Service Level Agreement Mediation, Negotiation and Evaluation for Cloud Services in Intercloud Environments

vorgelegt von
M.Sc. Dipl.-Ing. (FH)
Alexander Stanik
geb. in Berlin

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr.-Ing. Stefan Tai
Gutachter:	Prof. Dr. habil. Odej Kao
	Prof. Dr.-Ing. Gero Mühl
	Prof. Dr. Matthias Hovestadt

Tag der wissenschaftlichen Aussprache: 11. April 2016

Berlin 2016

Acknowledgement

I would like to thank all the people who supported me during my studies and made this PhD thesis possible. I especially want to thank my advisor Odej Kao for his guidance and inviting me to work in his group. Thanks for making this job probably the best in my life. Special thanks also go to the entire working group "Complex and Distributed IT Systems" at TU Berlin for the awesome past five years. I also thank my former colleague Matthias Hovestadt for his guidance for finding this topic of my PhD thesis. And also many thanks go to António Cruz, Gregory Katsaros, Mareike Höger, Andreas Kliem, Marc Körner, Florian Feigenbutz, Fridtjof Sander, Oliver Wäldrich, Andreas Weiss and Wolfgang Ziegler for insightful discussions about the research.

A special thanks to my family for their support during all the time. Words cannot express how grateful I am to my mother Irene and my father Robert for all of the sacrifices that you've made on my behalf. You educated me and have made me to the person who I am today. Thanks for always offering support and love, moreover, helping me whenever I needed you.

Finally, special thanks goes to my dear wife Jeanette and my kids Maximilian, Charlotte, and Anastasia for their endless love and patience especially during writing this thesis. You spent sleepless nights with and unfortunately also sometimes without me. You were always my support in the moments when there was no one to answer my queries. I married the best person out there for me. These past several years have not been an easy ride, both academically and personally, but you and our kids were always there and unconditionally loved me during my good and bad times. Thanks!

Abstract

The management of today's service level agreements on the cloud market is often nontransparent and inefficient for the customer and the provider. However, service level agreements are crucial for establishing trust between all participants, especially for companies whose success depends on the purchased cloud service with its appropriated service quality. This thesis aims to improve the current situation on the cloud market in terms of providing architectural, methodological, and functional approaches with standard-based protocol specifications for dynamic and situational service level agreement management. Therefore, this thesis analyses and presents current service level agreement standards and existing implementations. Based on these results, a set of concepts and approaches for agreement and service mediation are developed and experimentally evaluated.

However, all of these existing approaches either target a specific community or implement broker services with just a closed product portfolio for services delivered by a small set of providers. In order to establish a trusted provider overlay network as global marketplace for cloud services, where all participants can federate by nature, expose their product offer, and reach a wider customer community, this thesis introduces a novel concept for intercloud agreement mediation. The Agreement-Mediators in this global provider network act as neutral and autonomous entities in the cloud market and rise trust between independent providers and between providers and customers. The language and protocol for searching, negotiating, and establishing service level agreements is specified, developed, and evaluated. The outcomes of this thesis aim to increase efficiency and effectiveness for discovering cloud services of independent providers and the management of service level agreements established for each business relationship. The experimental results show significant improvements in terms of efficiency, scalability, and flexibility.

Contents

1. Introduction

Contents

Over the last years a solid technical foundation on cloud computing was developed ranging from resource allocation on the Infrastructure-as-a-Service (IaaS) layer, job and application execution on the Platform-as-a-Service (PaaS) layer, to software and service usage on the Software-as-a-Service (SaaS) layer [1] [2] [3]. This foundation allows the access of resources according to the pay-as-you-go principle. At this, the resources may be geographically distributed all around the world and users can access them remotely without the need for any upfront capital expenses, long-term commitment, or significant provisioning delay. For realizing this cloud concept, an architecture stack consisting of IaaS, PaaS and SaaS has been developed, where IaaS provides the fundament with provision of virtualized resources that can be used e.g. by upper layer services [4]. For example, users are able to request virtual machines which then can be used almost like regular physical computers whereupon the user is free to install custom operating systems, libraries or applications [5]. Virtualization is the powerful enabling technology in this space, allowing for flexible and on-demand provisioning of IT resources from the user's perspective, but also simplifying the management of the data center for the cloud operator [6].

Clouds are often provided within a data center of a single institution. Here, a cloud middleware manages the access to virtualized resources through a cloud interface that can be a web service Application Programming Interface (API) or a more user-friendly web application. Focusing on the IaaS layer, the resources that are virtualized and sold to the paying customer can be classified into the following categories: compute, storage and network. All of these resources appear to be physical, but they are actually virtualized by making use of hypervisors in the case for virtual machines. While the physical infrastructure and installation rarely changes, the virtually delivered resources are changing permanently [7].

Because of the novelty of the cloud paradigm, many companies started working on their own proprietary standards for virtual machine configurations, their associated file formats, the application packaging and its deployment through proprietary interfaces [8]. This lack of interoperability has been solved by standards like the Open Virtualization Format (OVF) [9], the Open

Cloud Computing Interface (OCCI) [10], and the Topology and Orchestration Specification for Cloud Applications (TOSCA) [11].

These open standards mentioned above address interoperability and portability issues existing in the cloud context. However, these standards are used to deploy, orchestrate or in general enable the access to a single cloud middleware at a time. In particular, if customers would like to migrate a set of virtual machines or a clustered application, these standards can be used to take out the individual installation in a standardized packaging format. and is then be able to This bundle can then be redeploy in another cloud of the customer's choice. This is not the sole issue that needs to be improved in terms of migration. The selection of a public cloud provider that in turn have not only to win the business confidence of the customer, but should also fulfill the customer's needs, is much more complicated. In fact, the customers have a wide choice of cloud services offered with different pricing conditions and different service levels by a variety of commercial cloud providers that are distributed around the world.

1.1. Problem Definition

The way legal contracts and Service Level Agreements (SLA) between cloud service providers and cloud service consumers are established and managed is currently far from being ideal from the customer's point of view [12]. This topic naturally gains crucial importance for customers being companies whose success depends (even partially) on the advertised Quality of Service (QoS) that the service provider delivers. Three facts documented by [13] show exemplarily the gap in terms of SLAs on the cloud market today: inflexible term and price conditions, nontransparent verification of advertised service quality, and inflexible expression of SLAs.

Cloud contracts and SLAs can exist in two forms: off-the-shelf and negotiated. Common off-the-shelf contracts typically over-favor cloud service providers significantly by offering a small set of guarantees (e.g. availability in percent) with low compensations in case of failure (e.g. percentage service credit based on the monthly fee) bounded to unlikely conditions (e.g. every instance has to fail simultaneously) [14] [15] [16] [17] [18] [19] [20]. Negotiated contracts are reserved for an elite class of customers having enough bargaining power, gained through potential financial or reputational benefits expected by the provider as a consequence of an agreement, to force a service provider away from its standard terms [13]. However, such negotiation is performed individually between people and focuses primarily on the price. Although negotiations bring better price conditions, customers are often unhappy with negotiated contracts, because important terms like liability are often excluded by service providers [13]. In general, cloud service customers have no chance to express on the provider's standard interface which service quality is required and which level of assurance they want.

If the parties or rather the customer agrees to the service conditions, the provider should deliver the service according to the advertised quality. In fact, the customers have to trust the provider that the service is delivered as agreed on or the customer has to monitor the service quality by himself. Anyway, the responsibility of monitoring the delivered service quality is on the customer's part [14] [15] [16]. This fact conflicts with the general purpose of cloud computing services, which is to avoid administrative overhead on the consumer's side [21]. Furthermore, each service provider has its own requirements and dictates its own protocol, which the service consumer has to follow in order to report an agreement violation and to receive a compensation [14] [15] [16] [17] [18].

Last but not least, many service providers cannot give more definitive guarantees for their services to a wide range of customers as in the current form of expressing cloud contracts. That is mostly formal language written by lawyers [14] [15] [16] [17] [18] [19] [20] [13], which neither respects the actual resource utilization of the provider nor leaves room for any approach to the customer's requirements. For solving these problems, cloud service providers need an instrument that gives them the ability to adjust their advertisements instantaneously depending on their current state and automatically in a machine-readable form. Thus, customized service offers with corresponding service levels could be delivered to paying customers.

1.2. Contribution

This thesis presents an approach of an autonomous API-based *Agreement-Mediator* as a third party in the cloud market that provides a way to source out affairs regarding the management of SLAs into a neutral zone without taking the consumer's or the provider's side. Thus the *Agreement-Mediator* adds an additional transitive relationship between customers and providers without breaking or directly affecting any existing one, like visualized in figure 1.1. The *Agreement-Mediator's* main purposes are (a) advertising services with individual quality levels and guarantees that a provider is willing and able to deliver; (b) giving consumers the opportunity to discover and to easily compare offered SLAs of providers; (c) registering declarations of mutual intentions for business relationships in the form of established SLAs; (d) offering services to feed in monitoring data of participating parties of an agreement for automatic compliance verification of SLA terms; and (e) notifying participating parties about agreement-related events like creation, violation or termination. Thus, the overhead for managing SLAs between consumers and providers is minimized for both and a higher trust is established by mediation of an external entity. However, all obligations associated with an agreement including the collection of data for compliance assessment remain in the responsibilities of consumers and providers.

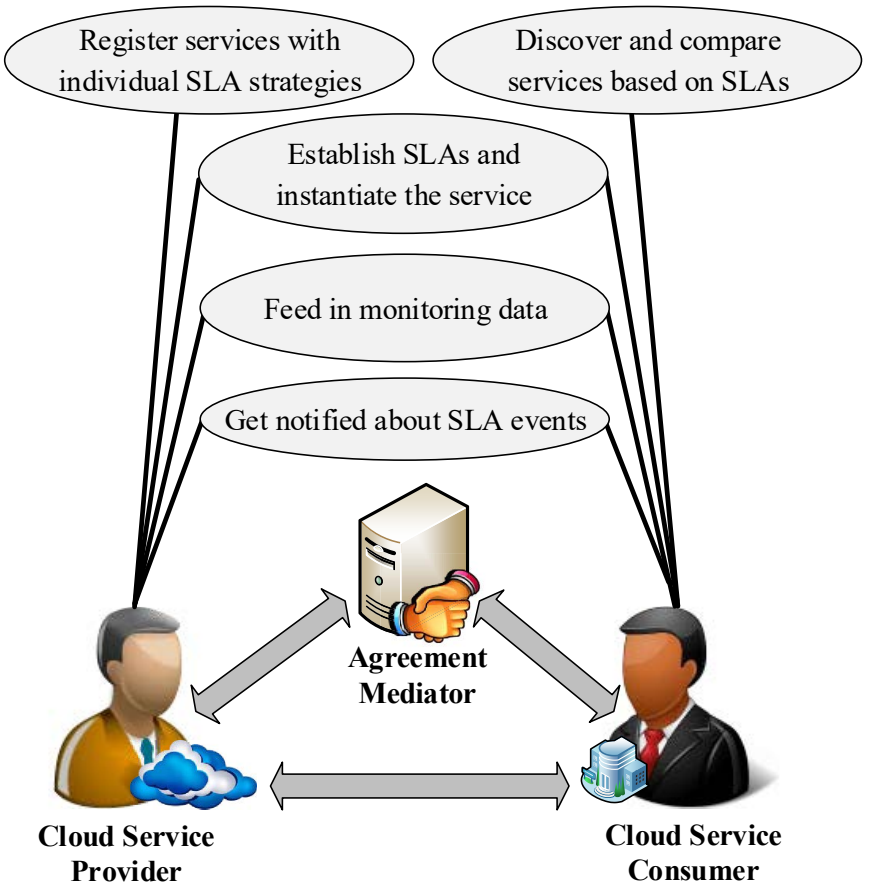

FIGURE 1.1.: Relationships and actions with the *Agreement-Mediator*

The *Agreement-Mediator* presented here has to respect already accepted and widely established standards for SLA and cloud management protocols. Further requirements include the applicability to broader service domains in order to use this approach also for every cloud-based service from the IaaS up to the SaaS layer. Similarly to traditional SLAs, agreements should describe the quality of a service, and what has to be monitored to verify agreement compliance. In contrast to plain SLAs this *Agreement-Mediator* approach should follow a normative format. This enables an automatic comparison of SLAs from different providers or the comparison of an expected SLA to given provider offers [12]. Thus it is possible to have a self-acting system to make decisions whether an SLA of a service is acceptable or not.

