u:JobExecutionState>SUCCESSFUL</wsag4u:JobExecutionState>
    <jsdl:JobDefinition xmlns:jsdl="...">
      ...
    </jsdl:JobDefinition>
  </wsag4u:ExecutionState>
</wsag:ServiceTermState>
```

The *Compute SLA* and *Advance Reservation Compute SLA* additionally expose a resource service term state (RESOURCE_SDT state). This state contains a JSDL document that denotes the actually allocated resources for the computational job, for example it contains the

IP addresses of the allocated compute nodes in the *Candidate Host* section of the JSDL document.

4.5.1.2 SLA Layer for Network Resources

During the Phosphorus project a network resource management system called Harmony [87] was developed. Harmony is capable of allocating network resources over multiple network domains. Network capabilities can either be reserved in advance or allocated on demand. The Harmony system provides a web service based interface in order to access the system. It also provides an API client to easily integrate Harmony's network management capabilities into other systems. The API client also implements fundamental security functions, such as encryption and digital signatures. The architecture for the Network SLA layer follows the same architectural pattern as the SLA layer for computational resources. Once again, the overlay model was adopted as the architectural pattern for the SLA layer.

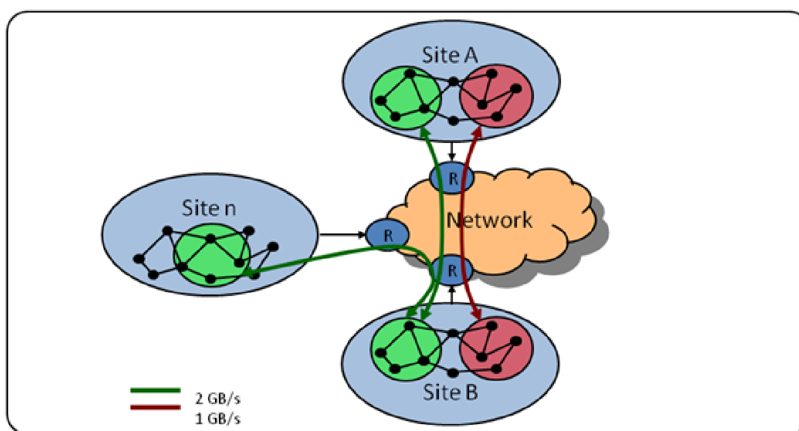


Figure 42: network reservation with the Harmony system

Figure 42 shows a simple network reservation scenario. Multiple Grid sites are inter-connected by a managed network, e.g. the Phosphorus testbed. Each site comprises a set of computational resources, which are connected to the managed network via a network router. These routers are the reservation endpoints of the network. A network management system is capable of reserving network capabilities between these reservation endpoints in order to guarantee the available bandwidth for an inter-site communication, for example for a file transfer from Site A to Site B. When a network reservation is made, the reservation endpoints and the required network QoS must be specified. Since multiple network reservations between two sites can exist in parallel, the network management must assure that the reserved quality of service is provided for the different sets of communication endpoints. Communication endpoints are in contrast to reservation endpoints the real endpoints of a network communication. Figure 42 illustrates this problem. Here, two jobs with different QoS requirements communicate in parallel over a wide area network. For that reason a network reservation R_1 is made for job 1, which reserves a bandwidth of 2 GBit/s. A second network reservation R_2 is made for job 2, which reserves a bandwidth of 1 GBit/s. For each job a set of computational resources (nodes) is allocated on Site A and Site B. These resources are the communication endpoints. The resources N_{A1} (Site A) and N_{B1} (Site B) are allocated for job 1, and the resources N_{A2} and N_{B2} are allocated for job 2 respectively. When the network reservation R_1 is active the network management must make sure that data traffic between the communication endpoints N_{A1} and N_{B1} is transferred according to the network QoS specified in R_1. The same applies for data traffic between the communication endpoints N_{A2} and N_{B2}.

Techniques such as VLANs (virtual local area networks) and traffic shaping are used to ensure that the reserved network QoS is supplied for each job. Therefore, the IP addresses of the communication endpoints that are associated with a reservation (N_{A1} and N_{B1} for R_1 respectively N_{A2} and N_{B2} for R_2) must be known to the network management system at job execution time. This information can be provided at reservation time or later.

The binding of the communication endpoint IP addresses to network reservations is essential for the reservation of network resources. However, this information is not always available during reservation time. Depending on the resource management system used by the computational resource providers, the IP addresses of the reserved resources may only be available once the resources are finally allocated. This requires that a late binding of the communication endpoint IPs to a network reservation must be possible. This in turn has an impact on the SLA design. Besides the description of the network service and the time constraints for the reservation, the network SLA must also contain a reference to the computational SLAs that serve as communication endpoints. As described before, the computational agreements expose the IP addresses of the allocated resources as part of their service term states. The network SLA implementation retrieves these IPs as soon they are available and binds them to the network reservation. The Phosphorus network SLA uses a domain specific language to describe the network service. It is expected that standardized languages for network services will be specified in the future, for example by the Network Mark-up Language Working Group [88] of the OGF. The time constrain service description term is defined according to the Time Constraint Profile [86].

4.5.2 Implementation of the Orchestration Service

As discussed before, the orchestration service was designed to execute data-intensive applications in a Software as a Service (SaaS) scenario. It uses the capabilities of the Infrastructure as a Service (IaaS) layer described before to distribute the workload of the docking application to available Grid resources. The sequence diagram depicted in Figure 43 shows the workflow implemented by the orchestration service. In a first step, the service consumer transfers the simulation input data to the orchestration service. The orchestration service therefore provides an interface to a storage service. Such a storage service is for example implemented by a Grid system like Globus or Unicore. Then, the service consumer creates a SLA with the Orchestration service in order to execute the docking application. In the next step, a resource discovery phase is performed. This phase comprises the required actions to find the resources that are suitable to execute the application, which includes the following steps:

- Resource Provider Discovery
- Authorization Filtering
- Minimal Requirements Filtering

The resource provider discovery is done by using a central registry. This registry contains the Agreement Factory endpoint of each known resource provider. In the next step authorization filtering is performed. This process identifies the resource providers that accept computational jobs from a given user. Checking the authorization of a user at a resource provider can either be done explicitly or implicitly. An orchestration service can for example explicitly ask a

resource provider if a given user is allowed to access the provider's resources, e.g. by calling a domain specific service. A different approach is the utilization of *trust delegations*, for example proxy certificates [89]. In this approach the orchestration service queries the available SLA templates from a resource provider behalf of a user. If the user is authorized to access resources from this particular resource provider the request succeeds, otherwise it fails. The outcome of the authorization filtering phase is a list of resource providers that accept jobs for the given user.

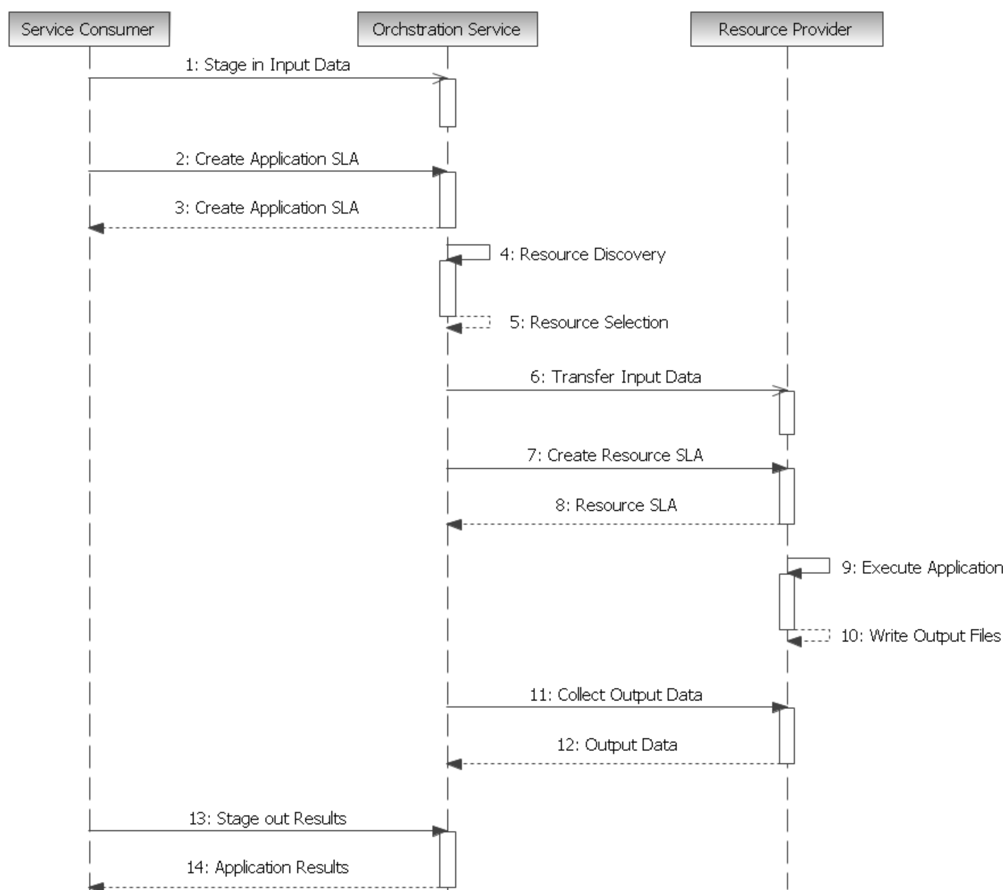


Figure 43: orchestration service workflow

Based on this list the minimal requirements filtering phase is performed. During this phase the orchestration service checks whether a resource provider is capable of executing the molecular docking application, e.g. verifying that the required software is installed. The orchestration service uses the *Compute SLA* to execute the application. It therefore queries the agreement templates from the selected resource providers. If the *Compute SLA* template contains an application entry for the molecular docking application, the application is supported by the resource provider. The outcome of the minimal requirements phase is a list of resource providers that accept computational jobs issued by a given user and that provide the user's application.

After the resource discovery phase, an execution plan for the application is generated, which comprises the calculation of the workload distribution. The overall workload is therefore divided into smaller units of work, which can be processed independently. One unit of work is

a standardized workload for all Grid resource providers, e.g. a fixed number of docking processes. A reasonable size for one unit of work is determined beforehand by consulting domain experts. Furthermore, the application entry in the SLA template is annotated with an execution time. This execution time specifies the required time for processing one unit of work on one compute node. The execution time is empirically determined by domain experts in advance or dynamically calculated based on historical data (e.g. monitoring data of previous application runs) [90]. Based on the estimated execution times and the available resources at the different sites the execution plan is generated. In the next step the input files are transferred to the selected computational resource providers. The network reservation capabilities are used in order to reserve bandwidth for the file transfers. Now, the computational jobs are submitted to the resource providers. The orchestration service therefore creates a set of SLAs with the computational resource providers according to the execution plan. Each SLA specifies the workload to be processed, the required resources, and the required time. Then the orchestration service monitors the SLAs and once they are completed, the simulation results are collected. This is again done by using the network reservation capabilities. This completes the service provisioning process. The consumer can stage out the collected simulation results from the orchestration service.

The Phosphorus orchestration service uses *Compute SLAs* to execute the application (see section 5.3.1.1). The *Compute SLA* executes an application on best effort base. It is therefore not possible to guarantee a completion time for the overall workflow. If the workflow should be completely planned in advance (e.g. to guarantee a completion time) the *Advance Reservation Compute SLAs* must be used instead. However, these advance reservation jobs don't take advantage of local backfilling. This limitation can be overcome by specifying deadlines rather than start and end times for computational SLAs. This gives a resource provider the freedom to use its resources in a more flexible way. At the same time the service consumer may benefit due to faster job completion times.

A similar backfilling problem arises for the stage-out process. When the processing of the application workload completed early, the stage-out process still takes place at the initially planned time. Since file transfers with guaranteed network QoS comprise resources allocated from multiple domains (computational and network resources), this problem must be handled by the orchestration service. The orchestration service can for example try to renegotiate the co-allocated SLAs for file transfer and therefore achieve backfilling functionality at the orchestration layer.

4.5.3 Co-Allocation of Computational and Network Resources

The Phosphorus orchestration service uses the capabilities of the network SLA layer described in section 5.3.1.2 in order to reserve network resources and realize file transfers with guaranteed network QoS. The network SLA layer provides the functionality to dynamically negotiate and allocate guaranteed network QoS between two communication endpoints. As described before a communication endpoint is a set of computational resources attached to the managed network. File transfers with dedicated network QoS differ from "*normal*" Grid file transfers, such as Globus Grid FTP or Unicore BFT. Normal Grid file transfers are conducted between the frontend nodes of a Grid system. Grid frontend nodes are

shared nodes, which can execute multiple file transfers for different users in parallel. This makes it hard to ensure that the reserved network QoS is provided for the file transfers initiated by a specific user. In order to overcome this issue, file transfers with guaranteed network QoS are performed between computational nodes of the two sites. These nodes are reserved exclusively for the duration of a file transfer and are co-allocated with the required network resources. This approach guarantees that the reserved network QoS is exclusively available for a given file transfer.

The co-allocation protocol implemented by the orchestration service was initially described in [65] and [66]. It was designed to co-allocate network resources and computational resources from different sites. The co-allocated resources are then used to execute meta-computing applications. The file meta-computing scenario is essentially a generalization of the file transfer scenario. The co-allocation protocol was therefore adapted for co-allocating SLAs, using the WS-Agreement and the WS-Agreement Negotiation protocols. Figure 44 shows the SLA co-allocation protocol at the example of a computational resource provider that uses the WSAG4U framework.

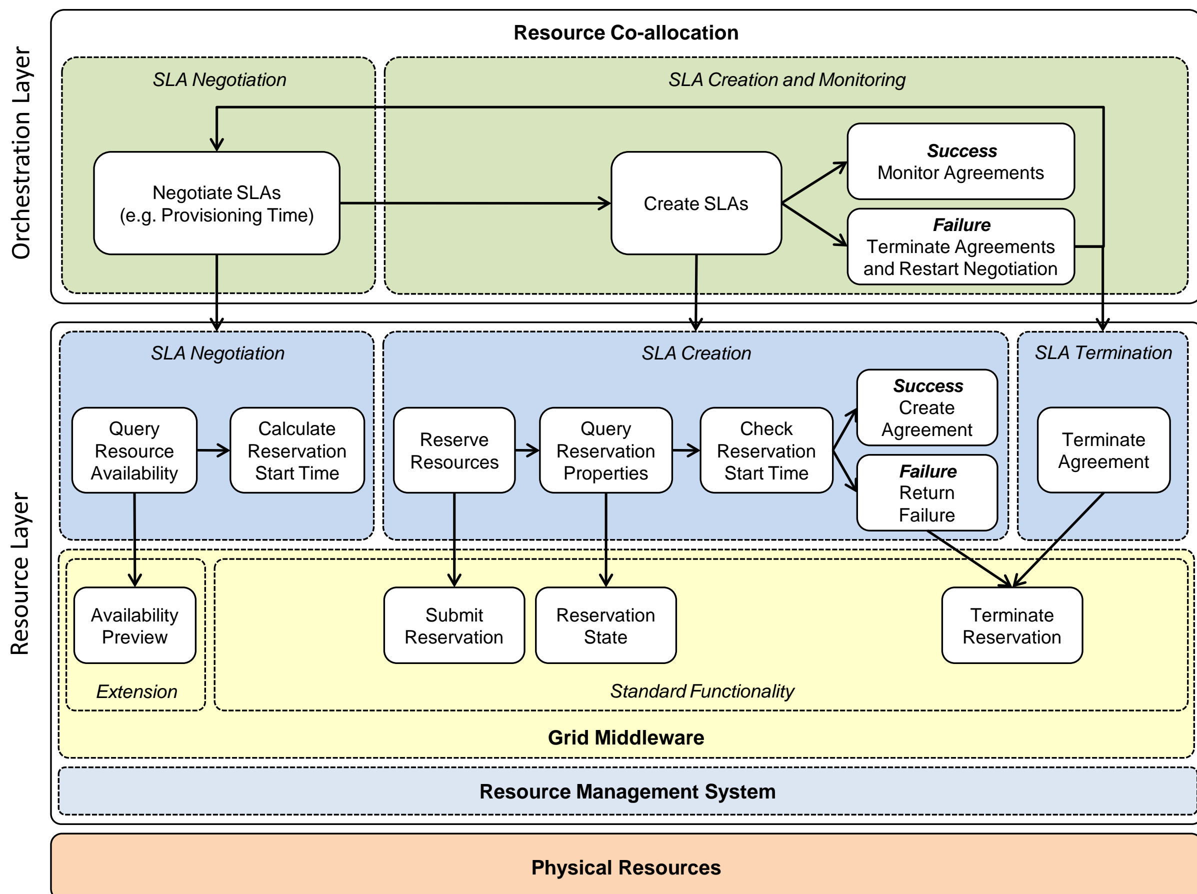


Figure 44: co-allocation of resources using service level agreements

The co-allocation process is divided in two phases, the negotiation phase and the SLA creation phase. During the negotiation phase the orchestration service communicates with the participating resource providers in order to find a common time when the requested resources are available. The WS-Agreement Negotiation protocol is used for this purpose. A negotiation

request contains a description of the requested service, the requested start time and duration of the service provisioning.

When the resource provider receives a negotiation request, it checks the availability of the requested resources. If the resources are available at the requested time, it creates a counter offer in the *Solicited* state indicating that the provider is willing to deliver the service at the requested time. Optionally, the provider can create additional counter offers in the *Advisory* state, for example to indicate alternative execution times. If the requested service cannot be provided at the requested but at a later time, the resource provider creates one or more counter offers in the *Advisory* state in order to indicate alternative execution times. If the requested service can't be provided at all, the resource provider creates a counter offer in the *Rejected* state. This counter offer can optionally include the reason for the rejection.

After the orchestration service received the negotiation responses from each negotiation participator it checks if all participators agreed to provide the particular services at the requested time. This is the case when each negotiation participator confirmed the initial offer by returning a counter offer in the *Solicited* state. If this is not the case, the orchestration service computes a next possible service provisioning time on the base of the received counter offers. Now, the negotiation process is repeated with the computed service provisioning time. The negotiation process completed if one of the following conditions is met:

1. All negotiation participators agree to provide the specified services at the requested time.
2. At least one service provider rejects a negotiation offer indicating that it is not capable to provide the requested service at all.
3. The end of the planning horizon of the orchestration service is reached, e.g. the orchestration service cannot meet a defined deadline.

In case 1 the negotiation process completed successful. The orchestration service creates the negotiated SLAs with the resource providers. In the cases 2 and 3 the co-allocation process failed. The orchestration service can employ failover strategies, for example it can try to allocate an alternative service (e.g. with lower requirements) or choose another resource provider.

Negotiation as discussed here is considered to be a non-binding process. Therefore, it is possible that one of the resource providers does not accept the creation of an agreement based on a negotiated offer. In that case the orchestration service cancels the agreements already created and restarts the negotiation process. It is assumed that there are no penalties for canceling agreements, at least for a defined time period (e.g. up to 5 minutes). Alternatively, the WS-Agreement Negotiation protocol can be used to implement binding negotiations. In binding negotiations the negotiated SLAs are created based on binding offers. In this case, the creation of the negotiated agreements must succeed.