Additionally, the *Agreement-Mediator* presented in this thesis tackles the problems described before in several ways: Firstly, the *Agreement-Mediator* as a third party has the potential of representing multiple providers and consumers. If the *Agreement-Mediator* represents multiple providers at the same time, it advertises SLAs of different providers in one central place in a standardized and normative way and thus increases the consumer's ability of comparing offers of different providers. That enables consumers to make better decisions about which provider

they want to work with. Thus the *Agreement-Mediator* also has the potential for increasing the providers' competition among each other about who offers the best relation between cost and terms. Secondly, the SLA language and protocol used for the *Agreement-Mediator* allows the expression of terms, guarantees and compensations in a way that automating the process of (formally) verifying contractual compliance of both parties by using external monitoring data becomes almost trivial. Lastly, the normative way in which agreement offers are expressed in this approach allows for rapidly changing terms, advertisements, and pricing conditions for cloud services offered by providers around the world.

Parts of this thesis have been released in the following publications:

- **G. Katsaros - Contributors: T. Metsch, J. Kennedy, A. Stanik, W. Ziegler**
  Open Cloud Computing Interface - Service Level Agreements
  Open Cloud Computing Interface (OCCI) Extension Specification (v1.2), Open Grid Forum (OGF), document reference: GFD-P-R.184, 2015

- **A. Stanik, M. Körner, O. Kao**
  SLA Aggregation for QoS-aware Federated Cloud Networking
  IET Networks Journal, The Institution of Engineering and Technology, Online ISSN 2047-4962, vol. 4, iss. 5, pp. 264-269, 2015

- **A. Stanik, O. Kao, R. Martins, A. Cruz, D. Tektonidis**
  MO-BIZZ: Fostering Mobile Business through Enhanced Cloud Solutions
  In: Proceedings of the 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE Computer Society, pp. 915-922, 2014

- **A. Stanik, F. Sander, O. Kao**
  Autonomous Agreement-Mediation based on WS-Agreement for improving Cloud SLAs
  In: Proceedings of the 2014 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom), IEEE Computer Society, pp. 583-590, 2014

- **A. Stanik, M. Körner, L. Lymberopoulos**
  SLA-driven Federated Cloud Networking: Quality of Service for Cloud-based Software Defined Networks
  In: Proceedings of the 2014 9th International Conference on Future Networks and Communications (FNC), Elsevier, vol. 34, pp. 655-660, 2014

- **F. Feigenbutz, A. Stanik, A. Kliem**
  REST as an Alternative to WSRF: A Comparison Based on the WS-Agreement Standard
  In: Proceedings of the 2014 15th International Conference on Web Information Systems Engineering (WISE), Springer International Publishing, vol. 8787, pp. 294-303, 2014

- **A. Stanik, M. Cecowski, D. Tektonidis, F. Cleary, C. Bhatt, A. Balasas**
  MO-BIZZ: Ecosystem for Mobile Cloud Services
  Web Publication at: 2014 7th IEEE/ACM International Conference on Utility and Cloud Computing (UCC), IEEE Computer Society, 2014

- **M. Körner, A. Stanik, O. Kao**
  Applying QoS in Software Defined Networks by Using WS-Agreement
  In: Proceedings of the 2014 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom), IEEE Computer Society, pp. 893-898, 2014

1.3. Outline of the Thesis

The remainder of this thesis is structured as follows:

Chapter 2: SLA Fundamentals and Definitions

Chapter 2 gives background information and presents SLA basics and approaches for machine processable SLA languages and protocols. Furthermore, a detailed explanation of WS-Agreement and WS-Agreement Negotiation is presented that helps to understand the analysis and the design decisions of further chapters.

Chapter 3: Standards and Implementation Analysis

Chapter 3 analyses standards and specifications belonging to WS-Agreement and WS-Agreement Negotiation. Moreover, both of these SLA standards and their reference implementations are analyzed. Additionally, improvements to WS-Agreement and WS-Agreement Negotiation are presented which aim to apply these standards to state-of-the-art technologies.

Chapter 4: Agreement Mediation Approaches

Chapter 4 presents *Agreement-Mediator* approaches and architectures which aim to solve SLA issues and to improve service offerings of state-of-the-art cloud environments. In particular, the approaches presented in this chapter implement concepts for neutral and autonomous *Agreement-Mediators* which mostly make use of advanced specifications based on WS-Agreement. This chapter not only presents a set of approach, but furthermore analyses their advantages and disadvantages which influence the design and the concept for an all-encompassing *Agreement-Mediator* presented in chapter 5.

Chapter 5: Intercloud SLA Management

Chapter 5 is the main chapter of this thesis and describes the architecture and the most important concepts for the next generation of cloud computing environments: The *IEEE Intercloud Project*. Based on the latest achievements and the current state of the standardization progress in this project, the all-encompassing *Agreement-Mediator* approach is presented. This approach addresses issues regarding cloud service discovery and SLA-management in terms of: flexibility of term and price conditions, transparency and verification of advertised service quality, and flexibility of SLA expressions. This *Agreement-Mediator* concept creates a situation that potentially increases the competition among providers by introducing a global marketplace where customers are able to compare service offers regarding attractiveness, quality, and pricing conditions. Furthermore, this concept reduces the overall overhead of SLA-management and increases trust in the business relationship between customers and providers. Thus, the *Agreement-Mediator* improves interoperability and efficiency of state-of-the-art cloud SLAs.

Chapter 6: Intercloud Prototyping

Chapter 6 presents the experimental developments of the *Agreement-Mediator* concept and its evaluation. In particular, chapter 5 not only presents the concept, but also defines a set of specifications for protocols which are implemented and evaluated in terms of functionality and performance. The results that are presented in this chapter illustrate significant improvements and the advantages coming along with the overall outcome of this thesis.

Chapter 7: Conclusion and Future Work

Chapter 7 concludes the thesis with a summary and an outlook on future work.

2. SLA Fundamentals and Definitions

Contents

The goal of this chapter is to explain fundamentals which are required for the overall understanding of this work. Section 2.1 not only describes fundamentals, but also introduces definitions and formal descriptions. Section 2.2 presents approaches for Service Level Agreements, compares these approaches, and discusses their advantages and disadvantages. Based on the outcome of this discussion, the language and the protocol of WS-Agreement and its extension WS-Agreement Negotiation are described in sections 2.3 and 2.4.

2.1. Disambiguation

A Service Level Agreement (SLA) is equivalent to a contract between an IT service provider and a customer. It describes a common understanding about the IT service with its characteristics and its quality, the necessary properties to use the service, and the responsibilities of all involved parties. SLAs are supposed to express the Quality of Service (QoS) and its associated guarantees, like the minimum of performance and the average performance to expect. Ideally, the level of a service is a measurable and thus a comparable value.

Such an agreement can be negotiated between parties. If both the customer and the provider agree to the negotiated conditions, an SLA is created that serves as a formal contract. Here, the most important declarations which are required to serve as a foundation that can also be used in court are: Who is involved in this contract? Who is the provider? When was this agreement created? How was the agreement accepted by the involved parties? These declarations are specified in the *context* of an agreement.

Definition 2.1. *The context of an agreement denoted as* $context$ *consists of a* $provider$ *and a* $customer$ *name, a unique signature delivered by and identifying both parties denoted as* $signature_{provider}$ *and* $signature_{customer}$*, an agreement instantiation* $date$*, and a validity period denoted as* $period$*.*

$$context := \{provider, signature_{provider}, customer, signature_{customer}, date, period\} \tag{2.1}$$

All other parts of an agreement such as the service description, the service references, the quality of the service and the associated guarantees, penalties and compensations that are paid in case of violations, or other business related clauses are expressed as $Terms$. Therefore, an SLA is defined as follows:

Definition 2.2. *A Service Level Agreement denoted as* sla *consists of a* $context$ *and a* n*-tuple of terms denoted as set* $Term$*.*

$$sla := \{context, (Term_1, \ldots, Term_n)\} \tag{2.2}$$

SLAs are often concerned among others with the following contents: Service Definition, Service Performance, Problem Management, Customer Duties, Security, and so on. SLAs have been initially used by telecom operators who incorporated the agreements for a specific service level in the customer contracts. However, SLAs are used in various fields today. Except from telecom and Internet Service Providers (ISP), SLAs are also used for example in E-Commerce systems, hosting and housing of servers, cloud computing services and outsourcing in general.

SLAs are specified in the Information Technology Infrastructure Library (ITIL) as part of a larger concept called Service Level Management (SLM) [22]. This SLM is a process that consists of five stages: Negotiation, Finalization, Monitoring, Report, Revision. In this process an SLA is the foundation that is used during the whole service provisioning as depicted in figure 2.1. Service Level Requirements (SLR) are defined previously and are the starting point for SLM. Here, SLR are designed by one party and are used as basis for the Negotiation stage. As

a result an SLA is defined in the Finalization stage whereby a service is instantiated after both parties agreed on it. This service has to be observed in the Monitoring stage where measurements are delivered and are used for the evaluation of an agreement. Here, the measurements are required to show that the delivered service complies or does not comply with the SLA in the Reporting stage. The process terminates after the Revision stage which is followed by a Service Improvement Process (SIP).

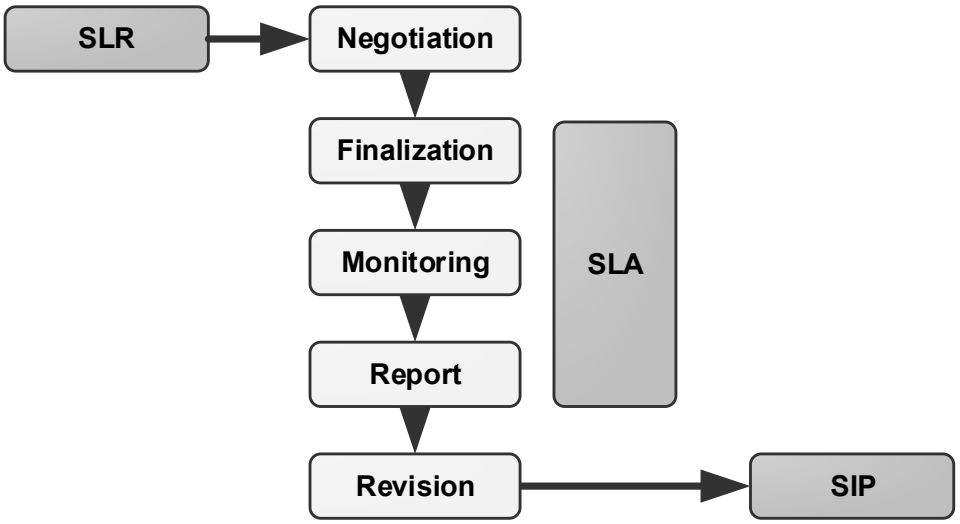

FIGURE 2.1.: Service Level Management process

The ITIL separates SLAs into several structure levels: Customer-based SLA, Service-based SLA and Multi-Level SLA. These SLA structure levels are defined as the following [22]:

Definition 2.3. *A **Customer-based SLA** is specific to a customer and can include several services with different service qualities.*

Definition 2.4. *A **Service-based SLA** is specific to a service with a specific service quality and has to be accepted by the customer as it is advertised.*

Definition 2.5. *A **Multi-Level SLA** is specific to a set of services that depend on each other and are used to specify agreements for various sets of users.*

2.2. Approaches for Service Level Agreements

In the beginning of service-orientated computing SLRs and SLAs were only defined in plain text. In general, the overall SLM process were performed by humans without any automation. But service providers quickly realized that they need more complex approaches to express

SLAs, in particular solutions and frameworks to offer, discover, negotiate, create, and monitor dedicated service level agreements.