4.6 Evaluation of the Orchestration Service Architecture

The applicability of the orchestration service architecture presented before has been validated for the Phosphorus resource orchestration scenario by a prototypical orchestration service implementation. In the following this architecture is evaluated in order to assess its fitness to support more complex orchestration scenarios. Therefore a set of additional use case scenarios

is generated and the architecture is analyzed to assess its suitability and to identify possible issues.

4.6.1 Dynamic File Transfer Scenario

An enhanced version of the orchestration service should be able to use the available compute resources at different sites even more effective. If at on compute site new resources become available, the orchestration service should allocate these resources dynamically in order to minimize the completion time The compute resources are again allocated from different resource providers and differ in terms of hardware and performance. The computational jobs are distributed to the allocated resources based on an initial submission plan. This submission plan is based on a predicted processing time of the application for a particular unit of work. Since the real processing time may differ from the prediction, the orchestration service should automatically adapt to this situation by rescheduling jobs to machines that perform better. If new compute resources become available at one of the compute resource providers, these resources should also be used for the computation. Since each site processes a subset of the overall workload only the input data that is really needed at one particular site should be transferred from the data storage. If a computational task is finished the results should be immediately transferred back from the compute site to the data storage, from where they can be accessed by the scientist. The file stage in process is initiated by the orchestration service in order to guarantee that the required data is available at job start time and to prevent duplicated data transfers for multiple jobs running at the same site. The file stage out process for each job is initiated by the grid middleware as soon the job completed. For the file transfer processes between the compute resources and the storage server a defined network bandwidth must be available. In order to guarantee that the requested bandwidth is available the orchestration service creates a SLA with a network service provider. This SLA defines a network service that guarantees that compute notes from a specified site can access the data storage server with a defined network QoS. The network provider additionally exposes a management interface for the network service that enables the orchestration service to define a set of compute nodes at that particular site for which the network QoS applies. This is done in order to prevent other nodes at that site to communicate with the data server and therefore reduce the bandwidth that is available to the application. The orchestration service constantly monitors the compute jobs at execution time. If a new job is started, the orchestration service reconfigures the network service in order to make sure that the node that runs the job can communicate with the storage server with the agreed network bandwidth. The same applies if additional resources are allocated at the site. If a compute job is finished and the results are transferred to the storage server, the corresponding compute node is de-allocated by the resource provider and used for other customers. In this case the orchestration service updates the network service configuration and removes the compute node. Therefore, this node does not take advantage of the network SLA anymore.

4.6.2 Dynamic Load Balancing of Web-Applications

In section 4.2.2, we described a dynamic load balancing scenario for web applications. An application service provider offers a web application as a service to its customers and guarantees an average and a maximum response time of the application for a defined number of users. The web application comprises three different components, a web server, a set of

application servers, and a set of database servers. The web server serves as a load balancer for the web application. It distributes incoming requests to the different application servers using a simple load balancing strategy such as round-robin. The business logic of the application is implemented in the application server layer. Finally, the database layer is used to store and retrieve persistent information. At service provisioning time the service provider continuously monitors the different components of its application. In times of high utilization, the provider dynamically scales up the web application, for example by attaching new web application servers and database servers. If the load goes back to normal, the additional allocated resources are detached again. The additional resources are allocated from a dedicated infrastructure provider. The infrastructure provider offers different infrastructure services to its customers. Customers can for example allocate web server instances, application server instances, and database server instances. For each of these infrastructure services the customer can additionally specify the quality with which the service is provided, namely the availability of each service (e.g. 98%, 99%, 99.9%, or 99.99%). The service contract between infrastructure provider and service provider is therefore backed up with an SLA. The infrastructure provider achieves the guaranteed availability rates by providing system redundancy for the server systems, redundancy of storage, preventive maintenance, highly available expert staff, etc. There is a clear separation between the responsibilities of the infrastructure provider and the service provider. The infrastructure provider operates the required hardware and standard software that is used by the service provider. It also implements fault tolerance mechanisms and failure recovery for the physical systems. The application service provider is responsible that its applications scale up and down according to the application utilization. Additional resources are acquired from the infrastructure provider on demand.

The task of scaling the provided web application up and down during runtime is automated by application service provider through an orchestration service. In order to decide if scaling of the web application is required, the orchestration service needs to monitor each of the different application components. In general it must be distinguished between infrastructure specific monitoring and application specific monitoring. Infrastructure specific monitoring produces lower level monitoring data than application specific monitoring. This is for example the load of a specific computer system, e.g. the current CPU utilization of the system, the current memory usage, or the maximum and average load during the last 5, 10 and 15 minutes. This type of monitoring data can be generated in a generic way, regardless of a customer's application. For the specific infrastructure services such as web server, application server and databases, the infrastructure provider can also expose more elaborate monitoring data, such as maximum and average response times, for example done by injecting different types of artificial load to these components. This monitoring data gives a general assessment on the specific service performance and is again implemented independently from customer applications. While infrastructure monitoring generates a view on how the different infrastructure systems and services perform, application monitoring is tailored to specific customer applications and measures the performance of the different application components in a deployment. This can again be done by injecting artificial load to the application components. In contrast to the generic service monitoring at infrastructure level, monitoring at application level gives more fine granular information how the different application

components perform. Infrastructure monitoring and application monitoring therefore provide the foundation for the scaling decisions of the orchestration service.

When the orchestration service determines the demand to scale up the web application due to a load increase it needs to generate a scaling strategy. It therefore uses the application and infrastructure monitoring data to identify the components that have to be scaled. In case the application servers are overloaded but the database layer has still free capacity, a set of new application servers are allocated from the infrastructure provider and a new SLA is created for that purpose. Then the orchestration service dynamically deploys the web application to these application servers and configures them appropriately. Once the web applications are deployed and started, the configuration of the load balancer has to be adopted. The orchestration service therefore updates the current configuration and gracefully restarts the load balancer without interrupting the overall web application. The new application servers are now attached to the web applications, allowing more users to access the application with the defined quality of service. The scale down of the application is realized in a similar way as described. Hereby, the additionally allocated resources are detached from the overall application and the corresponding SLA is terminated.

The orchestration of resources for application service provisioning and the dynamic adoption to users load in order to provide the application with a defined quality require exhaustive interaction between the orchestration service and the infrastructure service. The infrastructure services must provide a wide set of service monitoring and management capabilities in order to enable the orchestration service to optimize the application performance at provisioning time. The responsibilities of QoS-delivery are shared between infrastructure provider and service provider. The infrastructure provider guarantees the availability of the physical systems and infrastructure services by implementing for example redundancy and failover strategies. The service provider focuses on the performance at application level. It therefore implements dynamic scaling strategies to dynamically adapt to users load. The main requirements of the orchestration service on the infrastructure layer can be summed up as follows:

1. Allocate infrastructure services, not only plain resources
2. Capability to deploy applications and data
3. Capability to monitor infrastructure services (e.g. CPU, memory, response time)
4. Capability to configure services interactively
5. Manage the lifecycle of a service independently of the SLA

4.6.3 Evaluation Result

The evaluation scenarios show the requirements of orchestration services to dynamically interact with the infrastructure services for which a SLA was created. Interaction is for example required in order to retrieve detailed monitoring information, to manage the service life cycle, or to actually access the service. The interfaces that are used for these purposes are domain specific and strongly depend on the specific services. The Phosphorus orchestration service architecture does not define a service-oriented representation for infrastructure services. It rather anticipates that the execution of batch jobs does not require user interaction at provisioning time. The compute SLA layer therefore completely encapsulates resource

allocation and job management. The same applies to network SLA layer. For a network SLA the required resources are dynamically allocated and the necessary configuration actions are performed automatically when the network SLA is created. The network SLA layer is therefore integrated with a network management system, which actually allocates the required resources for a network service. The complexity of network management is therefore completely hidden from a user. Both SLA layer implementations do not expose any service to manage the provided resources at runtime. They can implement a non-interactive service provisioning strategy with the goal of autonomously recover from failures and therefore to improve the QoS of the service. This non-interactive service provisioning approach comes to the price of a limited flexibility in using the service. As the evaluation scenarios illustrate, flexibility in service management and usage are key aspects for enhanced orchestration scenarios. Moreover, the lack of such management capabilities results in an increased complexity at the SLA layer. This becomes obvious as soon dependencies between different services exist. If no dynamic service management capabilities exist, all parameters that are required to provide a service must be specified when the SLA is created. In this case, a compute SLA must for example specify the application to execute, the required computational resources, and it must define how input data is handed over to the service provider; respectively how output data is retrieved from the service provider. The same applies to network services. A network SLA must specify the required network QoS between two communication endpoints. If a communication endpoint is represented by a compute SLA, the IP addresses of the actual allocated compute resources must be resolved dynamically by the network SLA implementation. The IP addresses of the actual allocated resources for a compute service are exposed via the state information of a compute SLA. The network SLA implementation needs to realize a self-configuration mechanism in order to bind the IP addresses of the allocated compute nodes to a network reservation at runtime. This results in a dependency between the network SLA and computational SLAs. A similar dependency exists between the compute SLAs. If two computational services must interact in some way, they need to know how to contact each other at runtime. This implies that the IP addresses of the services must be known. A simple file transfer scenario illustrates this problem. In order to transfer a set of files from Site A to Site B, where Site A acts as file sender and Site B acts as file receiver, a set of resources must be co-allocated in order to execute the file sender and the file receiver application. The resource co-allocation is done at the SLA layer by creating appropriate compute SLAs.

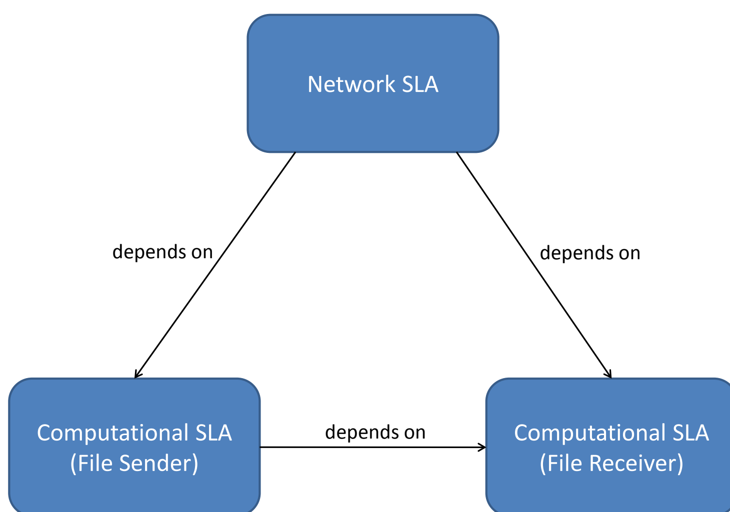


Figure 45: dependencies on SLA layer for file transfers with network QoS

Since the file sender application needs to know the IP address of the file receiver, a dependency between the two compute SLAs exist. If additionally a specific network

bandwidth must be guaranteed for the file transfer, an additional network SLA must be created. In this case, the network SLA is also co-allocated with the compute SLAs. The network SLA depends therefore on both computational SLAs, since the network SLA layer must dynamically resolve the IP addresses of the allocated computational resources at runtime. The dependencies between the different SLAs are depicted in Figure 45.

Another problem of providing computational services in a non-interactive way is the coupling of the service execution with the resource allocation. Especially for resource reservations this approach is rather inflexible in terms of using the reserved resources. For example when computational resources are reserved through a compute SLA, consumers cannot interactively submit activities to these resources. Instead, a predefined service is executed. If the service execution fails (e.g. due to configuration errors), the reserved resources are freed. A service consumer needs to re-schedule a computational job by creating a new SLA.

In order to resolve the before mentioned issues the current orchestration service architecture must be enhanced. The following section describes a reference architecture for orchestration services that addresses the before mentioned issues.

4.7 Enhanced Orchestration Service Architecture

The non-interactive service provisioning is a fundamental pillar for SLA-aware service provisioning. It provides basic functionality to allocate resources and to execute applications together with the capabilities of a SLA layer for electronic contracting, SLA monitoring and accounting. However, as discussed in the evaluation section non-interactive service provisioning is insufficient for more complex resource orchestration scenarios. In order to realize efficient orchestration services additional functionality for service configuration and management must be provided at the service layer. Independent of the service provisioning model (interactive or non-interactive), services should have a service-oriented representation that can be used to configure and manage the service at runtime.

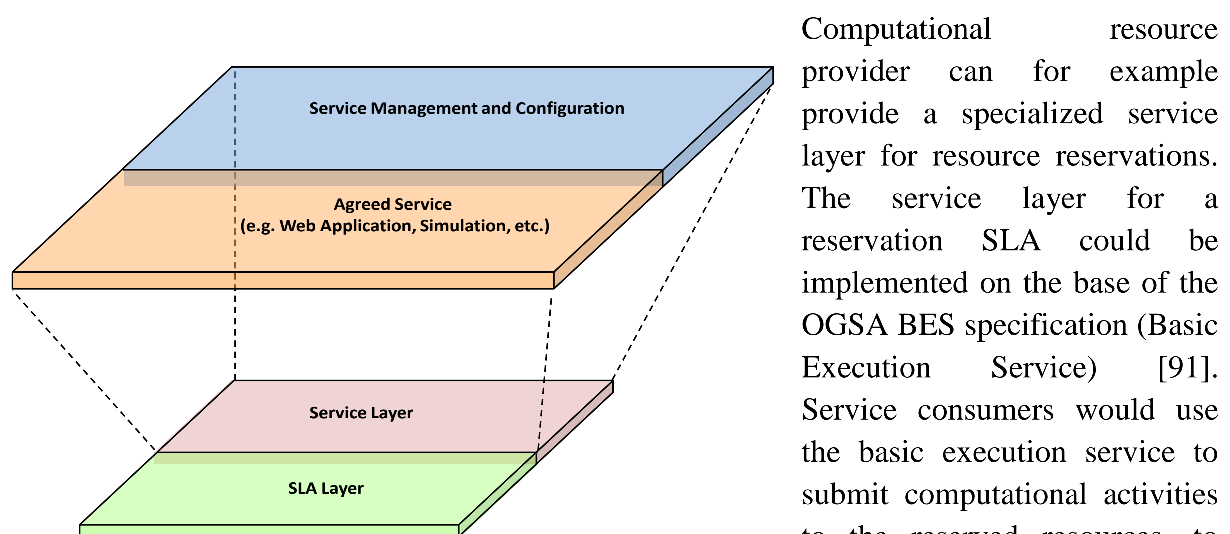


Figure 46: partition of the service layer