SLAng is used to define service level agreements for the provisioning of distributed applications [23] [24]. Such applications are a combination of components, where end-to-end QoS constraints for each application are satisfied by certain QoS constraints. This approach is an XML-based language and describes a service provisioning reference model, consisting of an application tier, a middle tier and a resource tier. These tiers are build on each other. Starting at the bottom, the resource tier comprises the underlying resources, for example storage and network resources. The middle tier comprises middleware components like service container or web service applications and the application tier comprises the provided application. SLAng distinguishes between vertical and horizontal SLAs, where vertical SLAs describe contracts between components of different tiers and horizontal SLAs describe contracts between components of the same layer.

Web Service Level Agreement (WSLA) is an SLA management framework specification created by IBM [25] [26]. This specification not only provides the required capabilities for creating and monitoring SLAs, but is also applicable for any other inter-domain management scenario. In particular, the WSLA specification defines a flexible and extensible language to enable service customers and providers to define SLAs and to specify SLA parameters. The WSLA specification has subsequently influenced the work on the WS-Agreement specification.

Web Service Offering Language (WSOL) is also an XML-based language definition that enables to offer web-services at different service levels [27]. Such a service offer represents a single class of services with a specific QoS. Furthermore, each service offer can also comprise functional and non-functional constraints. The idea of this approach is that the same service can be offered with different service levels. In particular, applications are able to search a desired service in a registry and are able to find multiple service instances, where each service instance is offered with a different service quality. Therefore, each service instance is offered with an associated service level.

WS-Agreement [28] and WS-Agreement Negotiation [29] are standards of the Grid Resource Allocation and Agreement Protocol Working Group (GRAAP-WG) of the Open Grid Forum (OGF). While WS-Agreement is the foundation specification, the WS-Agreement Negotiation specification is an extension of it. Both together define a protocol and a language to dynamically negotiate, renegotiate, create and monitor bi-lateral service level agreements in distributed systems. WS-Agreement and WS-Agreement Negotiation are the only specifications for SLAs which are standardized and accepted by a broad community. Therefore, the following subsections describe both in detail.

2.3. WS-Agreement

WS-Agreement is a language and a web service protocol that allows for establishing agreements between service provider and service consumer. It is designed according to the WS-* Specification family. Similar to traditional SLAs WS-Agreement can be used to describe a service with appropriated service qualities, guarantees, and business related definitions. Such an agreement can also be used to monitor the compliance of the agreement, because service levels are expressed in these agreements in a machine-readable form.

By using the Extensible Markup Language (XML) [30] for describing agreements and templates of agreements in WS-Agreement, this protocol enables to discover possible service offerings of independent providers. The specification of WS-Agreement is published by the OGF and relies on a set of established standards like XML, *Simple Object Access Protocol (SOAP)* [31], *Web Service Description Language (WSDL)* [32], and *Web Services Resource Framework (WSRF)* [33]. It consists of three main parts: Two schemas, one for specifying a WS-Agreement and another for the specification of WS-Agreement templates, additionally it contains operations and port-types for the lifecycle management. Here, a lifecycle of such an agreement includes creation, monitoring and termination. WS-Agreements in contrast to traditional SLAs follow a normative format. This enables the automation of processes for establishing agreements between provider and potential customer.

2.3.1. WS-Agreement Model

WS-Agreement allows service consumers to dynamically explore possible services with their appropriated service levels. The specification distinguishes between the following two layers which are also depicted in figure 2.2:

- The **service layer** focuses on the functionality provided by the service.
- The **agreement layer** focuses on the quality of the provided service.

In particular, an agreement has to be established on the agreement layer before the service is provided on the service layer. The monitoring of a delivered service is performed on the service layer. The information measured during the monitoring is than passed to the agreement layer. Thus, this information can be used for evaluating the compliance of an agreement on the agreement layer.

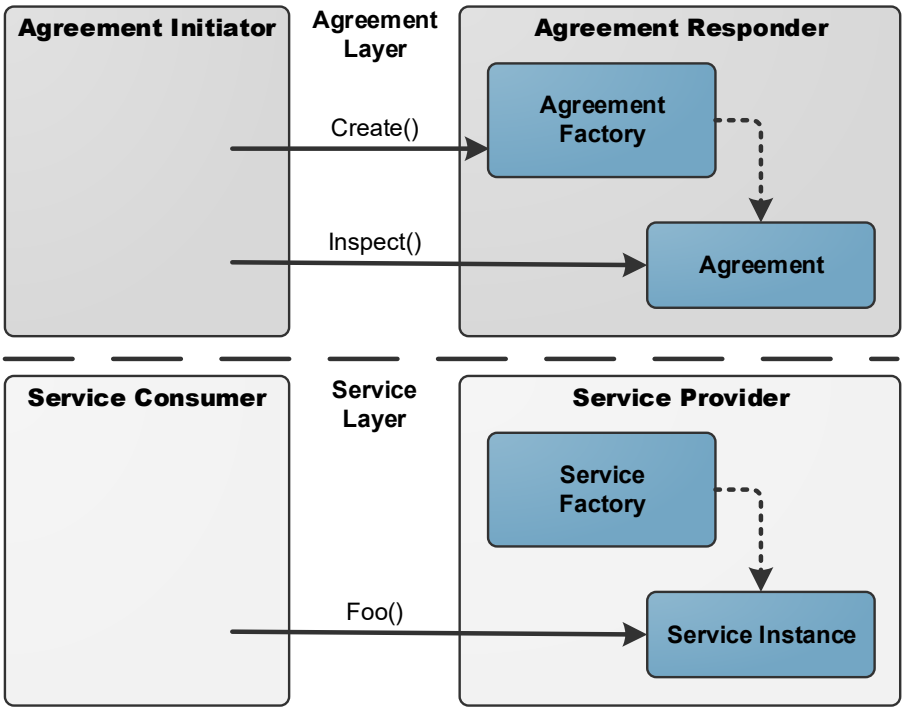

FIGURE 2.2.: Layered service model of WS-Agreement

2.3.2. WS-Agreement Protocol

The protocol of the WS-Agreement specification defines the required services and operations to create and monitor service level agreements. Therefore, two types of services are defined:

- The **agreement factory services** are used for creating agreements and instantiating the associated services according to the agreed QoS.

- The **agreement services** are used for observing the compliance and the state of agreements.

The specification defines a set of interfaces in the form of WSDL, thus enabling the interaction with the two core services that are already depicted in figure 2.2. The services are designed as web service resources according to the WSRF specification. These web service resources can be addressed via an endpoint reference (EPR), which is specified in the WS-Addressing specification [34].

2.3.3. WS-Agreement Factory Service

An agreement is a contract between a service provider and a customer. On the agreement layer, if the customer initiates the creation of an agreement, the customer is called agreement initiator and the service provider that creates an agreement instance is called agreement responder. However, these roles with appropriate responsibilities of each party can vary depending on the scenario.

The agreement factory service is located at the responder side and is used by the agreement initiator for SLA establishment. In order to create an agreement the initiator defines its requirements in an agreement offer. This offer comprises all necessary quality constrains for the service to provide and also additional guarantees that are associated with the service. Since the content of an agreement depends on the service to provide and can be very complex, a template mechanism guides the initiator in creating agreement offers. Therefore, the responder publishes a set of agreement templates that the initiators can use to create new offers. The template itself is basically a prototype of an agreement offer and specifies the structure and the content of the offer. In order to create a new agreement, the agreement factory service queries the available templates to the agreement initiator, as depicted in figure 2.3.

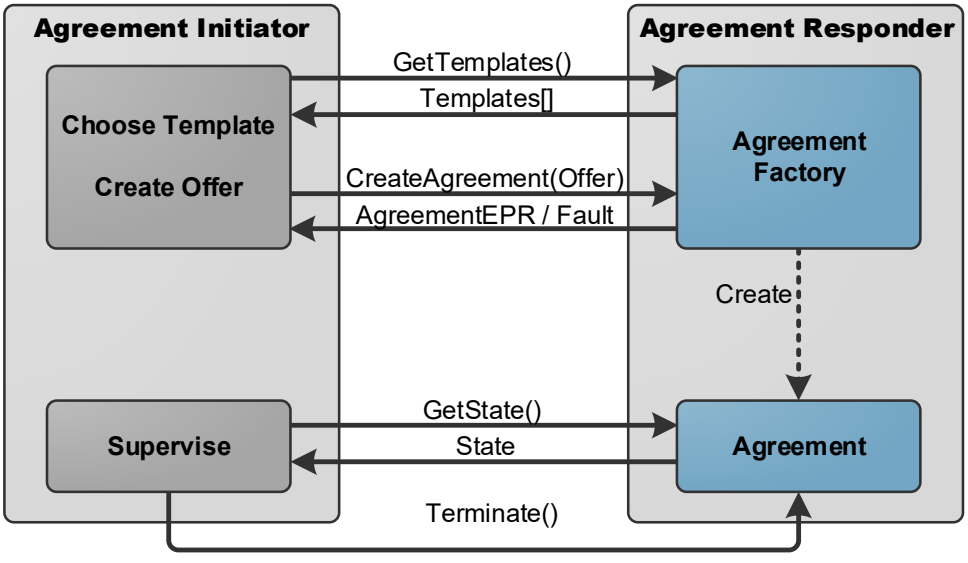

FIGURE 2.3.: Agreement creation process and lifecycle

The customer creates an agreement offer based on an appropriate template that fits the requirements and sends it to the service provider. After the transmission of an offer, the customer is bound to it. The WS-Agreement specification defines two methods for the agreement factory service for creating agreements:

- The synchronous *createAgreement* operation is specified by the *AgreementFactory* port type and requires an immediate decision to accept or reject an incoming agreement offer. In the case of acceptance, an endpoint reference is returned to the new agreement instance.

- The asynchronous *createPendingAgreement* operation is specified by the *PendingAgreementFactory* port type and doesn't need a decision immediately. Although a new agreement instance is created and an endpoint reference is returned, the agreement itself is in a pending state. Once the responder has made a decision, the agreement instance changes its state to either observed in case of acceptance or rejected. In both cases the initiator can be optionally notified about the decision.

2.3.4. WS-Agreement Agreement Service

Once the responder has received an agreement offer, the agreement factory instantiates an agreement instance in an appropriate state. Each agreement instance has a service interface for accessing the content of an agreement, for monitoring the agreement at runtime and for managing its lifecycle. The agreement service is designed as WSRF resource and the properties of an agreement service instance are WSRF resource properties. Two port types are specified for an agreement service:

- The **Agreement port type** defines a set of agreement resource properties (agreement context, terms, and id) and a method to terminate an agreement.

- The **AgreementState port type** defines a set of agreement resource properties (state of the agreement, state of the distinct service terms, and state of the guarantee terms) in order to observe the agreement compliance.

In order to couple the monitoring of a service with the evaluation of an SLA, the service term states provide the mechanisms and the relevant information from the service monitoring (e.g. average response time of a service). Additionally the service term states can include corresponding service deployment information which can also be exposed by the service provider on this machine-readable way.

2.3.5. WS-Agreement Language

In contrast to the WS-Agreement protocol that defines the services and methods, the WS-Agreement language defines the data types and the structure of the agreement document, the agreement template document, and the agreement offer document. This definition is separated

in two XML schemas, the agreement schema and the agreement state schema. While the agreement schema defines the core data types, the agreement state schema defines the data types for the agreement observation such as the agreement states, service term states, and guarantee term states. An agreement consists of an agreement identifier, an agreement name, an agreement context, and a term compositor with a detailed description of the service as depicted in the figure 2.4.