Computational resource provider can for example provide a specialized service layer for resource reservations. The service layer for a reservation SLA could be implemented on the base of the OGSA BES specification (Basic Execution Service) [91]. Service consumers would use the basic execution service to submit computational activities to the reserved resources, to manage the activities lifecycle and to monitor the state of the activities. Network service providers can implement similar service layer in order to enable service consumers to configure a network service at runtime, for example to provide a binding

for the communication endpoint IP addresses at runtime. From a conceptual point of view the service layer itself can consist of different parts, for example the provided service itself (e.g. a web based application), a management service and a monitoring service. Figure 46 illustrates this.

The orchestration service still uses the capabilities of the service provider's SLA layers for the service planning and allocation, but service providers implement dedicated, domain specific service layers. These service layers are service-oriented representations of the provided services. They implement functionality to configure, manage and/or access the provided service at runtime. The orchestration service interacts with service-oriented representation at service provisioning time to configure and manage the orchestrated services. This paradigm is reflected in the enhanced orchestration service architecture (see Figure 47).

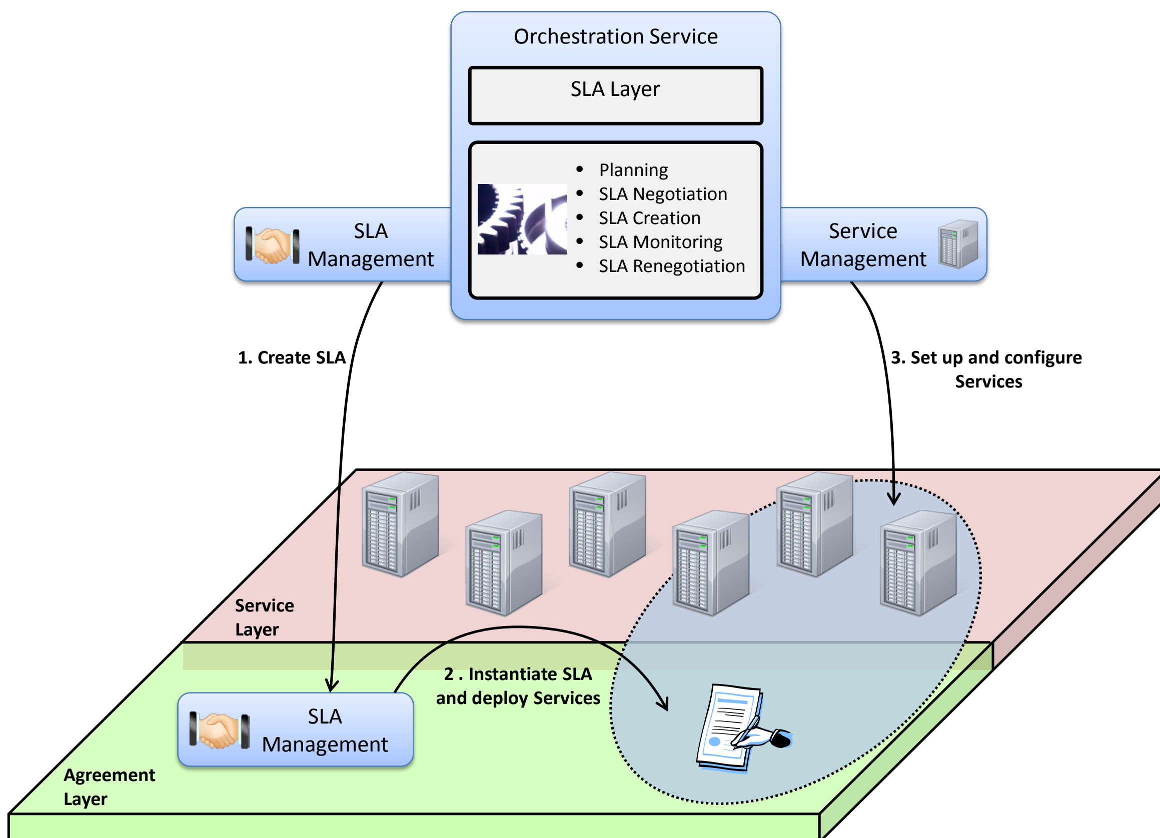


Figure 47: The orchestration service allocates resources via the agreement layer. The allocated resources are then accessed via the service layer.

In contrast to the initial orchestration service architecture, the enhanced architecture foresees that the orchestration service interacts with the resources providers at the SLA layer and on the service layer. The general resource planning and management is done on the SLA layer while the service configuration and management is done on the service layer. This decouples agreement and service layer, eliminates dependencies between service providers at the SLA layer and therefore results in a more flexible architecture.

The co-allocation scenario of computational and network resources for file transfers should serve as example to illustrate the flexibility of this approach. The orchestration service

negotiates and creates SLAs with the computational service provider A (file sender), computational service provider B (file receiver), and network service provider C. The SLAs with service provider A, B and C guarantee that the resources required for a file transfer are available at a requested time. As soon the requested computational resources are allocated by the service providers, the orchestration service looks up the IP addresses of the allocated resources. This information is retrieved via a dedicated monitoring service. Now, the orchestration service binds these IP addresses to the network service. The network service provider exposes an according service management interface at the service layer. The orchestration service then starts a file receiver process at service provider B by starting an according activity on the reserved resources via an OGSA BES interface. In the next step it starts a file sender process at service provider A, again using the OGSA BES interface. The workflow is illustrated in Figure 48.

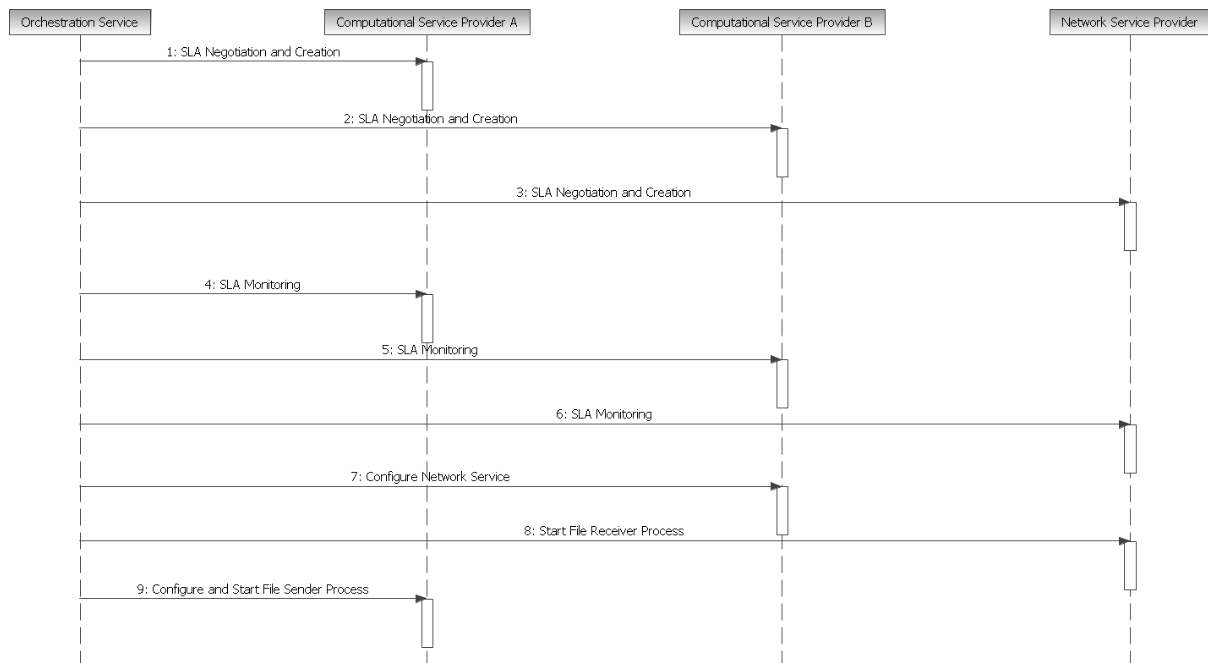


Figure 48: file transfer with network QoS

This enhanced architecture promotes loose coupling of the different resources providers. The dependencies between the computational and network SLAs are eliminated. The responsibility of service configuration and set up is moved to the orchestration service. In this architecture the orchestration service acts as a mediator [29]. In general it can be stated that the enhanced architecture introduces a new dimension into the orchestration service design. On the on hand there is the classical horizontal layering consisting of the user layer, the orchestration layer and the resource provider layer. On the other hand each of these layers comprises a SLA layer and a service layer. The service layer is a virtual representation of the provided service. It can consist of the service itself, service configuration and management, etc. The interfaces of the services exposed at the service layer must be again well defined. This introduces a new level of complexity to the overall system. On the other hand it reduces the complexity of the SLA layer, since dependencies on the SLA layer are eliminated. This

makes the architecture more flexible and robust. Figure 49 shows a detailed overview of the enhanced orchestration service architecture.

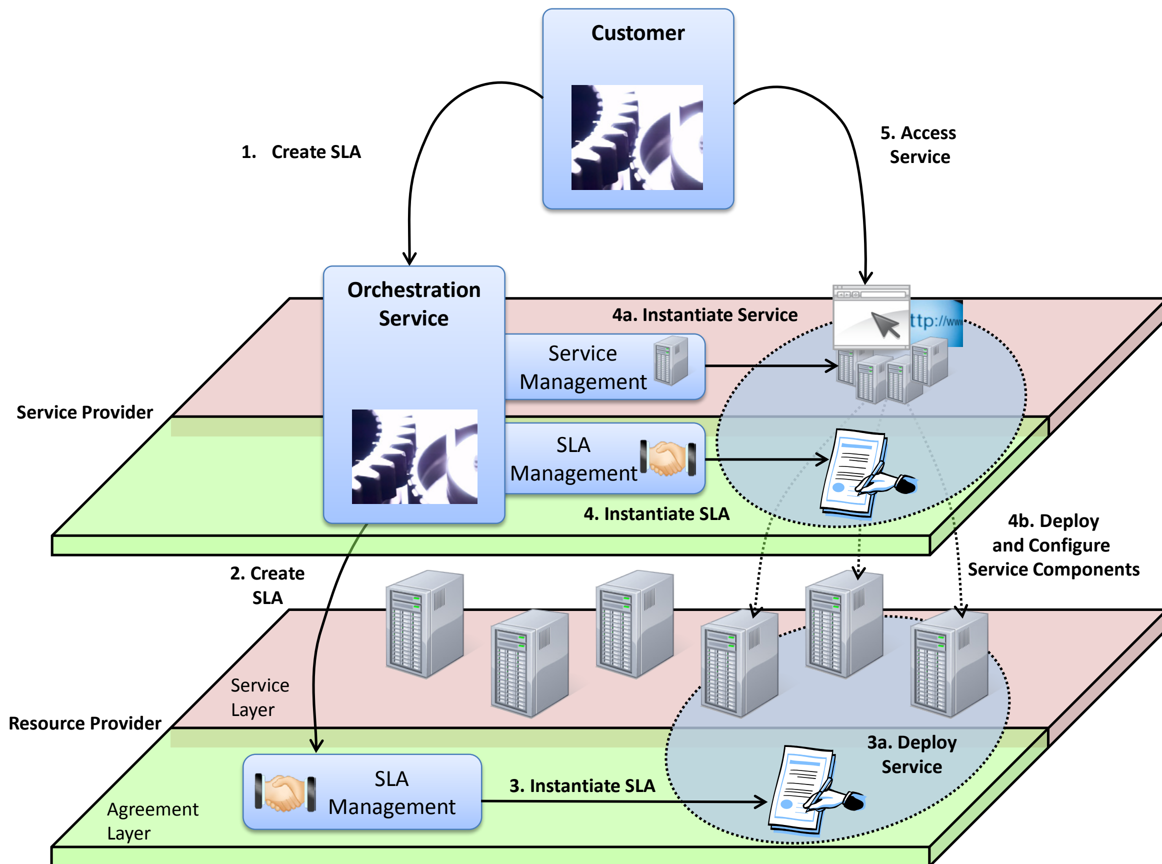


Figure 49: enhanced orchestration service architecture

The orchestration service architecture presented here enhances the initial architecture in order to be more flexibility, to foster robustness and loose coupling of the components. The enhanced architecture assumes that service providers implement service layers that are a service-oriented representation of the provided services. This comprises not only access to the provided service itself, but also service management and configuration capabilities. The service management and configuration capabilities are used by the orchestration service in order to set up the components of orchestrated services. The main advantages of the enhanced architecture can be summed up as follows:

- *Separation of concerns*: A clear separation of the agreement layer and the service layer fosters a clear design of the orchestration service. A generic orchestration service implements a dedicated SLA Management component (SLA negotiation, creation and monitoring) and a dedicated Service Management component (service configuration and management).

- *Reduced complexity of the Agreement Layer*: The orchestration service uses capabilities of the service layer to configure and manage the provided service at runtime. Dependencies on the service layer are no longer transformed to dependencies on the agreement layer. This eliminates dependencies on the agreement layer and therefore reduces the complexity of this layer.

- *Increased Flexibility of the Architecture*: The orchestration service acts as a mediator between the different service providers. The mediator pattern [29] promotes loose coupling of components, in this case the service providers. The implementation of the service providers can vary independently. Due to the loose coupling the reuse of the SLAs offered by the resource providers is promoted.

Figure 50 finally depicts a typical deployment of an orchestration service in an environment with multiple resource providers. The orchestration service discovers available service providers using a public *Service Provider Registry*. The orchestration service's *Planning and Execution* component generates a plan for allocating the required services, for example computational and network services. The *SLA Management* component then negotiates and creates the required SLAs. The active SLAs are then further monitored by this component. Once the agreed services are instantiated, the *Planning and Execution* component performs the required service configuration and management tasks. This is done by the *Service Management* component.

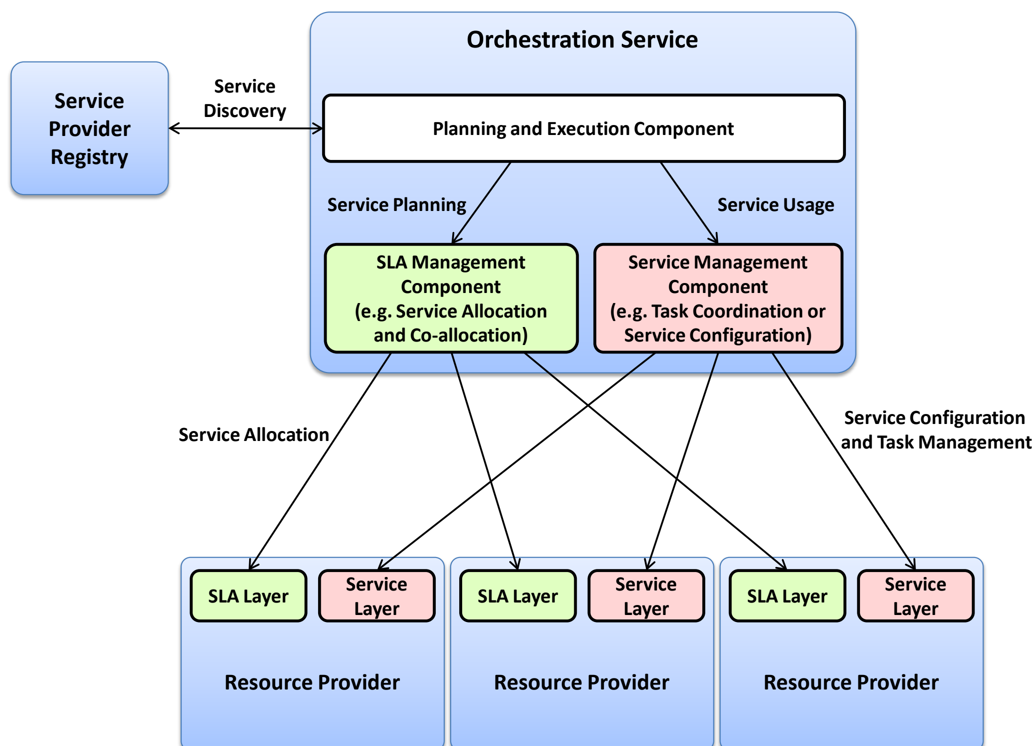


Figure 50: deployment of the orchestration service with multiple resource providers

From a conceptual point of view, the resulting architecture has similarities to the resource management architecture described in [58]. In this approach resource reservations and allocations are distinguished and treated as first class entities, which can be separately created, managed and monitored. Analogously, the architecture presented here distinguishes between *SLAs* and *Services* and treats both as first class entities which can be separately managed and used. However, the concept of SLAs exceeds the concept of simple resource reservations. While reservations can be used to implement specific SLAs, this is not a requirement.

4.8 Summary

In this chapter we presented how orchestration services can use service level agreements in order to coordinate services offered by different service providers and to provide higher-level services. Therefore, we developed an orchestration service architecture and implemented this architecture in order to validate its applicability and identify potential issues. The WSAG4J framework was used to implement the SLA management layers of the different resource providers and has proven itself as a solid foundation of SLA enabled service-provisioning systems. Nevertheless, the prototypical implementation of the orchestration service architecture has revealed an important issue. The initial architecture did not distinguish between service layer and agreement layer. The service orchestration and configuration was handled on the agreement layer. This results in an increasing complexity of the SLA layer, since dependencies that exist on only between services at runtime are transposed to dependencies at the agreement layer. In order to overcome this issue we presented an enhanced version of the orchestration service architecture. This architecture promotes a clear separation of concerns. It distinguishes between service allocation at the SLA layer and service usage at the service layer. The service layer can also comprise service management capabilities in order to allow applications to use the provided services in a flexible and dynamic way, e.g. to configure a service during runtime. This reduces the complexity of the SLA layer and promotes the reuse of SLAs. Moreover, dependencies that exist between services at runtime can be resolved by the orchestration service using the capabilities of the service layer. The overall orchestration service architecture therefore becomes more flexible.

CONCLUSION

Service Level Agreements provide a useful mechanism for automatic service provisioning in distributed environments such as the Grid. They help service consumers and providers to express the relevant aspects of a service, comprising functional and quality aspects, and to gain a mutual understanding of the service. The service quality is thereby often a unique selling point. The ability to dynamically negotiate the quality of a service is therefore essential for distributed systems where services are automatically acquired and provided. SLA-aware service provisioning essentially comprises service provisioning aspects and the SLA management aspects. Service provisioning investigates solutions to provide services with a defined quality, to increase the efficiency of the service, and to cope with failures during the provisioning process. SLA management deals with the process of SLA negotiation, creation, monitoring, and accounting. These aspects are in general independent of an application domain and can therefore be handled in a generic way.

In this thesis we focused on SLA management aspects in distributed systems. We discussed basic requirements for service level agreements in distributed systems and investigated existing SLA management approaches. We motivated our decision for WS-Agreement as foundation for a generic SLA management system and analyzed WS-Agreement with respect to the requirements defined before. Based on this analysis we identified open issues of WS-Agreement, namely the missing negotiation support.

SLA negotiation is an important aspect of dynamic, SLA-aware service provisioning. In this thesis we present the specification of a SLA Negotiation framework that complements WS-Agreement with negotiation capabilities. The specification comprises an abstract SLA negotiation model that extends the basic WS-Agreement model. We defined a protocol and language for bi-lateral agreement negotiation. Protocol and language are designed to be domain independent and flexible in order to be applicable in a wide range of usage scenarios. Furthermore, we discussed fundamental design features, such as protocol symmetry and coupling of negotiation and agreement layer. This specification constitutes a major contribution to the area of dynamic SLA management.

Then we presented our WSAG4J framework as an example of a generic, WS-Agreement based SLA management system. This framework automates common SLA management tasks such as SLA compliance validation, SLA monitoring, and accounting. SLA compliance validation ensures the integrity of agreement offers in a SLA creation processes. It protects service providers and consumers from malicious manipulation of SLA offers. The ability to validate the integrity of complex agreement offers is a fundamental requirement in distributed systems in order to automatically create electronic contracts without human intervention. We presented a dynamic offer validation model for WS-Agreement based systems, which is an important contribution of this thesis to the area of dynamic SLA processing. Furthermore, we presented a SLA monitoring and accounting model. Based on this model it is possible to dynamically resolve the properties of a service based on the service monitoring data, to

compute the state of a SLA and its guarantees based on the service properties, and to account penalties and rewards dependent on the fulfillment of SLA guarantees. This is another contribution to the area of dynamic SLA processing. Overall, the WSAG4J framework already automates big parts of SLA management in distributed systems. It fosters the design of SLAs and improves the traceability of SLA accounting.

Finally, we presented an architecture for SLA-based resource orchestration services in the Grid. In order to evaluate this architecture we implemented an orchestration service and a set of SLA-based resource providers. The orchestration service supports to provisioning of embarrassingly parallel applications in Software as a Service (SaaS) scenarios. Based on the architectural evaluation we identified shortcomings in the initial architecture, which were addressed in an enhanced architecture.

To sum up, this thesis presented important contributions in the area of automated, SLA-aware service provisioning in distributed systems. Future research can benefit from this work as another step in building QoS-aware distributed systems.

Contributions

In this thesis we investigated methods for SLA-aware service provisioning and SLA management based on the WS-Agreement specification [20] and how these methods can be adopted in resource orchestration scenarios. The thesis includes the following contributions:

Processing Model for SLAs

The SLA processing model describes the required mechanisms to validate, create, monitor and evaluate Service Level Agreements. SLA validation is a fundamental mechanism to automatically determine the integrity of SLA requests. Since SLAs are electronic contracts it is required that resource management systems fully understand the content of these contracts in order to prevent malicious manipulation of SLA requests. The SLA processing model furthermore describes a service monitoring model that builds the foundation of the SLA evaluation and accounting model.

Negotiation and Renegotiation Model for SLAs

In order to create a SLA service consumer and provider often need to dynamically exchange information in order to gain a common understanding of a service delivery. This process is called SLA negotiation. The same applies to situations in which existing SLAs should be modified in order to adapt to changed requirements of a service consumer or provider. This process is called the SLA renegotiation. The SLA negotiation model presented in this thesis describes the basic mechanisms to negotiate and renegotiate SLAs in an automated way.

An Architecture and Software for SLA Negotiation, Creation, Monitoring and Evaluation

SLA negotiation, validation, creation, and monitoring are common tasks in SLA-aware systems. Depending on the service provisioning model, these systems implement domain specific service provisioning and monitoring strategies of different complexity. However, the SLA management functionalities are essentially the same. In this thesis we present a generic

SLA framework that automates basic SLA management tasks in distributed systems in a standardized way based on the WS-Agreement specification.

An Architecture and Software for SLA-based Orchestration Services

We present an abstract model for a SLA-based Orchestration Service that is capable of using infrastructural services, such as computational and network services, in order to provide a higher level software service. The Orchestration Service is prototypically implemented and its architecture is evaluated in order to identify potential weaknesses of the design. Based on the findings of the evaluation process, we present an enhanced architectural model for the Orchestration Service.

Bibliography

1. **Foster, I. and Kesselman, C.** Computational Grids. *Vector and Parallel Processing (VECPAR 2000).* Springer Berlin / Heidelberg, 2001.
2. **Foster, I. and Kesselman, C.** *The Grid 2: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, 2004.
3. http://www.globus.org/toolkit/. *Globus Toolkit.* [Online] http://www.globus.org/toolkit/. Globus Alliance. [Cited: June 30, 2010.]
4. UNICORE. *Distributed computing and data resources.* [Online] http://www.unicore.eu/. Jülich Supercomputing Centre. [Cited: May 25, 2011.]
5. **Waeldrich, O.** *WSAG4J - WS-Agreement Framework for Java.* [Online] http://packcs-e0.scai.fraunhofer.de/wsag4j/. Fraunhofer. [Cited: April 10, 2010.]
6. Phosphorus Project. [Online] http://www.ist-phosphorus.eu/. [Cited: May 25, 2011.]
7. **MacKenzie, C. M. and Laskey, K. and McCabe, F. and Brown, P. F. and Metz, R.** Reference Model for Service Oriented Architecture 1.0. *Reference Model for Service Oriented Architecture 1.0.* [Online] http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html. October 12, 2006 [Cited: March 24, 2010.].
8. **Krafzig, D. and Banke, K. and Slama, D.** *Enterprise SOA: Service-Oriented Architecture Best Practices.* Prentice Hall International, 2005.
9. **Christensen, E. and Curbera, F. and Meredith, G. and Weerawarana, S.** Web Services Description Language (WSDL) 1.1. *Web Services Description Language (WSDL) 1.1.* [Online] http://www.w3.org/TR/wsdl. March 15, 2001 [Cited: March 26, 2010.].
10. *OASIS UDDI Specifications.* [Online] http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm. [Cited: July 21, 2010.].
11. W3C SOAP Specifications. [Online] http://www.w3.org/TR/soap/. [Cited: July 21, 2010.].
12. IT Infrastructure Library (ITIL). [Online] http://www.itil-officialsite.com. [Cited: July 22, 2010.]
13. **Lamanna, D. and Skene, J. and Emmerich, W.** SLAng: A Language for Defining Service Level Agreements. *FTDCS '03: Proceedings of the The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems.* IEEE Computer Society, 2003.
14. **Pontz, T. and Grauer, M. and Kuebert, R. and Tenschert, A. and Koller, B.** Evaluation of Service Level Agreements for Portfolio Management in the Financial Industry. *Grids and Service-Oriented Architectures for Service Level Agreements.* pp. 57-66, Springer US, 2010.
15. **Tang, Y. and Lutfiyya, H. and Tosic, V.** *An analysis of web service SLA management infrastructures based on the C-MAPE model.* In International Journal of Business Process Integration and Management, Vol. 4, Issue 3, pp. 209-218, Inderscience Publishers, 2009. ISSN 1741-8763.
16. **Keller, A. and Ludwig, H.** *The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services.* In Journal of Network and Systems Management, Vol. 11, Issue 1, pp. 57-81, Springer, New York, 2003. ISSN 1064-7570.

17. IBM. [Online] http://www.ibm.com. [Cited: May 25, 2011.]
18. Web Service Level Agreements (WSLA) Project. [Online] http://www.research.ibm.com/wsla/. [Cited: July 21, 2010.].
19. **Tosic, V. and Patel, K. and Pagurek, B.** WSOL - Web Service Offerings Language. *LNCS: Web Services, E-Business, and the Semantic Web.* Vol. 2512, pp. 57-67, Springer Berlin/Heidelberg, 2002.
20. **Andrieux, A. and Czajkowski, K. and Dan, A. and Keahey, K. and Ludwig, H. and Nakata, T. and Pruyne, J. and Rofrano, J. and Tuecke, S. and Xu, M.** WS-Agreement specification. [Online] http://www.ogf.org/documents/GFD.107.pdf. May 25, 2007 [Cited: November 24, 2009.].
21. Open Grid Forum. [Online] http://www.ogf.org/. [Cited: May 25, 2011.]
22. **Battré, D. and Wieder, P. and Ziegler, W.** WS-Agreement Specification Version 1.0 Experience Document. [Online] http://www.ogf.org/documents/GFD.167.pdf. March 8, 2010 [Cited: July 21, 2010.].
23. **Bray, T. and Paoli, J. and Sperberg-McQueen, C. M. and Maler, E. and Yergeau, F.** *Extensible Markup Language (XML) 1.0 (Fifth Edition).* [Online] http://www.w3.org/TR/REC-xml. November 26, 2008 [Cited: March 24, 2010.].
24. **Thompson, H. S. and Beech, D. and Maloney, M. and Mendelsohn, N.** XML Schema Part 1: Structures Second Edition. *XML Schema Part 1: Structures Second Edition.* [Online] http://www.w3.org/TR/xmlschema-1/. October 28, 2004 [Cited: March 24, 2010.].
25. **Biron, P. V. and Malhotra, A.** XML Schema Part 2: Datatypes Second Edition. *XML Schema Part 2: Datatypes Second Edition.* [Online] http://www.w3.org/TR/xmlschema-2/. October 28, 2004 [Cited: March 24, 2010.].
26. Organization for the Advancement of Structured Information Standards. [Online] http://www.oasis-open.org. [Cited: May 25, 2011.]
27. OASIS Web Services Resource Framework (WSRF) TC. [Online] http://www.oasis-open.org/committees/wsrf. [Cited: March 26, 2010.]
28. **Gudgin, M. and Hadley, M. and Rogers, T.** Web Services Addressing 1.0 - Core. *Web Services Addressing 1.0 - Core.* [Online] http://www.w3.org/TR/ws-addr-core/. W3C, May 9, 2006. [Cited: 3 29, 2010.]
29. **Gamma, E. and Helm, R. and Johnson, R. and Vlissides, J.** *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, 1995.
30. **Graham, S. and Treadwell, J.** Web Services Resource Properties 1.2. *Web Services Resource Properties 1.2.* [Online] http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-os.pdf. April 1, 2006 [Cited: March 30, 2010.].
31. **Berglund, A. and Boag, S. and Chamberlin, D. and Fernández, M. F. and Kay, M. and Robie, J. and Siméon, J.** XML Path Language (XPath) 2.0. *XML Path Language (XPath) 2.0.* [Online] http://www.w3.org/TR/xpath20/. January 23, 2007 [Cited: April 1, 2010.].
32. **Waeldrich, O. and Battré, D. and Brazier, F. and Clark, K. and Oey, M. and Papaspyrou, A. and Wieder, P. and Ziegler, W.** WS-Agreement Negotiation Specification (draft). [Online] http://www.ogf.org/Public_Comment_Docs/Documents/2011-03/WS-Agreement-Negotiation+v1.0.pdf. January 31, 2011 [Cited: May 25, 2011.].
33. **Bradner, S.** Key words for use in RFCs to Indicate Requirement Levels. *The Internet Engineering Task Force Best Current Practice.* [Online] http://www.ietf.org/rfc/rfc2119.txt. March 1997 [Cited: March 08, 2010.].

34. Java. [Online] http://www.java.com. Oracle. [Cited: May 25, 2011.]
35. **Seidel, J. and Waeldrich, O. and Ziegler, W. and Wieder, P. and Yahyapour, R.** Using SLA for Resource Management and Scheduling – A Survey. *Grid Middleware and Services – Challenges and Solutions. Proceedings of the Usage of Service Level Agreements in Grids Workshop in conjunction with the 8th IEEE International Conference on Grid Computing Grid 2007.* Springer US, 2008.
36. **Ludwig, H. and Dan, A. and Robert, K.** Cremona: An Architecture and Library for Creation and Monitoring of WS-Agreements. *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing.* pp. 65-74, ACM, New York, NY, USA, 2004.
37. AgentScape Operating System. [Online] http://www.agentscape.org/. [Cited: July 30, 2010.]
38. **Mobach, D. and Overeinder, B. and Brazier, F.** *A WS-Agreement based resource negotiation framework for mobile agents.* In Scalable Computing: Practice and Experience, Vol. 7, Issue 1, pp. 23-36, 2006.
39. AssessGrid. [Online] http://www.assessgrid.eu/. [Cited: June 30, 2010.]
40. **Voss, K. and Djemame, K. and Gourlay, I. and Padgett, J.** AssessGrid, Economic Issues Underlying Risk Awareness in Grids. *LNCS: Grid Economics and Business Models.* Vol. 4685, pp. 170-175, Springer Berlin/Heidelberg, 2007.
41. **Ardaiz, O. and Freitag, F. and Navarro, L. and Eymann, T. and Reinicke, M.** CatNet: Catallactic Mechanisms for Service Control and Resource Allocation in Large-Scale Application-Layer Networks. *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid.* IEEE Computer Society, Washington, DC, USA, 2002.
42. The BREIN Project. [Online] http://www.gridsforbusiness.eu/. [Cited: August 2, 2010.]
43. The BEinGRID Project. [Online] http://www.beingrid.eu. [Cited: August 2, 2010.]
44. JSS - The Job Submission Service. [Online] https://www8.cs.umu.se/research/grid/jss/. [Cited: August 2, 2010.]
45. **Elmroth, E. and Tordsson, J.** An Interoperable, Standards-Based Grid Resource Broker and Job Submission Service. *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing.* pp. 212-220, IEEE Computer Society, Washington, DC, USA, 2005.
46. Apache Muse. *A Java-based implementation of WSRF 1.2, WSN 1.3, and WSDM 1.1.* [Online] http://ws.apache.org/muse/. Apache Foundation. [Cited: June 30, 2010.]
47. Unicore WSRF Lite. [Online] http://www.unicore.eu/documentation/manuals/unicore6/wsrflite. JÜLICH SUPERCOMPUTING CENTRE. [Cited: June 30, 2010.]
48. WSRF::Lite. [Online] http://www.rcs.manchester.ac.uk/research/wsrflite. University of Manchester. [Cited: June 2010, 30.]
49. WebSphere Application Server . [Online] http://www-01.ibm.com/software/webservers/appserv/was/. IBM. [Cited: June 30, 2010.]
50. **Anjomshoaa, A. and Brisard, F. and Drescher, M. and Fellows, D. and Ly, A. and McGough, S. and Pulsipher, D. and Savva, A.** *Job Submission Description Language (JSDL) Specification, Version 1.0.* [Online] http://www.gridforum.org/documents/GFD.56.pdf. November 7, 2005 [Cited: April 12, 2010.].
51. Schematron. [Online] http://www.schematron.com. [Cited: May 25, 2011.]

52. XMLBeans. [Online] http://xmlbeans.apache.org. Apache Foundation. [Cited: May 25, 2011.]
53. Xerces-J. [Online] http://xerces.apache.org/xerces-j/. Apache Foundation. [Cited: July 1, 2010.]
54. Commons JEXL. [Online] http://commons.apache.org/jexl/. Apache Foundation. [Cited: April 19, 2010.]
55. **Boag, S. and Chamberlin, D. and Fernández, M. F. and Florescu, D. and Robie, J. and Siméon, J.** XQuery 1.0: An XML Query Language. *XQuery 1.0: An XML Query Language.* [Online] http://www.w3.org/TR/xquery/. W3C, January 23, 2007. [Cited: April 25, 2010.]
56. **Yahyapour, R. and Wieder, P.** Grid Scheduling Use Cases. [Online] http://www.ogf.org/documents/GFD.64.pdf. March 26, 2006 [Cited: August 3, 2010.].
57. **Schopf, J. M.** Ten actions when Grid scheduling: the user as a Grid scheduler. *Grid resource management: state of the art and future trends.* Kluwer Academic Publishers, Norwell, MA, USA, 2004.
58. **Foster, I. and Kesselman, C. and Lee, C. and Lindell, B. and Nahrstedt, K. and Roy, A.** A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. *Proceedings of the International Workshop on Quality of Service.* 1999.
59. The KOALA Grid Scheduler. [Online] http://www.st.ewi.tudelft.nl/koala/. TU Delft. [Cited: August 4, 2010.]
60. **Mohamed, H. and Epema, D.** *KOALA: a co-allocating grid scheduler.* In Concurrency and Computation: Practice and Experience, Vol. 20, Issue 16, pp. 1851-1876, John Wiley and Sons Ltd., Chichester, UK, 2008. ISSN 1532-0626.
61. GridWay Metascheduler. [Online] http://www.gridway.org. [Cited: August 4, 2010.]
62. **Montero, R. S. and Huedo, E. and Llorente, I. M.** Grid Resource Selection for Opportunistic Job Migration. *LNCS: Euro-Par 2003 Parallel Processing.* Springer, Berlin/Heidelberg, 2004.
63. **Huedo, E. and Montero, R. S. and Llorente, I. M.** *A modular meta-scheduling architecture for interfacing with pre-WS and WS Grid resource management services.* In Future Generation Computer Systems, Vol. 23, Issue 2, Elsevier Science Publishers, Amsterdam, The Netherlands, 2007. ISSN 0167-739X.
64. VIOLA - Vertically Integrated Optical Testbed for Large Applications. [Online] http://www.viola-testbed.de. 2004 [Cited: June 1, 2010.].
65. **Waeldrich, O. and Wieder, P. and Ziegler, W.** A Meta-scheduling Service for Co-allocating Arbitrary Types of Resources. *LNCS: Parallel Processing and Applied Mathematics.* Vol. 3911, Springer Berlin / Heidelberg, 2006.
66. **Eickermann, T. and Westphal, L. and Waeldrich, O. and Ziegler, W. and Barz, C. and Pilz, M.** Co-allocating compute and network resources - bandwidth on demand in the Viola testbed. *Towards Next Generation Grids.* pp. 193-202, Springer US, 2007.
67. **Keller, V. and Gruber, R. and Spada, M. and Tran, T.-M. and Cristiano, K. and Kuonen, P. and Wieder, P. and Ziegler, W. and Waeldrich, O. and Maffioletti, S. and Sawley, M.-C. and Nellari, N.** Integration of ISS into the VIOLA Meta-Scheduling Environment. *Integrated Research in GRID Computing.* Springer US, 2007.
68. **Rasheed, H. and Gruber, R. and Keller, V. and Ziegler, W. and Waeldrich, O. and Wieder, P. and Kuonen, P.** IANOS: Intelligent Application Oriented Scheduling for HPC Grids. [Online] http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0160.pdf. July 2008 [Cited: June 1, 2010.].

69. **Drotz, A. and Gruber, R. and Keller, V. and Thiémard, M. and Tolou, A. and Tran, T.-M. and Cristiano, K. and Kuonen, P. and Wieder, P. and Waeldrich, O. and Ziegler, W. and Manneback, P. and Schwiegelshohn, U. and Yahyapour, R. and Kunszt, P.** Application-oriented scheduling for HPC Grids. [Online] http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0070.pdf. February 2007 [Cited: June 1, 2010.].
70. **Gruber, R. and Keller, V. and Thiémard, M. and Waeldrich, O. and Wieder, P. and Ziegler, W. and Manneback, P.** Integration of Grid Cost Model into ISS/VIOLA Meta-scheduler Environment. *LNCS: Euro-Par 2006 Parallel Processing.* Vol. 4375, pp. 215-224, Springer Verlag, 2007.
71. **Keller, V. and Rasheed, H. and Waeldrich, O. and Ziegler, W. and Gruber, R. and Sawley, M.-C. and Wieder, P.** *Models and internals of the IANOS resource broker.* In , Vol. 23, Issue 3, pp. 259-266, Springer, Berlin/Heidelberg, June 2009.
72. **Shahid, M. and Hofmann-Apitius, M. and Waeldrich, O. and Ziegler, W.** A robust framework for rapid deployment of a virtual screening laboratory. [book auth.] Solomonides, T. and Hofman-Apitius, M. and Freudigmann, M. and Semler, S. and Legré, Y. and Kratz, M. *Healthgrid research, innovation and business case: Proceedings of HealthGrid 2009.* IOS Press, Amsterdam, 2009.
73. SmartLM. *Grid-friendly software licensing for location independent application execution.* [Online] http://www.smartlm.eu. [Cited: May 25, 2011.]
74. SLA4D-Grid. *Service Level Agreements for D-Grid.* [Online] http://www.sla4d-grid.de/. [Cited: May 25, 2011.]
75. DGSI. *D-Grid Scheduler Interoperability.* [Online] http://www.d-grid-ggmbh.de/index.php?id=98. [Cited: June 2, 2010.]
76. SLA@SOI. *Service Level Agreements at Service Oriented Infrastructures.* [Online] http://sla-at-soi.eu/. [Cited: May 25, 2011.]
77. **Comuzzi, M. and Spanoudakis, G.** Dynamic set-up of Monitoring Infrastructures for Service Based Systems. *Symposium on Applied Computing archive: Proceedings of the 2010 ACM Symposium on Applied Computing.* pp. 2414-2421, ACM New York, 2010.
78. The THESEUS Project. [Online] http://www.theseus-programm.de/. [Cited: July 5, 2010.]
79. **Spillner, J. and Winkler, M. and Reichert, S. and Cardoso, J. and Schill, A.** Distributed Contracting and Monitoring in the Internet of Services. *LNCS: Distributed Applications and Interoperable Systems.* pp. 129-142, Springer Berlin / Heidelberg, 2009.
80. **Shahid, M. and Klatt, T. and Rasheed, H. and Waeldrich, O. and Ziegler, W.** *SCAI-VHTS - A Fully Automated Virtual High Throughput Screening Framework.* In ERCIM News, Vol. 82, pp. 23-24, ERCIM EEIG, Sophia Antipolis, 2010. ISSN 0926-4981.
81. *gLite: Lightweight Middleware for Grid Computing.* [Online] http://glite.web.cern.ch/glite/. CERN. [Cited: July 2, 2010.]
82. TORQUE Resource Manager. [Online] http://www.clusterresources.com/products/torque/. Cluster Resources. [Cited: July 2, 2010.]
83. Platform LSF. [Online] http://www.platform.com/Products/platform-lsf. Platform. [Cited: July 2, 2010.]
84. Oracle Grid Engine. [Online] http://www.sun.com/software/sge/. Oracle. [Cited: July 2, 2010.]
85. **Waeldrich, O.** *WSAG4U - WS-Agreement for Unicore 6.* [Online] http://packcs-e0.scai.fraunhofer.de/wsag4unicore6/. Fraunhofer. [Cited: April 10, 2010.]
86. **Battré, D. and Waeldrich, O.** Time Constraints Profile. [Online]

https://forge.gridforum.org/sf/docman/do/downloadDocument/projects.graap-wg/docman.root.current_drafts.ws_agreement_advance_reservation/doc15832. October 11, 2009 [Cited: May 31, 2010.].

87. Harmony. *Harmony Network Resource Brokering System.* [Online] http://www.ist-phosphorus.eu/software.php?id=harmony. [Cited: May 20, 2010.]
88. Network Mark-up Language Working Group (NML-WG). [Online] http://www.ogf.org/gf/group_info/view.php?group=nml-wg. [Cited: May 31, 2010.].
89. **Tuecke, S. and Welch, V. and Engert, D. and Pearlman, L. and Thompson, M.** Internet X.509 Public Key Infrastructure (PKI) - Proxy Certificate Profile. [Online] http://www.ietf.org/rfc/rfc3820.txt. June 2004. [Cited: May 29, 2010.]
90. **Keller, V.** Optimal Application-Oriented Resource Brokering in a High Performance Computing Grid. [PhD Thesis]. École Polytechnique Fédérale de Lausanne, Lausanne, November 12, 2008.
91. **Foster, I. and Grimshaw, A. and Lane, P. and Lee, W. and Morgan, M. and Newhouse, S. and Pickles, S. and Pulsipher, D. and Smith, C. and Theimer, M.** OGSA Basic Execution Service. [Online] http://www.ogf.org/documents/GFD.108.pdf. 2007

Appendix

WS-Agreement Negotiation XML Schema