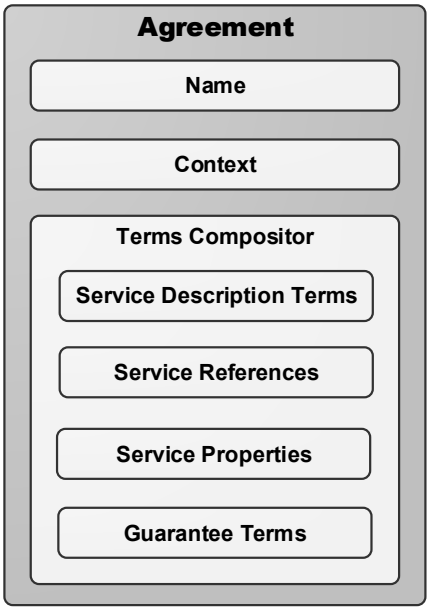

FIGURE 2.4.: Structure of an agreement

Agreement Context

The agreement context contains information about the parties that are involved. These are the agreement initiator and the agreement responder. This information can be expressed either in a domain specific description of each party or as an endpoint reference or a distinguished name that identifies the party. Furthermore, each role of the participants has to be specified in the agreement context. In fact, who is the service provider and who is the service consumer. Additionally, the agreement context may also contain an expiration time, a reference to the template that was used to create the agreement, and further domain specific data.

Agreement Terms

The agreement terms describe the service to be provided and its associated guarantees. Therefore terms are split into two groups: *Service Terms* and *Guarantee Terms*. While service terms

describe the different aspects of a service, guarantee terms specify the guarantees that are associated with the services with optional compensation methods which came into effect in case of guarantee fulfillment or violation. Besides an identifier, service terms also have a service name that can be used to semantically group multiple service terms. In particular, a single service can be described by multiple service terms where each service term describes a specific aspect of the service to provide. Service terms are subdivided in *Service Description Terms*, *Service References*, and *Service Properties* as depicted in figure 2.4.

Service Description Terms

Service Description Terms are functional and domain specific descriptions of a service to provide. Therefore, the service description can be any valid XML document that describes a service completely or partially. Because of the domain independency of WS-Agreement, the content of a service description term could only be understood by the involved participants (agreement initiator and agreement responder).

Service References

Service References can be used to refer to existing services with certain service quality constrains within an agreement. This service reference can be expressed like a service description term as an arbitrary XML document.

Service Properties

Service Properties are used to define variables in the context of an agreement, which are later used to evaluate the guarantees of this agreement. In order to define a variable a name, a metric, and a location is necessary. The location refers to a distinct element by using an XML query language like XPath.

Guarantee Terms

Guarantee Terms are used to express service guarantees and consist of four elements: a *Service Scope*, a *Qualifying Condition*, a *Service Level Objective*, and a *Business Value List*. Since a single agreement can comprise a set of different services, the *Service Scope* specifies the service that is covered by the guarantee. Because not all guarantees apply during the whole lifetime of an agreement, the *Qualifying Condition* specifies preconditions that must be fulfilled before a guarantee can be evaluated. The *Service Level Objective* defines an objective that must be met in order to provide a service with a particular service level. This *Service Level Objective*

basically defines how the agreed service levels are related to the actual QoS properties. The *Business Value List* defines the penalties and rewards that are associated with a guarantee.

Term Composition

The *Term Composition* of the WS-Agreement language is a simple grammar to structure terms in an agreement. This structure is called term tree and represents terms in a tree-like data structure, where each node is a term. Additionally, the term tree implements the composite pattern that enables the representation of part-whole hierarchies. The WS-Agreement language defines three types of term compositors: the *All* compositor, the *OneOrMore* compositor, and the *ExactlyOne* compositor. These term compositors are equivalent to the logical AND, OR, and XOR functions.

2.4. WS-Agreement Negotiation

The WS-Agreement protocol enables to compare agreement offers, to create agreements, and to evaluate agreements automatically [35]. Thus, it is possible to have a self acting system to make a decision whether an SLA offer is acceptable or not. However, the system is only able to either agree to all parts of an SLA offer or to refuse it, according the "take-it-or-leave-it" concept. The negotiation of SLAs is still up to the end user.

In order to change this situation, GRAAP-WG of the OGF published the WS-Agreement Negotiation specification which is an extension of the WS-Agreement specification. For the negation of SLAs, the service provider offers its services with advertised service levels via WS-Agreement templates. A template contains the general description of the service, the guarantees that can be given, and some options for the customer to choose from.

With WS-Agreement Negotiation the consumer is now able to declare all desired requirements is a negotiation template before it is send to the provider. The provider checks the requirements and whether he is able to fulfil or to accept them. If so, he can send back a counter offer with corresponding price information. If the requirements cannot be fulfilled, the provider can also send a counter offer with likewise service levels which he is able to fulfil (e.g. less computing nodes for a longer time frame instead of more computing nodes in a shorter time frame). This may take several rounds until both parties have come to a final agreement. The same process is used if an SLA has to be re-negotiated [35].

2.4.1. WS-Agreement Negotiation Model

The WS-Agreement Negotiation model consists of two parts: the negotiation offer/counter offer model and the layered architecture model. While the negotiation offer/counter offer model describes the negotiation process to create a new agreement, the layered architecture model describes the WS-Agreement Negotiation layer on top of the WS-Agreement layer and the service layer.

The WS-Agreement Negotiation offer/counter offer model describes the dynamic exchange of information between the negotiation initiator and the responder. In order to create an agreement both sides can negotiate an acceptable agreement in multiple rounds. The negotiation process starts with an initial offer that is based on a negotiation template. This initial offer is send to the other party, which in turn creates a counter offer for the negotiation offer received. Each counter offer is always based on a negotiation offer that was previously received and has an associated state that reflects the acceptability of the party.

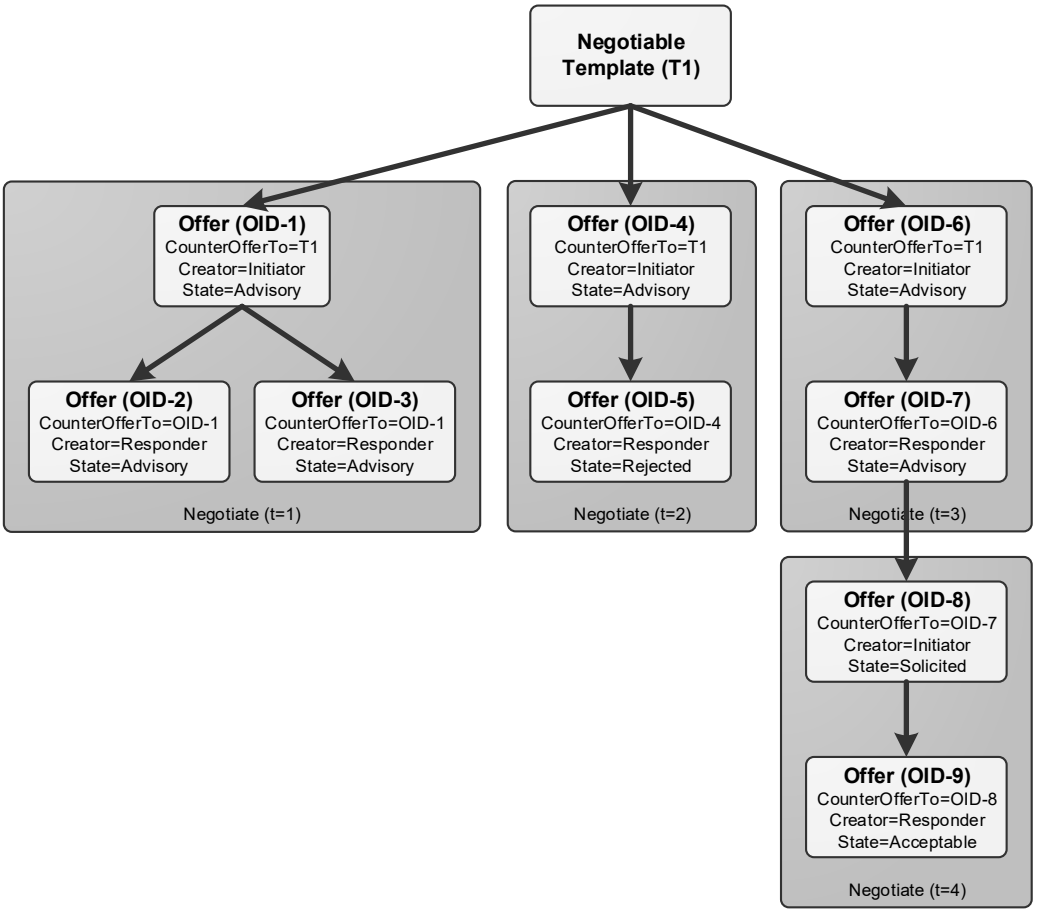

FIGURE 2.5.: Example negotiation tree of multiple negotiation rounds

In a multiple round negotiation, each exchanged negotiation offer can be represented in a rooted tree with a negotiable template as root node. Each child of the root node is an initial offer and each negotiation offer in this negotiation tree is a counter offer to its parent node. Leaf nodes are negotiation offers that either are in the acceptable state and can be used to create an agreement or are in the terminal rejected state.

In the example negotiation tree depicted in figure 2.5, the negotiation initiator creates an initial negotiation offer with an offer id 1 (OID-1) that is based on a negotiable template from the responder. This offer is then send to the negotiation responder, where the responder examines the incoming offer (OID-1) and creates two counter offers with OID-2 and OID-3. If the negotiation initiator processes returned any counter offer because it decides that both counter offers do not fulfil its requirements, than the initiator starts a new negotiation branch by creating a new initial negotiation offer (OID-4) based on template T1. If this offer is in turn unacceptable for the responder, a counter offer (OID-5) is created in the rejected state. Finally, the negotiation initiator creates a third negotiation branch, where after several rounds the negotiation responder returns a counter offer (OID-9) that fulfils the requirements of both sides and is in the acceptable state. Based on this offer the negotiation initiator creates the new agreement.

The WS-Agreement Negotiation architecture model extends the WS-Agreement model with an additional negotiation layer. The three layers are depicted in figure 2.6 and have a clear separation to decouple them from each other:

- The **negotiation layer** provides a protocol and a language to negotiate agreement offers and counter offers based on negotiated templates. This offers and counter offers don't imply a promise of the agreement responder. In fact, they are rather an instrument that is used to find an acceptable agreement to subsequently create an agreement.
- The **agreement layer** is part of the WS-Agreement specification and was already described in detail in the sections before. If an acceptable agreement has been negotiated in multiple rounds, an agreement can be created based on the negotiated offers by calling either the *createAgreement* or the *createPendingAgreement* operation on the agreement responder's *Agreement Factory* port type. Furthermore, the functionality for agreement evaluations, agreement observations, and all other agreement management tasks in general are still provided on this agreement layer.
- The **service layer** provides the actual service (e.g. a web service, resource provisioning service, etc.) and is governed by the agreement layer.