```
<xsd:schema
    elementFormDefault="qualified" attributeFormDefault="qualified"
    targetNamespace="http://schemas.ogf.org/graap/2009/11/ws-agreement-negotiation"
    xmlns:wsag-neg="http://schemas.ogf.org/graap/2009/11/ws-agreement-negotiation"
    xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement"
    xmlns:wsa="http://www.w3.org/2005/08/addressing"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <xsd:import
      namespace="http://schemas.ggf.org/graap/2007/03/ws-agreement"
      schemaLocation="http://schemas.ggf.org/graap/2007/03/ws-agreement" />
    <xsd:import namespace="http://www.w3.org/2001/XMLSchema"
      schemaLocation="http://www.w3.org/2001/XMLSchema.xsd" />
    <xsd:import namespace="http://www.w3.org/2005/08/addressing"
      schemaLocation="http://www.w3.org/2005/08/addressing/ws-addr.xsd"/>

    <xsd:element name="NegotiationContext"
        type="wsag-neg:NegotiationContextType" />
    <xsd:element name="NegotiatiableTemplate"
        type="wsag:AgreementTemplateType" />
    <xsd:element name="NegotiationOffer"
        type="wsag-neg:NegotiationOfferType" />
    <xsd:element name="NegotiationCounterOffer"
```

```
        type="wsag-neg:NegotiationOfferType" />
    <xsd:element name="NegotiationOfferContext"
        type="wsag-neg:NegotiationOfferContextType" />
    <xsd:element name="NegotiationExtension"
        type="wsag-neg:NegotiationExtensionType" />
    <xsd:element name="RenegotiationExtension"
        type="wsag-neg:RenegotiationExtensionType" />

    <xsd:complexType name="NegotiationContextType">
      <xsd:sequence>
        <xsd:element name="NegotiationType"
            type="wsag-neg:NegotiationType" />
        <xsd:element name="ExpirationTime"
            type="xsd:dateTime" minOccurs="0" />
        <xsd:element name="NegotiationInitiator"
            type="xsd:anyType" minOccurs="0" />
        <xsd:element name="NegotiationResponder"
            type="xsd:anyType" minOccurs="0" />
        <xsd:element name="AgreementResponder"
            type="wsag-neg:NegotiationRoleType"/>
        <xsd:element name="AgreementFactoryEPR"
            type="wsa:EndpointReferenceType" minOccurs="0"/>
        <xsd:any namespace="##other" processContents="lax"
            minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>

    <xsd:simpleType name="NegotiationRoleType">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="NegotiationInitiator" />
        <xsd:enumeration value="NegotiationResponder" />
```

```
      </xsd:restriction>
    </xsd:simpleType>

    <xsd:complexType name="NegotiationType">
      <xsd:choice>
        <xsd:element name="Negotiation">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:any namespace="##other" processContents="lax"
                   minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="Renegotiation">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="ResponderAgreementEPR"
                   type="wsa:EndpointReferenceType" />
              <xsd:element name="InitiatorAgreementEPR"
                   type="wsa:EndpointReferenceType" minOccurs="0" />
              <xsd:any namespace="##other" processContents="lax"
                   minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:choice>
    </xsd:complexType>

    <xsd:complexType name="NegotiationOfferType">
        <xsd:complexContent>
            <xsd:extension base="wsag:AgreementType">
```

```
                <xsd:sequence>
                  <xsd:element name="NegotiationOfferContext"
                       type="wsag-neg:NegotiationOfferContextType"/>
                  <xsd:element name="NegotiationConstraints"
                       type="wsag-neg:NegotiationConstraintSectionType" />
                </xsd:sequence>
                <xsd:attribute name="OfferId" type="xsd:string" />
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="NegotiationConstraintSectionType">
        <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="0" name="Item"
                type="wsag-neg:NegotiationOfferItemType" />
            <xsd:element maxOccurs="unbounded" minOccurs="0"
                ref="wsag:Constraint" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="NegotiationOfferItemType">
        <xsd:complexContent>
            <xsd:extension base="wsag:OfferItemType">
                <xsd:attribute name="Type"
                               type="wsag-neg:NegotiationConstraintType"
                               use="required" />
                <xsd:attribute name="Importance" type="xsd:integer"
                               default="0" use="optional"/>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
```

```xml
<xsd:simpleType name="NegotiationConstraintType">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Required" />
        <xsd:enumeration value="Optional" />
    </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="NegotiationOfferContextType">
  <xsd:sequence>
    <xsd:element name="CounterOfferTo"
          type="xsd:string"/>
    <xsd:element name="ExpirationTime"
          type="xsd:dateTime" minOccurs="0" />
    <xsd:element name="Creator"
          type="wsag-neg:NegotiationRoleType"/>
    <xsd:element name="State"
          type="wsag-neg:NegotiationOfferStateType" />
    <xsd:any namespace="##other" processContents="lax"
          minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="NegotiationOfferStateType">
  <xsd:choice>
    <xsd:element name="Advisory"
          type="wsag-neg:InnerNegotiationStateType"/>
    <xsd:element name="Solicited"
          type="wsag-neg:InnerNegotiationStateType"/>
    <xsd:element name="Acceptable"
          type="wsag-neg:InnerNegotiationStateType"/>
```

```
      <xsd:element name="Rejected"
           type="wsag-neg:InnerNegotiationStateType"/>
    </xsd:choice>
  </xsd:complexType>

  <xsd:complexType name="InnerNegotiationStateType">
    <xsd:sequence>
        <xsd:any namespace="##other" processContents="lax"
             minOccurs="0" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="NegotiationExtensionType">
    <xsd:sequence>
        <xsd:element name="ResponderNegotiationEPR"
             type="wsa:EndpointReferenceType" minOccurs="0" />
        <xsd:element name="InitiatorNegotiationEPR"
             type="wsa:EndpointReferenceType" minOccurs="0" />
        <xsd:element name="NegotiationOfferContext"
             type="wsag-neg:NegotiationOfferContextType" minOccurs="1" />
        <xsd:any namespace="##other" minOccurs="0"
             processContents="lax"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="RenegotiationExtensionType">
    <xsd:sequence>
        <xsd:element name="ResponderNegotiationEPR"
             type="wsa:EndpointReferenceType" minOccurs="1" />
        <xsd:element name="InitiatorNegotiationEPR"
             type="wsa:EndpointReferenceType" minOccurs="0" />
```

```
          <xsd:element name="ResponderAgreementEPR"
               type="wsa:EndpointReferenceType" minOccurs="1" />
          <xsd:element name="NegotiationOfferContext"
               type="wsag-neg:NegotiationOfferContextType" minOccurs="1" />
          <xsd:any namespace="##other" processContents="lax"
               minOccurs="0" />
      </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

WS-Agreement Negotiation Factory WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsa="http://www.w3.org/2005/08/addressing"
    xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement"
    xmlns:wsag-neg="http://schemas.ogf.org/graap/2009/11/ws-agreement-
negotiation"
    xmlns:wsrf-rp="http://docs.oasis-open.org/wsrf/rp-2"
    xmlns:wsrf-bf="http://docs.oasis-open.org/wsrf/bf-2"
    xmlns:wsrf-rw="http://docs.oasis-open.org/wsrf/rw-2"
    xmlns:wsrf-rl="http://docs.oasis-open.org/wsrf/rl-2"
    xmlns:wsrf-rpw="http://docs.oasis-open.org/wsrf/rpw-2"
    targetNamespace="http://schemas.ogf.org/graap/2009/11/ws-agreement-
negotiation">

    <wsdl:import namespace="http://docs.oasis-open.org/wsrf/rw-2"
        location="http://docs.oasis-open.org/wsrf/rw-2.wsdl"/>

    <wsdl:types>
        <xs:schema
```

```
            targetNamespace="http://schemas.ogf.org/graap/2009/11/ws-
agreement-negotiation"

            xmlns:wsag-neg="http://schemas.ogf.org/graap/2009/11/ws-
agreement-negotiation"

            xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement"

            xmlns:wsa="http://www.w3.org/2005/08/addressing"

            elementFormDefault="qualified"

            attributeFormDefault="qualified">


            <xs:import namespace="http://www.w3.org/2005/08/addressing"

                schemaLocation="http://www.w3.org/2006/03/addressing/ws-
addr.xsd"/>


            <xs:import namespace="http://schemas.ggf.org/graap/2007/03/ws-
agreement"

                schemaLocation="agreement_types.xsd" />


            <xs:include schemaLocation="agreement_negotiation_types.xsd" />


            <xs:element name="InitiateNegotiationInput"

                type="wsag-neg:InitiateNegotiationInputType"/>

            <xs:complexType name="InitiateNegotiationInputType">

                <xs:sequence>

                    <xs:element ref="wsag-neg:NegotiationContext" />

                    <xs:element name="InitiatorNegotiationEPR"

                        type="wsa:EndpointReferenceType" minOccurs="0" />

                    <xs:element name="NoncriticalExtension"

                        type="wsag:NoncriticalExtensionType"

                        minOccurs="0" maxOccurs="unbounded" />

                    <xs:any namespace="##other" processContents="lax"

                        minOccurs="0" maxOccurs="unbounded" />

                </xs:sequence>
```

```
        </xs:complexType>

        <xs:element name="InitiateNegotiationOutput"
            type="wsag-neg:InitiateNegotiationOutputType"/>
        <xs:complexType name="InitiateNegotiationOutputType">
            <xs:sequence>
                <xs:element name="CreatedNegotiationEPR"
                    type="wsa:EndpointReferenceType"
                    minOccurs="1" maxOccurs="1" />
                <xs:any namespace="##other" processContents="lax"
                    minOccurs="0" maxOccurs="unbounded" />
            </xs:sequence>
        </xs:complexType>
    </xs:schema>
</wsdl:types>

<wsdl:message name="InitiateNegotiationInputMessage">
    <wsdl:part name="parameters"
        element="wsag-neg:InitiateNegotiationInput" />
</wsdl:message>

<wsdl:message name="InitiateNegotiationOuputMessage">
    <wsdl:part name="parameters"
        element="wsag-neg:InitiateNegotiationOutput" />
</wsdl:message>

<wsdl:message name="InitiateNegotiationFaultMessage">
    <wsdl:part name="fault" element="wsag:ContinuingFault"/>
</wsdl:message>