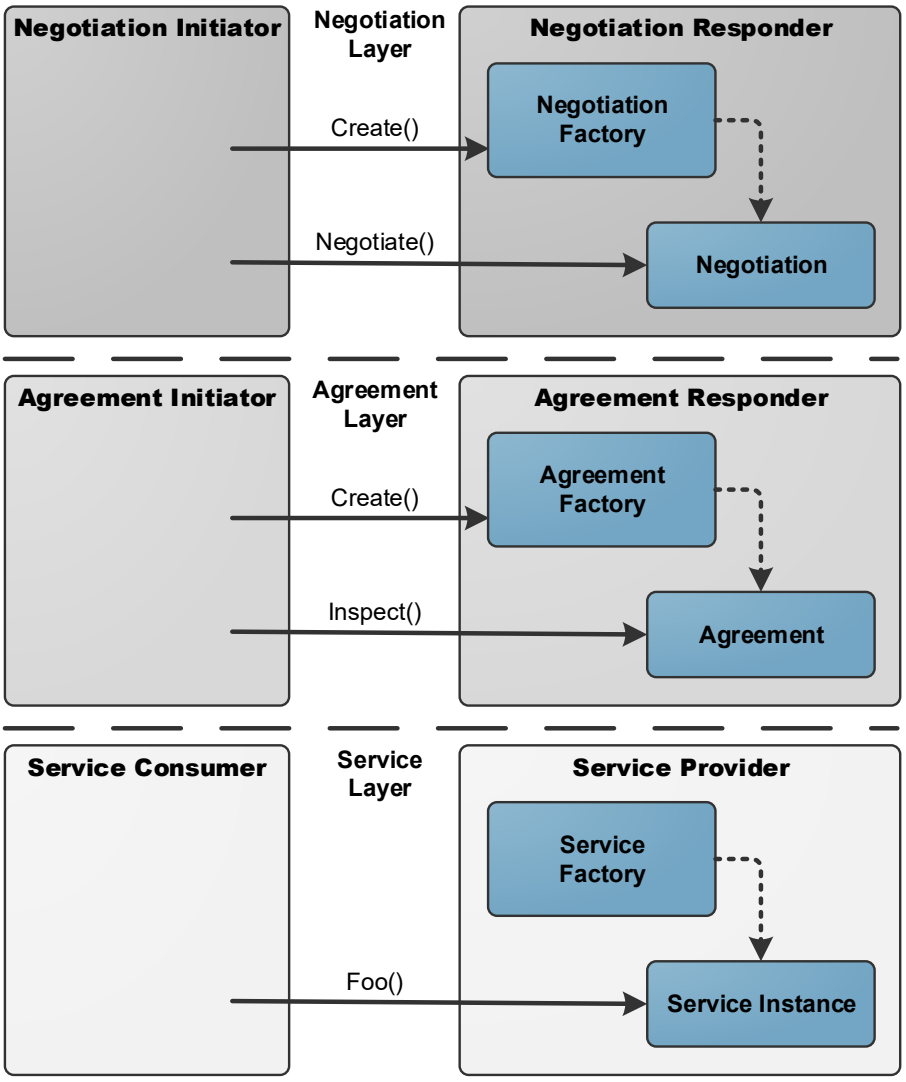

FIGURE 2.6.: Layered service model of WS-Agreement Negotiation

2.4.2. WS-Agreement Negotiation Protocol

The WS-Agreement Negotiation protocol is used to dynamically exchange information in order to reach a common understanding of a valid agreement offer. The service that is required to establish a negotiation between two parties is enabled by a negotiation service instance. This instance is limited in its lifetime or by the maximum negotiation rounds. These limitations are defined in the negotiation context. Furthermore, the negotiation context defines besides the expiration time also the roles of the negotiation participators and their obligations.

Moreover, also the nature of the negotiation process is defined in negotiation context. This specific reference of the negotiation instance can be either a negotiation of a new agreement or

a renegotiation of an existing agreement. In case of a renegotiation of an existing agreement, the negotiation type must include an endpoint reference to the responder's agreement that is the subject of this renegotiation. Additionally, the element in the negotiation type can also contain domain specific data that can be used to control the negotiation process in a domain-specific way.

In general, the WS-Agreement Negotiation specification is an extension of the WS-Agreement specification and thus extends all core agreement types. This makes it very easy to convert a negotiation offer into an agreement. In order to create an agreement based on an agreement offer, the agreement factory on the agreement layer is used. The negotiation layer on top and the negotiation process itself do not bear any obligations for any negotiating party.

2.4.3. WS-Agreement Negotiation Language

As described before, a negotiation process is initiated by an initial offer based on a negotiation template. In order to negotiate the service level, the initiator and the responder exchange offer and counter offer while each counter offer based on a previous offer. The structure of such an offer is very similar to the structure of an agreement of the WS-Agreement specification. Additionally to the agreement elements, a negotiation offer contains also a *Negotiation Offer Id*, a *Negotiation Offer Context*, and some *Negotiation Constraints*. It extends the agreement schema of the WS-Agreement specification and therefore inherits the agreement name, agreement context, and the agreement terms. The basic structure of a negotiation offer is depicted in figure 2.7.

Negotiation Offer ID

The *Negotiation Offer ID* is the identifier of a specific offer and is unique for both parties. In turn, the name is an optional element that is the name of the agreement and can also be later set during the agreement creation as described in the WS-Agreement specification. While the *Negotiation Offer ID* and the name are additional elements in reference to the basic structure of an agreement, the term element is inherited from the WS-Agreement core data types. In the context of a negotiation process, the terms are the content of the negotiation process and specify both, the structure and the values of the agreement terms.

Negotiation Constraints

The *Negotiation Constraints* are used to express the requirements of a negotiation participant and thus to control the negotiation process. Therefore, the constraints define restrictions for

FIGURE 2.7.: Structure of a negotiation offer

structure and values of negotiation counter offers. These *Negotiation Constraints* are not constant and can change during the negotiation process. In particular, if the negotiation initiator chooses one specific service term from a set of predefined terms, the responder can adopt them to his preferences by changing the negotiation constraints in a counter offer.

Negotiation Offer Context

The *Negotiation Offer Context* represents metadata associated with a specific negotiation offer, e.g. the id of the originating negotiation offer, its expiration time, and its state. Additionally, it may contain domain specific elements in order to provide negotiation extensions, for example to realize offers of a binding negotiation or additional compensation methods.

3. Standards and Implementation Analysis

Contents

The goal of this chapter is to analyse the WS-Agreement and the WS-Agreement Negotiation standards, their dependencies on other standards, and their applicability to an external *Agreement-Mediator* as third party. Furthermore, this chapter focuses not only on these standards, but also on their implementation. Therefore, this chapter is structured as follows: In section 3.1 the dependency on other standards is analysed and drawbacks are presented. Section 3.2 analyzes the fundamentals and the applicability of these standards to an autonomous *Agreement-Mediator* that allows for outsourcing SLA management tasks into a neutral zone.

The reference implementation *WSAG4J* is presented in section 3.3 and its applicability to an external *Agreement-Mediator* approach is discussed. In section 3.4 an alternative to WSRF is discussed and a theoretical comparison of both approaches focusing on motivation, features as well as interoperability and network load is presented. Section 3.5 describes a practical approach for this comparison, where performance benchmarks have been performed and show the differences in terms of scalability, availability, and efficiency. Finally, section 3.6 presents further REST-based standards which are already applied in today's cloud middlewares.

3.1. WS-Agreement Dependency Analysis

The WS-Agreement specification describes the essential functionality for advertising, establishing, and monitoring SLAs for distributed and service-oriented environments. This specification is based on WSRF and thus also the WS-Agreement Negotiation specification have to be based on this fundamental standard. WSRF is a set of specifications which define operations for web services in order to become stateful. Although the WSRF specifies stateful operations among other things, WS-Agreement makes only use of the *GetResourceProperty* operation for reading data. The reason for this is obviously the conceptual separation of resources on the one hand and their properties on the other. Different operations are defined for both entities. While resource properties have a wide range of operations such as read, add, replace, delete, or search; a resource itself can only be deleted [36]. The creation of resource services is not standardized. Therefore, WS-Agreement defines proprietary operations such as *CreateAgreement* or *CreatePendingAgreement* which produce a new resource as a result. Why WS-Agreement also defines a specific *Terminate* operation for the deletion of Agreements (resource services) and do not make use of the WSRF operation *Destroy* is highly questionable. Furthermore, the standardized WSRF *Destroy* operation is explicitly mentioned in the WS-Agreement specification as an alternative variant for this functionality [28]. Since the *Terminate* operation of WS-Agreement can have additional parameters, one reason could be the transmission of a justification for canceling an agreement. Another reason for this decision may be the lack of extension points in the WSRF specification, because the specified *Destroy* operation by WSRF, like any other WSRF operation, permits no extension of arguments or additional content.

For the design of WS-Agreement only a standardized way can be used to implement the required operations. In particular, only other standards can be used to specify another standard that has to build on already accepted standards. Since conformity and interoperability are the foundation for wider accepted standards, the use of WSRF in this approach is in general to be regarded as meaningful. However, the use of WSRF for this purpose is simply oversized, because WS-Agreement only makes use of a single operation from five specifications of the heavy WSRF specification family. By the current state of scientific knowledge and technology maturity the use of WSRF can be considered as impractical, because the only noteworthy Java

implementations of WSRF are Apache Muse and the Globus Toolkit. Besides the low distribution, the development and support for Apache Muse have been stopped in 2013 [37] and for WSRF in Globus Toolkit already in 2010 [38].

3.2. WS-Agreement Applicability Analysis

The overall goal of this thesis is the design of an autonomous web service based *Agreement-Mediator* as a third party in the cloud market that provides a way to source out affairs regarding the management of SLAs into a neutral zone without taking the customer's or the provider's side. Thus, the *Agreement-Mediator* adds an additional transitive relationship between a provider and a customer without breaking or directly affecting any existing ones. Here, the applicability of the two existing standardized SLA protocols (WS-Agreement [28] and WS-Agreement Negotiation [29]) is analysed. Furthermore, this analysis also proposes changes to these specifications in order to reach the requirements for the applicability to an autonomous *Agreement-Mediator*.

3.2.1. Participant roles

WS-Agreement Negotiation defines a layered architectural model as depicted in figure 2.6 consisting of three layers: the negotiation layer, the agreement layer, and the service layer. These layers are clearly separated and specify two roles per layer: the negotiation layer is composed of the negotiation initiator (NI) and the negotiation responder (NR), the agreement layer is composed of the agreement initiator (AI) and the agreement responder (AR), and the service layer is composed of the service consumer (SC) and the service provider (SP). The roles of these parties (who is the provider and who is the consumer) are not specified and have to be declared during the interaction of these parties. In particular, this separation in roles allows that both SC and SP could be the AI or the AR as well as the NI or the NR.

The WS-Agreement standard specifies that the *AgreementFactory* and the *PendingAgreementFactory* port types are always provided by the AR. This means that the *Agreement-Mediator* is in any case the AR. In the *AgreementContext* of an agreement template, two optional elements can be used to identify the AI and the AR. These elements may be an Uniform Resource Identifier (URI) [39], an EPR [34], or may identify the parties by a more abstract type of naming [28]. Additionally, a required element called */wsag:Context/wsag:ServiceProvider* identifies the service provider and can either take the value *AgreementInitiator* or *AgreementResponder*. This point in the specification is an obstacle for the applicability of WS-Agreement to a third party (*Agreement-Mediator*), because neither the AI nor the AR is the SP in this case. Therefore, a proposal to solve this limitation could be to keep this required element as part of the specification, but to change the type of this element to *xs:anyType* as also specified for

/wsag:Context/wsag:AgreementInitiator and */wsag:Context/wsag:AgreementResponder*. Thus, a simple comparison of the */wsag:Context/wsag:ServiceProvider* element to the */wsag:Context/wsag:AgreementInitiator* and the */wsag:Context/wsag:AgreementResponder* would deliver the same result. Furthermore, one of the AI or the AR elements would get dispensable in the traditional applicability of WS-Agreement, because if the */wsag:Context/wsag:AgreementResponder* element is not used, the */wsag:Context/wsag:ServiceProvider* is obviously the AR.

On the negotiation layer the WS-Agreement Negotiation standard specifies a similar context type called *NegotiationContext*. Similar to the WS-Agreement specification, the WS-Agreement Negotiation also specifies two optional elements, which identify the NI and the NR, where the NR implements the *NegotiationFactory* port type. Here, a required element called */wsag-neg:NegotiationContext/wsag-neg:AgreementResponder* identifies the party in the negotiation process that acts on behalf of the agreement responder and can either take the value *NegotiationInitiator* or *NegotiationResponder* [29]. In terms of the *Agreement-Mediator* approach, the *Agreement-Mediator* is the NR, because it is the party that provides the *AgreementFactory* and the *PendingAgreementFactory*, which have to be also referred to in the negotiation context within the *AgreementFactoryEPR* element. A negotiation offer extends the *wsag:AgreementType* and therefore inherits the *AgreementName*, *AgreementContext*, and the agreement terms. Thus, the proposed change of the type to *xs:anyType* of the */wsag:Context/wsag:ServiceProvider* element would also influence the negotiation layer and does not require additional changes to the WS-Agreement Negotiation standard.