<wsdl:portType name="NegotiationFactory">
```

```
        <wsdl:operation name="InitiateNegotiation">

            <wsdl:input

                  message="wsag-neg:InitiateNegotiationInputMessage"/>

            <wsdl:output

                  message="wsag-neg:InitiateNegotiationOuputMessage"/>

            <wsdl:fault name="ResourceUnknownFault"

                  message="wsrf-rw:ResourceUnknownFault" />

            <wsdl:fault name="ResourceUnavailableFault"

                  message="wsrf-rw:ResourceUnavailableFault" />

            <wsdl:fault name="NegotiationInitiationFault"

                  message="wsag-neg:InitiateNegotiationFaultMessage" />

        </wsdl:operation>

    </wsdl:portType>

</wsdl:definitions>
```

WS-Agreement Negotiation WSDL

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"

    xmlns:xs="http://www.w3.org/2001/XMLSchema"

    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

    xmlns:wsa="http://www.w3.org/2005/08/addressing"

    xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement"

    xmlns:wsag-neg="http://schemas.ogf.org/graap/2009/11/ws-agreement-
negotiation"

    xmlns:wsrf-rp="http://docs.oasis-open.org/wsrf/rp-2"

    xmlns:wsrf-bf="http://docs.oasis-open.org/wsrf/bf-2"

    xmlns:wsrf-rw="http://docs.oasis-open.org/wsrf/rw-2"

    xmlns:wsrf-rl="http://docs.oasis-open.org/wsrf/rl-2"

    xmlns:wsrf-rpw="http://docs.oasis-open.org/wsrf/rpw-2"

    targetNamespace="http://schemas.ogf.org/graap/2009/11/ws-agreement-
negotiation">
```

```
    <wsdl:import namespace="http://docs.oasis-open.org/wsrf/rw-2"

        location="http://docs.oasis-open.org/wsrf/rw-2.wsdl"/>


    <wsdl:import namespace="http://docs.oasis-open.org/wsrf/rpw-2"

        location="http://docs.oasis-open.org/wsrf/rpw-2.wsdl" />


    <wsdl:types>

        <xs:schema

            targetNamespace="http://schemas.ogf.org/graap/2009/11/ws-
agreement-negotiation"

            xmlns:wsag-neg="http://schemas.ogf.org/graap/2009/11/ws-
agreement-negotiation"

            xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement"

            xmlns:wsa="http://www.w3.org/2005/08/addressing"

            elementFormDefault="qualified"

            attributeFormDefault="qualified">


            <xs:import namespace="http://schemas.ggf.org/graap/2007/03/ws-
agreement"

                schemaLocation="agreement_types.xsd" />


            <xs:include schemaLocation="agreement_negotiation_types.xsd" />


            <xs:element name="NegotiationProperties"

                type="wsag-neg:NegotiationPropertiesType" />

            <xs:complexType name="NegotiationPropertiesType">

                <xs:sequence>

                    <xs:element ref="wsag-neg:NegotiationContext" />

                    <xs:element ref="wsag-neg:NegotiationOffer"

                        minOccurs="0" maxOccurs="unbounded"/>

                </xs:sequence>

            </xs:complexType>
```

```
<xs:element name="NegotiateInput"
    type="wsag-neg:NegotiateInputType"/>
<xs:complexType name="NegotiateInputType">
    <xs:sequence>
        <xs:element ref="wsag-neg:NegotiationOffer"
            minOccurs="1" maxOccurs="unbounded" />
        <xs:any namespace="##other" processContents="lax"
            minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
</xs:complexType>

<xs:element name="NegotiateOutput"
    type="wsag-neg:NegotiateOutputType"/>
<xs:complexType name="NegotiateOutputType">
    <xs:sequence>
        <xs:element ref="wsag-neg:NegotiationCounterOffer"
            minOccurs="0" maxOccurs="unbounded" />
        <xs:any namespace="##other" processContents="lax"
            minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
</xs:complexType>

<xs:element name="TerminateInput"
    type="wsag-neg:TerminateInputType" />
<xs:complexType name="TerminateInputType">
    <xs:sequence>
        <xs:any processContents="lax" namespace="##other"
            minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
</xs:complexType>
```

```
        <xs:element name="TerminateResponse"
            type="wsag-neg:TerminateOutputType" />
        <xs:complexType name="TerminateOutputType" />
    </xs:schema>
</wsdl:types>

<wsdl:message name="NegotiateInputMessage">
    <wsdl:part name="parameters"
        element="wsag-neg:NegotiateInput" />
</wsdl:message>

<wsdl:message name="NegotiateOuputMessage">
    <wsdl:part name="parameters"
        element="wsag-neg:NegotiateOutput" />
</wsdl:message>

<wsdl:message name="NegotiationFaultMessage">
    <wsdl:part name="fault"
        element="wsag:ContinuingFault"/>
</wsdl:message>

<wsdl:message name="TerminateNegotiationInputMessage">
    <wsdl:part name="parameters"
        element="wsag-neg:TerminateInput" />
</wsdl:message>

<wsdl:message name="TerminateNegotiationOuputMessage">
    <wsdl:part name="parameters"
        element="wsag-neg:TerminateResponse" />
</wsdl:message>
```

```
    <wsdl:portType name="Negotiation"
          wsrf-rp:ResourceProperties="wsag-neg:NegotiationProperties">

        <wsdl:operation name="Negotiate">
            <wsdl:input
                message="wsag-neg:NegotiateInputMessage" />
            <wsdl:output
                message="wsag-neg:NegotiateOuputMessage" />
            <wsdl:fault name="ResourceUnknownFault"
                message="wsrf-rw:ResourceUnknownFault" />
            <wsdl:fault name="ResourceUnavailableFault"
                message="wsrf-rw:ResourceUnavailableFault" />
            <wsdl:fault name="NegotiationFault"
                message="wsag-neg:NegotiationFaultMessage" />
        </wsdl:operation>
        <wsdl:operation name="Terminate">
            <wsdl:input
                message="wsag-neg:TerminateNegotiationInputMessage" />
            <wsdl:output
                message="wsag-neg:TerminateNegotiationOuputMessage" />
            <wsdl:fault name="ResourceUnknownFault"
                message="wsrf-rw:ResourceUnknownFault" />
            <wsdl:fault name="ResourceUnavailableFault"
                message="wsrf-rw:ResourceUnavailableFault" />
        </wsdl:operation>
    </wsdl:portType>
</wsdl:definitions>
```

WS-Agreement Negotiation Advertisement WSDL

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"

    xmlns:xs="http://www.w3.org/2001/XMLSchema"

    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

    xmlns:wsa="http://www.w3.org/2005/08/addressing"

    xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement"

    xmlns:wsag-neg="http://schemas.ogf.org/graap/2009/11/ws-agreement-
negotiation"

    xmlns:wsrf-rp="http://docs.oasis-open.org/wsrf/rp-2"

    xmlns:wsrf-bf="http://docs.oasis-open.org/wsrf/bf-2"

    xmlns:wsrf-rw="http://docs.oasis-open.org/wsrf/rw-2"

    xmlns:wsrf-rl="http://docs.oasis-open.org/wsrf/rl-2"

    xmlns:wsrf-rpw="http://docs.oasis-open.org/wsrf/rpw-2"

    targetNamespace="http://schemas.ogf.org/graap/2009/11/ws-agreement-
negotiation">


    <wsdl:import namespace="http://docs.oasis-open.org/wsrf/rw-2"

        location="http://docs.oasis-open.org/wsrf/rw-2.wsdl"/>


    <wsdl:import namespace="http://docs.oasis-open.org/wsrf/rpw-2"

        location="http://docs.oasis-open.org/wsrf/rpw-2.wsdl" />


    <wsdl:types>

        <xs:schema

            targetNamespace="http://schemas.ogf.org/graap/2009/11/ws-
agreement-negotiation"

            xmlns:wsag-neg="http://schemas.ogf.org/graap/2009/11/ws-
agreement-negotiation"

            xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement"

            xmlns:wsa="http://www.w3.org/2005/08/addressing"

            elementFormDefault="qualified"

            attributeFormDefault="qualified">
```

```
            <xs:import namespace="http://schemas.ggf.org/graap/2007/03/ws-
agreement"
                       schemaLocation="agreement_types.xsd" />

            <xs:include schemaLocation="agreement_negotiation_types.xsd" />

            <xs:element name="AdvertiseInput"
                type="wsag-neg:AdvertiseInputType"/>
            <xs:complexType name="AdvertiseInputType">
                <xs:sequence>
                    <xs:element ref="wsag-neg:NegotiationOffer"
                        minOccurs="1" maxOccurs="unbounded" />
                    <xs:any namespace="##other" processContents="lax"
                        minOccurs="0" maxOccurs="unbounded" />
                </xs:sequence>
            </xs:complexType>

            <xs:element name="AdvertiseOutput"
                type="wsag-neg:AdvertiseOutputType"/>
            <xs:complexType name="AdvertiseOutputType" />
        </xs:schema>
    </wsdl:types>

    <wsdl:message name="AdvertiseInputMessage">
        <wsdl:part name="parameters"
            element="wsag-neg:AdvertiseInput" />
    </wsdl:message>

    <wsdl:message name="AdvertiseOuputMessage">
        <wsdl:part name="parameters"
            element="wsag-neg:AdvertiseOutput" />
```

```
    </wsdl:message>

    <wsdl:message name="AdvertiseFaultMessage">
        <wsdl:part name="fault"
            element="wsag:ContinuingFault"/>
    </wsdl:message>

    <wsdl:portType name="Advertise">
        <wsdl:operation name="Advertise">
            <wsdl:input
                message="wsag-neg:AdvertiseInputMessage" />
            <wsdl:output
                message="wsag-neg:AdvertiseOuputMessage" />
            <wsdl:fault name="ResourceUnknownFault"
                message="wsrf-rw:ResourceUnknownFault" />
            <wsdl:fault name="ResourceUnavailableFault"
                message="wsrf-rw:ResourceUnavailableFault" />
            <wsdl:fault name="Advertise"
                message="wsag-neg:AdvertiseFaultMessage" />
        </wsdl:operation>
    </wsdl:portType>
</wsdl:definitions>
```

Agreement Template Example

This example shows an agreement template that consists of a set of service terms, service properties, and guarantee terms. The service term describes a computational service and the resources required during service provisioning. The service properties define a set of variables for resolving the actual service state, the requested CPU speed, and the actual CPU speed. These variables are used in the guarantee term in order to define qualifying condition and service level objective of a CPU speed guarantee.

Moreover, the template contains two complex creation constraints that restrict structure and value spaces for valid agreement offers. The *AgreementConstraint* restricts the structure and value spaces of elements that are defined in the WS-Agreement namespace. It defines amongst

others the structure and values of the agreement context and can easily be extended to also restrict the structure and values of service properties and guarantee terms in order to prevent that an agreement responder can change the content of these elements. The *ServiceTerm-Constraint* restricts the content of the service description term, which is a JSDL document.

```
The methods to validate creation constraints, to monitor service term
states, to evaluate guarantee term states, and to account guarantees
accordingly are described in Chapter 3nder>

       CN=ACME Corporation,DC=ACME,DC=COM

    </wsag:AgreementResponder>

    <wsag:ServiceProvider>AgreementResponder</wsag:ServiceProvider>

    <wsag:ExpirationTime>2012-01-01T00:00:00+02:00</wsag:ExpirationTime>

    <wsag:TemplateId>1</wsag:TemplateId>

    <wsag:TemplateName>ComputationalServiceTemplate</wsag:TemplateName>

  </wsag:Context>

  <wsag:Terms>

    <wsag:All>

      <wsag:ServiceDescriptionTerm

            wsag:Name="ComputeSDT"

            wsag:ServiceName="ComputationalService">

        <jsdl:JobDefinition

         xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl">

          <jsdl:JobDescription>

            <jsdl:Application>

              <jsdl:ApplicationName>ACME_Webservice</jsdl:ApplicationName>

              <jsdl:ApplicationVersion>1.0</jsdl:ApplicationVersion>

              <jsdl:Description>The ACME Webservice.</jsdl:Description>

            </jsdl:Application>

            <jsdl:Resources>

              <jsdl:IndividualCPUSpeed>

                <jsdl:LowerBoundedRange>2.0E9</jsdl:LowerBoundedRange>

              </jsdl:IndividualCPUSpeed>

              <jsdl:IndividualCPUCount>
```

```
            <jsdl:Exact>2.0</jsdl:Exact>
          </jsdl:IndividualCPUCount>
          <jsdl:TotalResourceCount>
            <jsdl:Exact>16.0</jsdl:Exact>
          </jsdl:TotalResourceCount>
        </jsdl:Resources>
      </jsdl:JobDescription>
    </jsdl:JobDefinition>
  </wsag:ServiceDescriptionTerm>
  <wsag:ServiceProperties
        wsag:Name="Service_Properties"
        wsag:ServiceName="ComputationalService">
    <wsag:VariableSet>
      <wsag:Variable wsag:Name="SERVICE_STATE" wsag:Metric="xs:string">
        <wsag:Location>
         declare namespace
         wsag='http://schemas.ggf.org/graap/2007/03/ws-agreement';
         $this/wsag:Terms/wsag:All/wsag:ServiceDescriptionTerm
         [@wsag:Name='ComputeSDT']/wsag:State
      </wsag:Location>
      </wsag:Variable>
      <wsag:Variable wsag:Name="REQ_CPU_SPEED" wsag:Metric="xs:double">
        <wsag:Location>declare namespace
         jsdl='http://schemas.ggf.org/jsdl/2005/11/jsdl';
         declare namespace
         wsag='http://schemas.ggf.org/graap/2007/03/ws-agreement';
         $this/wsag:Terms/wsag:All/wsag:ServiceDescriptionTerm
         [@wsag:Name = 'ComputeSDT']/jsdl:JobDefinition
         /jsdl:JobDescription/jsdl:Resources/jsdl:IndividualCPUSpeed
         /jsdl:LowerBoundedRange
      </wsag:Location>
```

```
          </wsag:Variable>
          <wsag:Variable wsag:Name="ACT_CPU_SPEED" wsag:Metric="xs:double">
            <wsag:Location>
             declare namespace
             jsdl='http://schemas.ggf.org/jsdl/2005/11/jsdl';
             declare namespace
             wsag='http://schemas.ggf.org/graap/2007/03/ws-agreement';
             $this/wsag:ServiceTermState[@wsag:termName='ComputeSDT']
             /jsdl:JobDefinition/jsdl:JobDescription/jsdl:Resources
             /jsdl:IndividualCPUSpeed/jsdl:Exact
           </wsag:Location>
          </wsag:Variable>
        </wsag:VariableSet>
      </wsag:ServiceProperties>
      <wsag:GuaranteeTerm wsag:Name="CPU_SPEED_GUARANTEE">
        <wsag:ServiceScope wsag:ServiceName="ComputationalService"/>
        <wsag:QualifyingCondition>
             SERVICE_STATE eq 'Ready'
        </wsag:QualifyingCondition>
        <wsag:ServiceLevelObjective>
          <wsag:KPITarget>
            <wsag:KPIName>CPU SPEED</wsag:KPIName>
            <wsag:CustomServiceLevel>
                 ACT_CPU_SPEED ge REQ_CPU_SPEED
            </wsag:CustomServiceLevel>
          </wsag:KPITarget>
        </wsag:ServiceLevelObjective>
        <wsag:BusinessValueList>
          <wsag:Penalty>
            <wsag:AssessmentInterval>
              <wsag:TimeInterval>P5M</wsag:TimeInterval>
```

```
          </wsag:AssessmentInterval>
          <wsag:ValueUnit>€</wsag:ValueUnit>
          <wsag:ValueExpression>5</wsag:ValueExpression>
        </wsag:Penalty>
        <wsag:Reward>
          <wsag:AssessmentInterval>
            <wsag:TimeInterval>P5M</wsag:TimeInterval>
          </wsag:AssessmentInterval>
          <wsag:ValueUnit>€</wsag:ValueUnit>
          <wsag:ValueExpression>10</wsag:ValueExpression>
        </wsag:Reward>
      </wsag:BusinessValueList>
    </wsag:GuaranteeTerm>
  </wsag:All>
</wsag:Terms>
<wsag:CreationConstraints>
  <!--
   This is an example of a complex creation constraint that restricts
   the structure of incoming agreement offers and defines element values
   that must not be changed in an offer, such as AgreementResponder
   identification and the ServiceProvider role. Attributes such as
   name or service name attribute in a ServiceDescritionTerm can be
   defined in a similar way.
   -->
  <wsag:Item wsag:Name="AgreementConnstraint">
    <wsag:Location>
        declare namespace
        wsag='http://schemas.ggf.org/graap/2007/03/ws-agreement';
        $this/wsag:AgreementOffer
    </wsag:Location>
    <wsag:ItemConstraint>
```