3.2.2. Protocol Variants

The overall process for establishing agreements can be divided into four steps (1 - 4): The AR advertises services through templates in which terms are proposed. The AR is able to accept these terms along with a set of creation constraints that mark formal boundaries for adjusting terms (step 1). The AI retrieves the advertised templates from the AR, chooses the one that fits the AI's requirements best and derives from the template an agreement offer (step 2). The AI then adjusts the terms of the agreement offer to its needs (according to the creation constraints of the template) and sends the offer back to the AR (step 3). The AR then checks its ability of fulfilling the AI's requirements and responses its decision in form of a newly created agreement in case of acceptance or a fault message otherwise (step 4). This scenario is illustrated in figure 2.3, where the states of the agreement, the services, and the guarantees are available for retrieval along with an endpoint reference to terminate the agreement.

The described scenario is only one of six protocol variants which are depicted in figure 3.1 and is called synchronous asymmetric (variant 1), because the AR decides to accept or reject the agreement offer immediately after retrieving it and also immediately notifies the AI about his decision within his answer to the AI's request. In the asynchronous protocol variant this

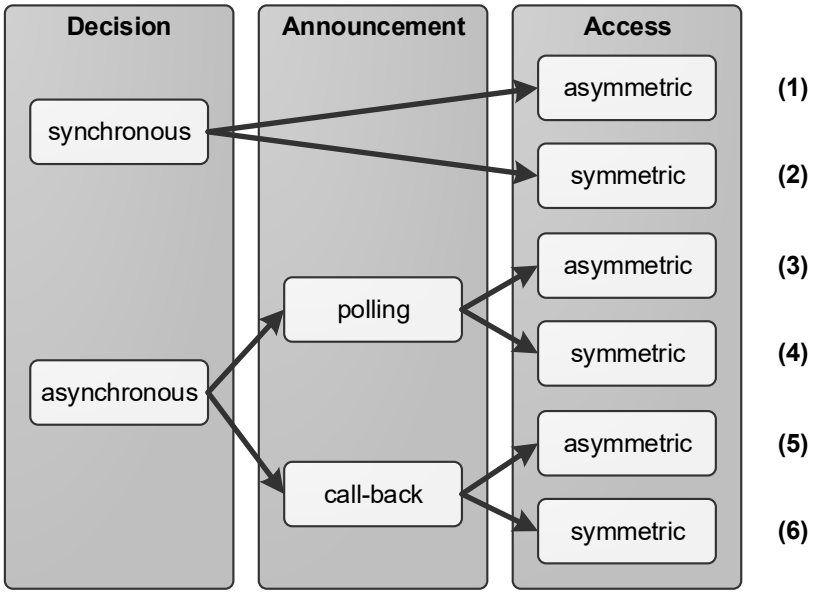

FIGURE 3.1.: WS-Agreement protocol variants

decision can be deferred, resulting in an agreement that resists in a "pending" state, which may change to a state representing acceptance or rejection over time. WS-Agreement specifies two procedures for the AI to get informed about the AR's decision: polling and call-back. Polling describes the behavior of the AI to constantly check the agreement states for changes. In order to follow the call-back variant an AI has to send an EPR to an implementation of the acceptance port type additionally to his agreement offer in step 3. This port type includes two methods representing acceptance and rejection and is called respectively by the AR when his decision is made. Furthermore, these three described types are all asymmetric because only the AR holds an instance of the actual agreement. If the AI does the same to represent his own view of the states, both behave in a symmetric way. That results in six variants in total: synchronous asymmetric (variant 1), synchronous symmetric (variant 2), asynchronous polling asymmetric (variant 3), asynchronous polling symmetric (variant 4), asynchronous call-back asymmetric (variant 5), and asynchronous call-back symmetric (variant 6).

Because of the complexity of the WS-Agreement protocol the only suitable protocol variants for a third party *Agreement-Mediator* are synchronous asymmetric, asynchronous call-back asymmetric, and asynchronous polling asymmetric, because only the asymmetric variants do not require the AI to implement any kind of server functionality that replicates agreement instances on the service consumer side. So, a consumer initiates agreements by calling either the *createAgreement* or *createPendingAgreement* endpoint of a service provider's factory. Which of the two endpoints should be exposed and what circumstances lead to acceptance or rejection of an agreement-offer is entirely defined by the provider.

3.2.3. Service Terms

The terms are the main subjects of an agreement. They are a set of service descriptions, references to service descriptions, properties of services that may be used to describe the quality of a service, and guarantees coupled with Service Level Objectives (SLO), which are bounded to fees and compensations. All these terms may be composed and combined to valid selectable service entities. In case of a provider offer only certain combinations of services and terms are possible. For instance, a guarantee may be expressed via the *GuaranteeTerm* element that references to a *ServiceDescriptionTerm* and contains (among others) a *ServiceLevelObjective* and a *BusinessValueList* element. The *ServiceLevelObjective* elements advertise the QoS a provider is willing to commit for a service. These are formulated by any kind of expression, which may be evaluated true or false. These expressions utilize variables that are defined by the *ServiceProperties* element. The *BusinessValueList* element contains any number of compensations in form of penalties or rewards which can contain a unit in which the compensation is measured like a currency and its corresponding amount. This normative format allows not only programmatic inspection and comparison of terms but also programmatic compliance verification.

Other data structures like *Templates*, *AgreementOffer* or types used for negotiation inherit from this agreement type or include one. Only the semantics of the terms differ in distinct types: In a *Template* the terms represent an AR's advertisement or a recommendation for an AI; in an *AgreementOffer* the terms represent the demand of an AI; and in an agreement the terms represent what AR and AI have agreed upon.

Considering that both standards are aiming to be universal standards in a multifarious environment like distributed service oriented architectures, many parts of the specification are defined as "domain-specific" and are explicitly left open. One of these parts is, for instance, the declaration of a term language to describe services. Another one is the connection between the agreement layer, which is described by this section, and the service layer, in which the actual usage of the service takes place. Because of that, a situation might occur where the actual instantiation of a service is triggered as a result of an established agreement. Doing so, the agreements managed by WS-Agreement would become more than SLAs and could be considered as instruments for managing services in general.

3.3. WSAG4J Framework Analysis

The WS-Agreement for Java (WSAG4J) framework [40] is the most complete and advanced solution according an evaluation by the GRAAP-WG from 2010 on existing WS-Agreement and WS-Agreement Negotiation implementations [41]. WSAG4J is written in Java, provides features for validating offers against creation constraints defined by templates, has a built-in

persistence layer for agreements and allows template creation with pure XML without requiring to write Java code. WSAG4J also made several concretizations that were necessary for a ready to use implementation like the usage of the *Java Expression Language (JEXL)* as expression language. Its ability is a very important feature to automatically verify the guarantee terms of an agreement: WSAG4J evaluates expressions, i.e. a qualifying condition inside the *ServiceLevelObjective* element, and consequently triggers events for compensations, that take place in a customizable accounting system.

WSAG4J also supports multiple agreement factories per host. A WSAG4J factory consists of an arbitrary number of WSAG4J actions that in turn consist of three strategies providing control logic for delivering a template, negotiating an offer and creating an agreement based on an offer. These strategies define domain-specific behavior, i.e. the actual dependency of the advertised templates on resource availability or how the decision regarding the acceptance of an agreement offer is made. Since WSAG4J actions are not defined by the standards they appear transparent to an agreement initiator. In particular, requests for templates, negotiation or the creation of agreement are received and answered by a WSAG4J factory which delegates them to their associated actions.

However, WSAG4J actions are statically plugged into a WSAG4J server instance and cannot be modified at runtime. In order to enable the service provider to modify its service advertisements in an external *Agreement-Mediator*, several ways are identified and are discussed in the following text. Fundamentally, all approaches must reach the requirement defined for an *Agreement-Mediator* as third party: the effort for SLA management has to be reduced and to keep the resulting efforts of all participants as low as possible.

The first and most trivial way to provide a customizable advertisement by the *Agreement-Mediator* on behalf of a service provider is implied by the way advertisements are represented in WS-Agreement. By requesting an agreement factory's *GetTemplate* endpoint an agreement initiator basically gets a set of XML documents in return. Providing a way to modify these XML documents (i.e. some sort of upload) would give the providers comprehensive control about their advertisements. Despite its simplicity, this approach has two downsides: Firstly, if a service provider has to upload his templates, this consequently implies that this provider has in-depth knowledge about the WS-Agreement standard and is responsible for building the templates by himself. Therefore, this approach violates the main goal of this use case. A conceivable workaround for this problem may consist of a graphical front-end that allows service providers to design their templates in a more comfortable way than writing XML while manually complying all specifications by WS-Agreement and all concretizations by the *Agreement-Mediator*. Nevertheless, the applicability of this approach is restricted to cases where the only variable of a provider's offering-affairs is the exposed advertisement while other parts i.e. the decision making about the acceptance of a consumer's offer, remain static and not secret. That is unlikely to be the case for most of the real-world providers. For example, in this model it

is not possible to implement even the most basic constraints of consumers, i.e. only offers by customers having a valid account at the target provider should be considered potentially acceptable. Therefore, a more generic approach is needed, in which a provider can define strategies for template delivery, offer negotiation and offer acceptance.

The main idea of the second approach is to let a provider freely define WSAG4J actions along with its three strategies for template delivery, offer negotiation and offer acceptance. Obviously this approach causes major efforts on the provider's side, but that seems to be an unavoidable sacrifice for the sake of flexibility. Since WSAG4J action strategies are defined by three Java classes, a provider needs the ability to provide the underlying logic. Doing so by uploading Java classes is an unpractical solution for two reasons: Firstly, the *Agreement Mediator* could get injected by foreign code causing security issues like possible information leakage. Secondly, a provider reveals its likely secret strategies. Strategy disclosure can only be prevented if all decision-making processes remain at the provider's side, hence a provider must outsource strategies in a web service. That does not inject foreign code into the mediator and does not reveal a provider's strategies, but providing web services for WSAG4J strategies is near equally elaborate as implementing a WSAG4J instance itself and thus renders to give no benefit regarding the main goal of this proposal whatsoever.

Thinking about possible contents of strategies might lead to a satisfying solution. WSAG4J strategies have well defined results: Either a template is delivered or not, a counter-offer is sent or not, or an offer is accepted or rejected. This behavior can be pre-implemented because it never changes. Which result in particular is rendered depends on certain conditions. So the third approach proposes that the provider only has to define templates and express conditions that lead to the described results. This could be comfortably configured from a pre-defined set of variables in the *Mediator's* back-end. For those cases where conditions shall remain secret or the pre-defined set of possibilities is not sufficient to express a condition, a simple web service endpoint can be provided by the service provider for evaluation. This seems to be the best tradeoff between producing the lowest possible effort on the provider's site, while keeping most possible flexibility and preventing unavoidable strategy disclosure.

3.4. REST as an alternative to WSRF

Software architects and developers have the fundamental choice between two major approaches when creating web services: WS-* based or RESTful web services [42]. Both acronyms describe popular approaches for distributed services: the WS-* family describes a large stack of specifications based on the *Simple Object Access Protocol (SOAP)* [31] while *Representational State Transfer (REST)* is more an "architectural style" [43] than a standard that strongly relies on the *Hypertext Transfer Protocol (HTTP)* as the application-level protocol [44]. WS-Agreement, WS-Agreement Negotiation, and the WSRF protocol family belong to the notable

number of well defined WS-* specifications [45]. Many of these specifications specify interfaces which are usually defined in WSDL documents [32]. The operations defined in a WSDL document have expected inputs and outputs which also have to be defined by a schema document according to the XML standard [30]. With the upcoming of *Web Application Description Language (WADL)* [46] [47] as equivalent to WSDL, such a specification chain can also be applied to REST. Therefore, this section investigates the possibility for porting the WS-Agreement and the WS-Agreement Negotiation specification to a RESTful web service description. The main reason for this purpose is the elimination of the WSRF dependency.

The conceptual development of a RESTful web service description with static interfaces that converts the specified WS-Agreement functionality has been already proposed in two scientific papers [48] [49]. The authors of [48] described a rough architecture for RESTful web services extracted from WS-Agreement. They proposed a resource hierarchy, a set of representation approaches, and some possible HTTP methods with their semantics. Here, the authors limited their work to the synchronous asymmetric protocol variant of WS-Agreement. The authors of [49] presented a similar concept for converting WS-Agreement to a RESTful web service description. Here, also collections had been used for transposing the functionalities of the *AgreementFactory*. In contrast to [48], the authors of [49] additionally presented an *AgreementDraft* collection which enables the implementation of negotiation processes. This concept can be considered as an alternative approach for negotiating SLA terms as specified in WS-Agreement Negotiation. Worth mentioning is also, that in [49] the approach is an implementation of the asynchronous protocol variant which extends the uniform HTTP Interface with an *X-Allow-Deferral* header. This header can be used in a *POST* request to the *Agreement* collection resource for expressing the desired protocol variant and is thus establishing a certain functional parameter. In the same way, also an *Acceptance* resource (which is called Subscription in [49]) can be defined by the link header in order to get notified about decisions in the asynchronous call-back variant. Thus, the same decision as in WS-Agreement has been made in [49]: to model functionally rather than at the level of representable information.

Neither in [48] nor in [49] a reference to an implementation of these concepts is announced. In contrast, in WSAG4J a RESTful web service stack module exists that makes use of the same WS-Agreement core functionalities as used by the SOAP web service module [50]. This RESTful implementation is not documented or conceptionally described, presented or published in scientific papers [40]. A characteristic feature of this implementation is the attempt to convert the functional model of WS-Agreement to the uniform methods of HTTP. In particular, no representations of resources are modeled and for data transmission the same input and output messages are being re-used as in the SOAP web service module, which are also originally defined by WS-Agreement in its WSDL documents. URIs and HTTP methods are designed so that a combination of URI and method can be assigned to each WSDL message. The following subsection opposes WSRF to REST in terms of history, features as well as interoperability and network load. These comparisons are making use of the existing implementation of

WSAG4J.

3.4.1. History and Motivation

SOAP and REST were both invented to establish specifications for machine communication and to avoid creation of proprietary solutions for Business-to-Business (B2B) integration. While SOAP supports a set of different transport protocols [31] the RESTful approach mostly relies on HTTP and aims to unfold its full power by applying best practices from the World Wide Web. This includes e.g. usage of HTTP methods and headers to profit from widely deployed caching mechanisms between communication partners.

Figure 3.2 compares the technology stacks of WSAG4J's implementations based on WSRF and REST. These technological footprints underline the claim of the SOAP based WSRF implementation to support a wide range of transport protocols for complex B2B applications by including protocols such as HTTP(S), SMTP, FTP or JMS. On the other hand it becomes obvious that the RESTful variant of WSAG4J implements its version of WS-Agreement based on a smaller technology and protocol stack by using XML and HTTP(S) only. Both WSAG4J implementations provide service descriptions for their clients: The WSRF variant provides a WSDL document while the RESTful variant supplies a WADL document.

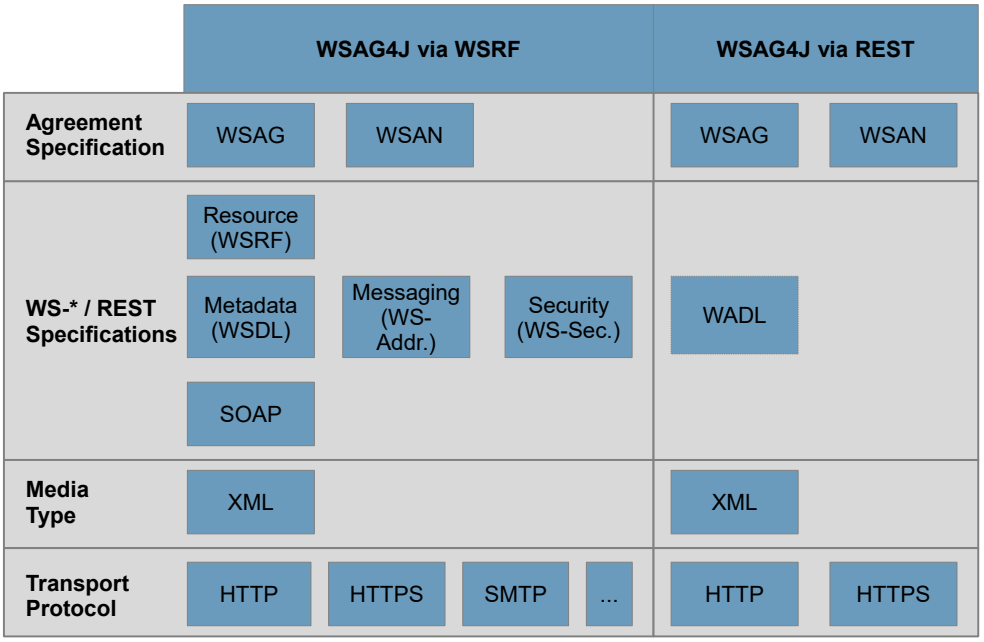

FIGURE 3.2.: Technology stack of WSAG4J via WSRF and via REST

While both variants can be used with HTTP(S), SOAP based WSRF also supports other transport protocols. As most SOAP services are available via HTTP(S), REST profits from a more

specific implementation exhausting more features of the protocol.

3.4.2. Features

Having grown largely within the last 10+ years the family of WS-* standards offers a very wide set of features that ranges from reliable messaging over transaction management to machine-readable service description up to sophisticated security features. While some of these features are already built into HTTP, others have not yet been standardized for RESTful services although they may be of use.

In terms of service description there are comparable specifications in both worlds: The *Web Application Description Language (WADL)* was ported from the WS-* to the RESTful world based on the concept of the well known *Web Services Description Language (WSDL)*. WADL basically enables machine-readable specification of web services via XML. It was developed by SUN Microsystems Inc. and submitted to the W3C in 2009 but has not been standardized [51]. Until today no generic service description format was standardized or has been widely accepted for RESTful services, because it is difficult to find a single format that matches all kinds of services. Instead application specific protocols such as OpenSearch [52] or AtomPubProtocol [53] may be employed depending on the service topic. In other cases linking RESTful resources via media types such as the *Hypertext Application Language (HAL)* [54] may be sufficient to announce a RESTful service's capabilities.

Consequently, WSRF profits from many good WS-* specifications that may exceed existing HTTP features for the RESTful world. Examples in the area of security and service description reveal that REST's concept as an "architectural style" hinders the specification of standards that will apply to every possible application or use case.

3.4.3. Interoperability

RESTful services rely on HTTP as the transport protocol and may be used by any client with access to an HTTP library. The concepts of *uniform interface* and *self-descriptive messages* (also known as *HATEOAS: Hypermedia as the engine of application state*) [44] [43] enable a wide range of different clients such as web browsers, mobile apps and business partners to consume the exact same service using different media types. On the other side SOAP services are not consumable by web browsers because they offer a specialized service to dedicated clients. Any website accompanying a SOAP service would therefore require the creation of a separate interface to serve web browsing users.

These differences reveal perspectives that are also reflected in the way clients are usually implemented. In the WS-* world most services as well as clients are generated by a heavy weight

tool chain hiding underlying complexity. In this way WS-* authors don't need to care about the underlying transfer protocol and start their development process with a set of generated code stubs. The world of RESTful services takes another approach: Typically developers rely less on code generation and try to keep their services available to many kinds of different clients. While the WS-* approach is quite convenient for developers it comes with two major drawbacks: First the requirement of a tool chain restricts the amount of development environments as such tooling may not exist or be incompatible for less common programming languages. Second the abstraction from underlying transport protocols allows developers to start coding immediately at the cost of long debugging sessions when problems arise later on (e.g. due to restrictive firewalls or incompatible schema validators).

3.4.4. Network Load and Performance

Communication is a crucial aspect to performance in distributed systems. Depending on the underlying network protocol and infrastructure, features such as caching or compression may speed up network communication. Because RESTful services are strictly tied to HTTP they can be designed to benefit from HTTP caching either through intermediate middleboxes or on the client side. Caching features available for WS-* depend on the chosen communication technology. Most SOAP tools use POST over HTTP with arbitrary data which renders standard HTTP caching impossible. Although there are approaches to enhance caching for SOAP communication these approaches mostly ignore caching mechanisms of the specific transport protocol but implement caching at higher levels.

Furthermore, performance needs to be weighed up against security requirements for the usage scenario. As an example, using secure communication such as HTTPS makes any caches useless as the actual packet payload is no more visible to intermediate routers, caches or proxies. In the context of the performance comparison presented in section 3.5 the effective response time of both approaches is measured. To gain a better understanding of communication between server and clients within the distributed system, the network load for both cases WSAG4J's SOAP with WSRF and REST module is logged.

3.4.5. Related Work

The comparison of WS-* respectively SOAP and RESTful web services has already been performed by several scientists [42] [44] [55] [56] [57] and many more. However, the comparison of stateful approaches with the intention to include WADL in a WS-* standard is still an open issue. Thus, the following papers either compared both in different contexts or migrated applications between WS-* and RESTful approaches.

Pautasso et al. [44] compared both WS-* and RESTful web services from a conceptual and technological perspective and developed advice on when to use which approach. They presented a general and comprehensive summary to support architectural decisions. In contrast, the focus of this comparison is on a specific standard that requires stateful web services. Furthermore, in the next section also a performance comparison of both approaches is presented.

Upadhyaya et al. [56] provided a semi-automatic approach to migrate existing SOAP based services into RESTful services and compared performance measurements of both solutions showing slightly better performance of REST based services. Compared to their work, this comparison focuses on one single WS-* standard and compares already existing services rather than generating them which allows a more detailed inspection of both solutions.

Mulligan and Gračanin [57] developed a middleware component for data transmission offering both a SOAP and a REST interface. They evaluated their implementations with regard to performance and scalability requirements. Other than their work, this section compares both approaches using very specific WS-* standards: WSRF and WS-Agreement.

Kübert et al. [48] and Blumel et al. [49] are the only ones who analyzed the WSRF based WS-Agreement specification and designed a RESTful service with a feature set close to the standard. Their work proved that porting a WS-Agreement service to REST is possible in theory, but their scope ended with the proof of feasibility. This comparison and the performance comparison in the next section uses an actual implementation of WS-Agreement and gain insights about performance gaps between both solutions. Based on their work as well as the existing RESTful implementation of WSAG4J this section adds an evaluation of both approaches which has not been shown before.

3.5. Performance Evaluation

This section evaluates the SOAP and the REST based implementation in terms of the performance. This comparison is based on the existing REST implementation of *WSAG4J* [50] [40]. Any WS-Agreement service provided by an autonomous *Agreement-Mediator* acts as a neutral component between a service provider and a service consumer for establishing SLAs. As such it needs to be available to both at any time, which implies hard requirements for availability and scalability of such a service. Therefore, the expected performance benchmarks indicate whether the WSRF or the REST based implementation allows to handle more concurrent clients with given hardware. WSAG4J is designed modularly and consists amongst others of an *Engine Module* that implements functionalities for processing agreement offers, creating agreements, feed in monitoring date, and evaluating agreement guarantees, and two *Web Service Modules* (one for WSRF and one for REST) that are the implementation of the web service stack with the sole function to delegate the calls to the *Engine Module*. This makes it an ideal candidate

for black box testing using the provided client and server distributions of both the WSRF and the REST variant to compare equivalent operations.