```
        <xs:sequence xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement">
          <xs:element name="Name" type="xs:string"
                      fixed="ComputationalServiceTemplate"/>
          <xs:element name="Context">
            <xs:complexType>
              <xs:complexContent>
                <xs:restriction base="wsag:AgreementContextType">
                  <!--
                  Simple values can easily be restricted by using e.g. the
                  fixed attribute. In this example the value of the
                  AgreementResponder and the ServiceProvider can not be
                  changed by the agreement initiator.
                   -->
                  <xs:sequence>
                    <xs:element name="AgreementInitiator" type="xs:anyType"
/>
                    <xs:element name="AgreementResponder" type="xs:anyType"
                     fixed="CN=ACME Corporation,DC=ACME,DC=COM"/>
                    <xs:element name="ServiceProvider"
                                type="wsag:AgreementRoleType"
                                fixed="AgreementResponder"/>
                    <xs:element name="ExpirationTime" type="xs:dateTime" />
                    <xs:element name="TemplateId" type="xs:string"
                                fixed="1"/>
                    <xs:element name="TemplateName" type="xs:string"
                                fixed="ComputationalServiceTemplate"/>
                  </xs:sequence>
                </xs:restriction>
              </xs:complexContent>
```

```
          </xs:complexType>
        </xs:element>
        <xs:element name="Terms">
          <xs:complexType>
            <xs:complexContent>
              <xs:restriction base="wsag:TermTreeType">
                <xs:sequence>
                  <xs:element name="All">
                    <xs:complexType>
                      <xs:complexContent>
                        <xs:restriction base="wsag:TermCompositorType">
                <!--
                  This constraint does not guarantee that exactly one
                  ServiceDescriptionTerm, one ServicePropertiesTerm, and
                  one GuaranteeTerm is selected. Therefore, an additional
                  Schematron-Constraint would be required.
                -->
                          <xs:choice minOccurs="3" maxOccurs="3">
                            <xs:element name="ServiceDescriptionTerm">
                              <xs:complexType>
                                <xs:complexContent>
                                  <xs:restriction
                                  base="wsag:ServiceDescriptionTermType">
                                    <xs:sequence>
                <!--
                  The JobDefinition element is defined in another
                  namespace. It must therefore be restricted by a
                  separate item constraint. The restriction of elements
                  defined in different namespaces is not possible in the
                  same item constraint.
                -->
```

```
                                        <xs:element
                                         ref="jsdl:JobDefinition"/>
                                      </xs:sequence>
                                    </xs:restriction>
                                  </xs:complexContent>
                                </xs:complexType>
                              </xs:element>
                              <xs:element name="ServiceProperties">
                                <xs:complexType>
                                  <xs:complexContent>
                                    <xs:restriction
                                     base="wsag:ServicePropertiesType">
                                      <xs:sequence>
                  <!--
                    The structure and the values of the elements in this
                    namespace can further be restricted as shown above.
                    The definition of element values is shown in the
                    restriction of the AgreementResponder and the
                    ServiceProvider elements of the agreement context.
                   -->
                                        <xs:element name="VariableSet"
                                         type="wsag:VariableSetType"/>
                                      </xs:sequence>
                                    </xs:restriction>
                                  </xs:complexContent>
                                </xs:complexType>
                              </xs:element>
                              <xs:element name="GuaranteeTerm"
                               type="wsag:GuaranteeTermType" />
                            </xs:choice>
                          </xs:restriction>
```

```
                    </xs:complexContent>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
            </xs:restriction>
          </xs:complexContent>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </wsag:ItemConstraint>
</wsag:Item>
<!--
 This is an example of a service term constraint. This constraint does
 not only  restrict the structure of the JSDL service term. It also
 restricts the value space of some of the elements in the JSDL
 document.
 -->
<wsag:Item wsag:Name="ServiceTermConstraint">
  <wsag:Location>
   declare namespace
   jsdl='http://schemas.ggf.org/jsdl/2005/11/jsdl';
   declare namespace
   wsag='http://schemas.ggf.org/graap/2007/03/ws-agreement';
   $this/wsag:AgreementOffer/wsag:Terms/wsag:All/wsag:ExactlyOne
   /wsag:ServiceDescriptionTerm/jsdl:JobDefinition
  </wsag:Location>
  <wsag:ItemConstraint>
    <xs:sequence xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl"
                 xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="JobDescription">
        <xs:complexType>
```

```xml
        <xs:complexContent>
          <xs:restriction base="jsdl:JobDescription_Type">
            <xs:sequence>
              <xs:element ref="jsdl:Application" />
              <xs:element name="Resources" >
                <xs:complexType>
                  <xs:complexContent>
                    <xs:restriction base="jsdl:Resources_Type">
                      <xs:sequence>
                        <xs:element ref="jsdl:IndividualCPUSpeed"/>
                        <xs:element ref="jsdl:IndividualCPUCount"/>
                        <!--
                         The value of TotalResourceCount must exactly
                         be specified by the agreement initiator.
                         Valid values are 16, 32, 64, and 128.
                         -->
                        <xs:element name="TotalResourceCount">
                          <xs:complexType>
                            <xs:complexContent>
                              <xs:restriction
                               base="jsdl:RangeValue_Type">
                                <xs:sequence>
                                  <xs:element name="Exact">
                                    <xs:complexType>
                                     <xs:simpleContent>
                                       <xs:restriction
                                        base="jsdl:Exact_Type">
                                        <xs:enumeration value="16"/>
                                        <xs:enumeration value="32"/>
                                        <xs:enumeration value="64"/>
                                        <xs:enumeration value="128"/>
```

```
                                              </xs:restriction>
                                            </xs:simpleContent>
                                          </xs:complexType>
                                        </xs:element>
                                      </xs:sequence>
                                    </xs:restriction>
                                  </xs:complexContent>
                                </xs:complexType>
                              </xs:element>
                            </xs:sequence>
                          </xs:restriction>
                        </xs:complexContent>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:restriction>
              </xs:complexContent>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </wsag:ItemConstraint>
    </wsag:Item>
  </wsag:CreationConstraints>
</wsag:Template>
```

Agreement Properties Document Example

This example shows an agreement properties document that contains domain specific service monitoring information in the service term states. This domain specific information is used by the WSAG4J framework to determine the state of a guarantee. In this particular example only one guarantee term is defined. The guarantee specifies a penalty and a reward in order to assure that the provided resources meet a user specified lower bound for the CPU speed. In contrast to an agreement template all variables defined in the service properties can be resolved within the

agreement properties document. Domain specific extensions are used in the service term states to express more detailed information on the service delivery process.

```
<wsag:AgreementProperties

  xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement">

  <wsag:Name>ComputationalService</wsag:Name>

  <wsag:AgreementId>1</wsag:AgreementId>

  <wsag:Context>

    <wsag:AgreementInitiator/>

    <wsag:AgreementResponder>

       CN=ACME Corporation,DC=ACME,DC=COM

    </wsag:AgreementResponder>

    <wsag:ServiceProvider>AgreementResponder</wsag:ServiceProvider>

    <wsag:ExpirationTime>2012-01-01T00:00:00+02:00</wsag:ExpirationTime>

    <wsag:TemplateId>1</wsag:TemplateId>

    <wsag:TemplateName>ComputationalServiceTemplate</wsag:TemplateName>

  </wsag:Context>

  <wsag:Terms>

    <wsag:All>

      <wsag:ServiceDescriptionTerm

            wsag:Name="ComputeSDT"

            wsag:ServiceName="ComputationalService">

        <jsdl:JobDefinition

         xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl">

          <jsdl:JobDescription>

            <jsdl:Application>

              <jsdl:ApplicationName>ACME_Webservice</jsdl:ApplicationName>

              <jsdl:ApplicationVersion>1.0</jsdl:ApplicationVersion>

              <jsdl:Description>The ACME Webservice.</jsdl:Description>

            </jsdl:Application>

            <jsdl:Resources>

              <jsdl:IndividualCPUSpeed>
```

```
            <jsdl:LowerBoundedRange>2.0E9</jsdl:LowerBoundedRange>
          </jsdl:IndividualCPUSpeed>
          <jsdl:IndividualCPUCount>
            <jsdl:Exact>2.0</jsdl:Exact>
          </jsdl:IndividualCPUCount>
          <jsdl:TotalResourceCount>
            <jsdl:Exact>16.0</jsdl:Exact>
          </jsdl:TotalResourceCount>
        </jsdl:Resources>
      </jsdl:JobDescription>
    </jsdl:JobDefinition>
  </wsag:ServiceDescriptionTerm>
  <wsag:ServiceProperties
        wsag:Name="Service_Properties"
        wsag:ServiceName="ComputationalService">
    <wsag:VariableSet>
      <wsag:Variable wsag:Name="SERVICE_STATE" wsag:Metric="xs:string">
        <wsag:Location>
         declare namespace
         wsag='http://schemas.ggf.org/graap/2007/03/ws-agreement';
         $this/wsag:Terms/wsag:All/wsag:wsag:ServiceTermState
         [@wsag:Name='ComputeSDT']/wsag:State
      </wsag:Location>
      </wsag:Variable>
      <wsag:Variable wsag:Name="REQ_CPU_SPEED" wsag:Metric="xs:double">
        <wsag:Location>declare namespace
         jsdl='http://schemas.ggf.org/jsdl/2005/11/jsdl';
         declare namespace
         wsag='http://schemas.ggf.org/graap/2007/03/ws-agreement';
         $this/wsag:Terms/wsag:All/wsag:wsag:ServiceTermState
          [@wsag:Name = 'ComputeSDT']/jsdl:JobDefinition
```

```
          /jsdl:JobDescription/jsdl:Resources/jsdl:IndividualCPUSpeed
          /jsdl:LowerBoundedRange
      </wsag:Location>
      </wsag:Variable>
      <wsag:Variable wsag:Name="ACT_CPU_SPEED" wsag:Metric="xs:double">
        <wsag:Location>
         declare namespace
         jsdl='http://schemas.ggf.org/jsdl/2005/11/jsdl';
         declare namespace
         wsag='http://schemas.ggf.org/graap/2007/03/ws-agreement';
         $this/wsag:ServiceTermState[@wsag:termName='ComputeSDT']
         /jsdl:JobDefinition/jsdl:JobDescription/jsdl:Resources
         /jsdl:IndividualCPUSpeed/jsdl:Exact
       </wsag:Location>
      </wsag:Variable>
    </wsag:VariableSet>
  </wsag:ServiceProperties>
  <wsag:GuaranteeTerm wsag:Name="CPU_SPEED_GUARANTEE">
    <wsag:ServiceScope wsag:ServiceName="ComputationalService"/>
    <wsag:QualifyingCondition>
         SERVICE_STATE eq 'Ready'
    </wsag:QualifyingCondition>
    <wsag:ServiceLevelObjective>
      <wsag:KPITarget>
        <wsag:KPIName>CPU SPEED</wsag:KPIName>
        <wsag:CustomServiceLevel>
             ACT_CPU_SPEED ge REQ_CPU_SPEED
        </wsag:CustomServiceLevel>
      </wsag:KPITarget>
    </wsag:ServiceLevelObjective>
    <wsag:BusinessValueList>
```

```xml
          <wsag:Penalty>
            <wsag:AssessmentInterval>
              <wsag:TimeInterval>P5M</wsag:TimeInterval>
            </wsag:AssessmentInterval>
            <wsag:ValueUnit>€</wsag:ValueUnit>
            <wsag:ValueExpression>5</wsag:ValueExpression>
          </wsag:Penalty>
          <wsag:Reward>
            <wsag:AssessmentInterval>
              <wsag:TimeInterval>P5M</wsag:TimeInterval>
            </wsag:AssessmentInterval>
            <wsag:ValueUnit>€</wsag:ValueUnit>
            <wsag:ValueExpression>10</wsag:ValueExpression>
          </wsag:Reward>
        </wsag:BusinessValueList>
      </wsag:GuaranteeTerm>
    </wsag:All>
  </wsag:Terms>
  <wsag:AgreementState>
    <wsag:State>Observed</wsag:State>
  </wsag:AgreementState>
  <wsag:GuaranteeTermState wsag:termName="CPU_SPEED_GUARANTEE">
    <wsag:State>Fulfilled</wsag:State>
  </wsag:GuaranteeTermState>
  <wsag:ServiceTermState wsag:termName="ComputeSDT">
    <wsag:State>Ready</wsag:State>
    <jsdl:JobDefinition
     xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl">
      <jsdl:JobDescription>
        <jsdl:Application>
          <jsdl:ApplicationName>ACME_Webservice</jsdl:ApplicationName>
```

```
          <jsdl:ApplicationVersion>1.0</jsdl:ApplicationVersion>
          <jsdl:Description>An ACME Webservice.</jsdl:Description>
        </jsdl:Application>
        <jsdl:Resources>
          <jsdl:IndividualCPUSpeed>
            <jsdl:Exact>2.0E9</jsdl:Exact>
          </jsdl:IndividualCPUSpeed>
          <jsdl:IndividualCPUCount>
            <jsdl:Exact>2.0</jsdl:Exact>
          </jsdl:IndividualCPUCount>
          <jsdl:TotalResourceCount>
            <jsdl:Exact>16.0</jsdl:Exact>
          </jsdl:TotalResourceCount>
        </jsdl:Resources>
      </jsdl:JobDescription>
    </jsdl:JobDefinition>
  </wsag:ServiceTermState>
</wsag:AgreementProperties>
```

Lebenslauf

Persönliche Daten

Name:	Oliver Wäldrich
Anschrift:	Bonner Str. 158
	53757 Sankt Augustin
Geburtstag:	24. Juni 1975
Geburtsort:	Rüdersdorf b. Berlin
Nationalität:	deutsch

Beruflicher Werdegang

seit 10/2005	Wissenschaftlicher Mitarbeiter am Fraunhofer Institut SCAI
10/2003 - 10/2005	Master of Science in Computer Science (MSc)
	Fachhochschule Bonn-Rhein-Sieg
10/2000 - 09/2003	Systemintegrator, T-Systems International GmbH
10/1995 - 10/2000	Diplom-Wirtschaftsinformatiker (FH)
	Technische Fachhochschule Wildau
06/1995	Allgemeine Hochschulreife (Abitur)
	Geschwister-Scholl-Gymnasium, Fürstenwalde/Spree