In order to compare both the RESTful and the WSRF variants, it is important to select a real life usage scenario with significant complexity. A typical use case for WS-Agreement (Negotiation) is automated SLA negotiation which is already used by research projects in the area of fully automated SLA processing [50] [58]. Web services handling SLAs as defined as an overall goal for the *Agreement-Mediator* approach which acts as neutral notary. These web services must always be reachable to both agreement parties guaranteeing verification of established agreements. For this reason, performance, scalability and availability are hard requirements for any production system. In order to measure the performance of both SOAP and REST based communication in a black box approach, the WSAG4J framework in version 2.0 was used. Results of these performance measurements indicate which of both variants allows sufficient service quality at appropriate operation costs.

3.5.1. Test Scenarios

Based on the use case of automated SLA delivery, three sample tests that reflect the WS-Agreement usage in the field of cloud computing were selected. Basically every WS-Agreement (Negotiation) service has to provide at least one *Agreement Factory* containing one or more *Agreement Templates* that describe the provided services and serves as a sample for incoming *Agreement Offers* the customer is willing to accept. For the given scenarios both WSRF and REST services were configured with one *Agreement Factory* holding three templates.

GetFactories

The first scenario is a very basic step in which an *Agreement Initiator* requests all *Agreement Factories* served by the *Agreement Responder*. This use case is usually the first step an *initiator* has to go through to discover services of an unknown *responder*.

GetTemplates

The next scenario reflects the follow-up step in service discovery: The *initiator* needs to gain knowledge about available *Agreement Templates* for any *factory* of interest.

Negotiation Scenario

The third scenario runs a complex negotiation process between the *initiator* and the *responder*. In this case the *responder* implements the agreement on behalf of the service provider while the *initiator* acts on behalf of the service consumer. The offered service computes resources for certain time frames using negotiable templates. Within the scenario the *initiator* sends a first *Negotiation Counter Offer* to which the *responder* replies with another counter offer for less resources at the same time or an equal amount of resources at a later time. The *initiator* evaluates given options and sends a third counter offer which is finally accepted leading to a *Negotiated Offer* used by the *initiator* to create the offer.

3.5.2. Test Infrastructure

The load tests use two commodity servers providing four virtual machines as shown in figure 3.3. Each server was equipped with two Intel Xeon E5430 2.66 GHz CPUs (four cores per CPU) and 32 GB RAM. The nodes were connected via regular Gigabit Ethernet links and ran Linux (kernel version 3.2.0-57). Both nodes ran KVM virtual machines with two cores. Inside the virtual machines Ubuntu Linux 12.04 (kernel version 3.2.0-57) and Java 1.6.0.26 (OpenJDK) was used. Tests were coordinated by using the Java based load testing framework The Grinder in version 3.11 [59].

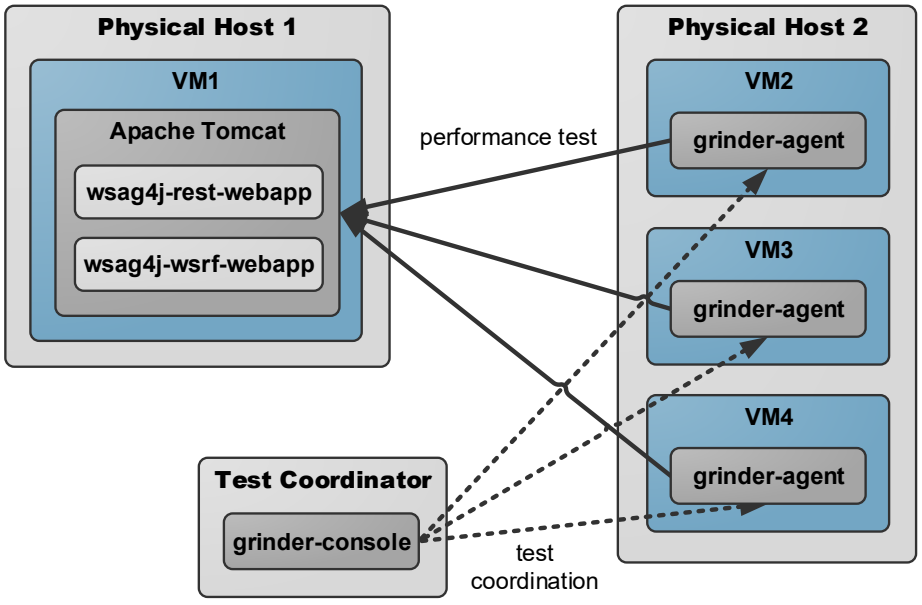

FIGURE 3.3.: Physical architecture of test environment

Host 1 provides *VM1* which ran Apache Tomcat 7.0.50 and serves the WSAG4J web apps with a maximum of 2 GB heap space. For WSRF based tests the WSRF service is deployed while the REST based tests ran with the REST web app. In order to allow dedicated usage of the available heap space only one of both apps is deployed simultaneously. Host 2 provides *VM2*, *VM3*, and *VM4* which execute the test runner component of the Grinder framework named *grinder-agent*. The three agents are coordinated by another host running the Grinder's *grinder-console* component which handle code distribution, test synchronization and collection of measurement results. The distributed code contains specific test code for each test scenario as well as the WSAG4J client for WSRF or REST. This infrastructure allows short network paths avoiding biased results due to network issues while still being close enough to real life scenarios in which clients will always be located on other machines than the WSAG service.

3.5.3. Impact of Security Technology

In the context of SLA negotiation security features like non-repudiation form the technological foundation for general feasibility and acceptance. Since the load tests should handle real life scenarios the setup has been adjusted in order to ensure a comparable level of trust for both approaches: WSRF and REST.

WSAG4J's WSRF based solution utilized WS-Security standards such as *BinarySecurityToken* and *XML signature* [60] [61] by default while the RESTful distribution shipped without adequate replacement. Therefore, the decision has been made to run all tests with HTTPS and to replace WSRF's security tokens with *TLS Client Certificates* which verify the identity of request senders. Because message payload was neither encrypted within the WSRF nor in the REST variant by default, TLS is enabled for both variants considering the sensitive nature of SLAs to protect communication from any kind of eavesdropping.

Table 3.1 shows a comparison of relevant security features. Using TLS and client certificates provides a similar set of security features for RESTful end-to-end-security. Nevertheless this setup could not ensure message integrity if any intermediate host would be able to tamper with the message's content. Given that intercepting a TLS connection would require substantial effort this discrepancy is assessed as negligible for test results.

TABLE 3.1.: Comparison of security implementations

Feature	WSRF	REST
Authentication	BinarySecurityToken (X.509 certificate)	Client Certificates (X.509 certificate)
Signature	XML Signature	—
Encryption	TLS (HTTPS)	TLS (HTTPS)

3.5.4. Measurement Results

The load tests measured 200 test runs for each test scenario with both WSRF and REST code bases. Each test scenario was executed with a different number of concurrent clients to evaluate the scalability of both solutions. All tests started with a single client and increased up to 8 concurrent clients. All JVMs of the Grinder agents as well as Apache Tomcat are restarted after each run to minimize effects of JVM internal optimizations. The test runners *(grinder agents)* are coordinated by a single machine as described in section 3.5.2.

Figures 3.4, 3.5 and 3.6 show response times of all test scenarios. It is important to note that the scale of each diagram's response time axis is adapted to fit the measured values. The results of the load tests reveal that the RESTful code base provides better performance than WSRF in most cases. More specifically, there are only two results which show better response times of WSRF: *GetFactories* with 1 and 2 concurrent users. Starting with 4 concurrent users the RESTful stack performs better.

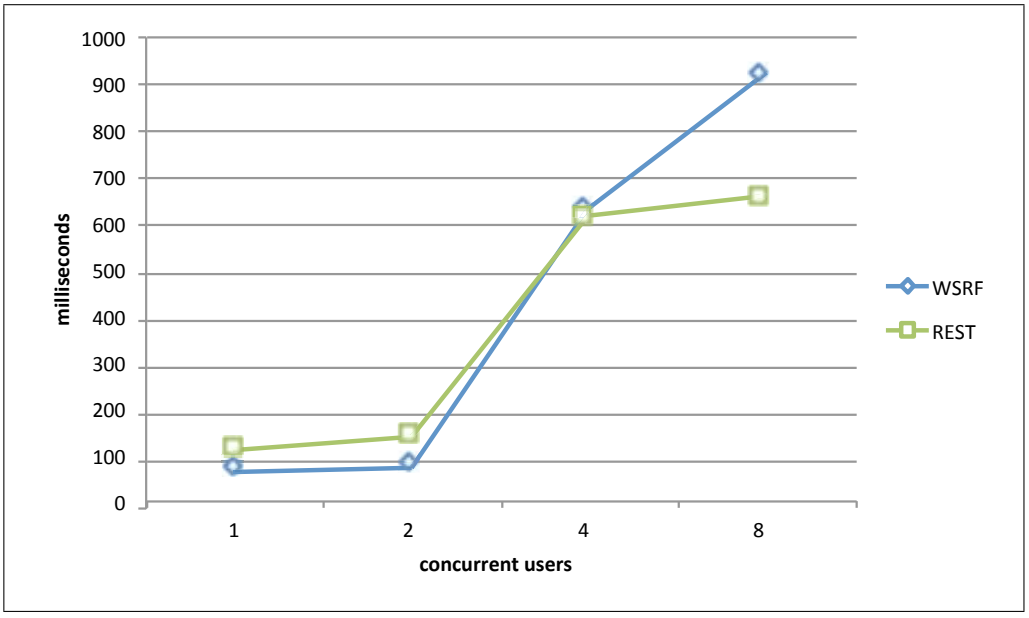

FIGURE 3.4.: Response times for GetFactories

For *GetTemplates* and the *Negotiation Scenario* results reveal lower response time of the RESTful approach in all cases. It is important to point out that while running the *Negotiation Scenario* an increasing number of test failures appeared with rising numbers of concurrent users. Using the WSRF stack the first failures appeared with 4 concurrent users and concerned already 80% of all tests while the RESTful stack showed 59% of failures under the same load. This is also the reason why figure 3.6 only reveals times up to 4 concurrent users. With more than 4 concurrent users the number of failures increased rapidly leading to unreliable measurement results.

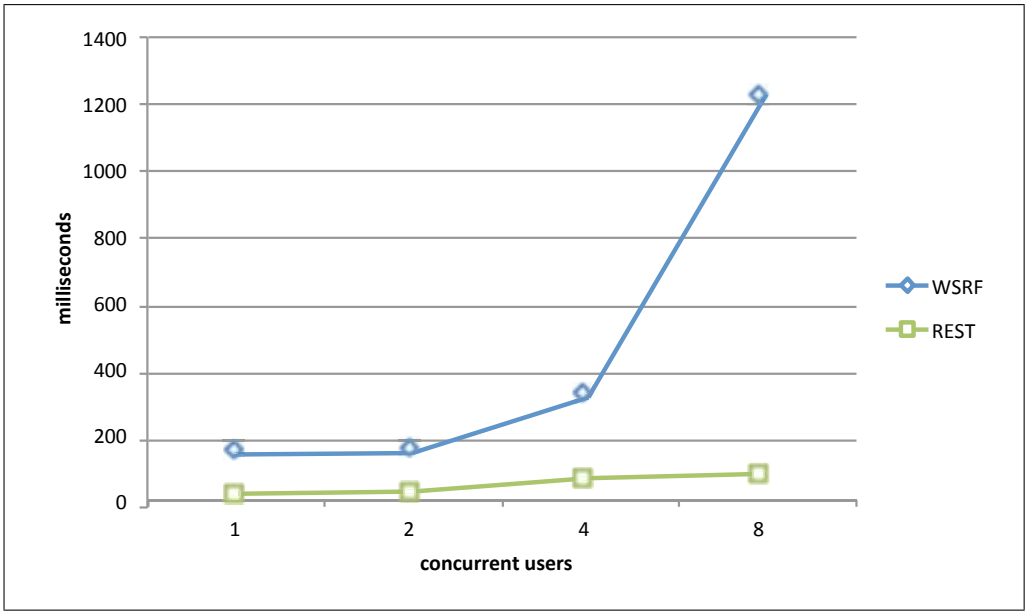

FIGURE 3.5.: Response times for GetTemplates

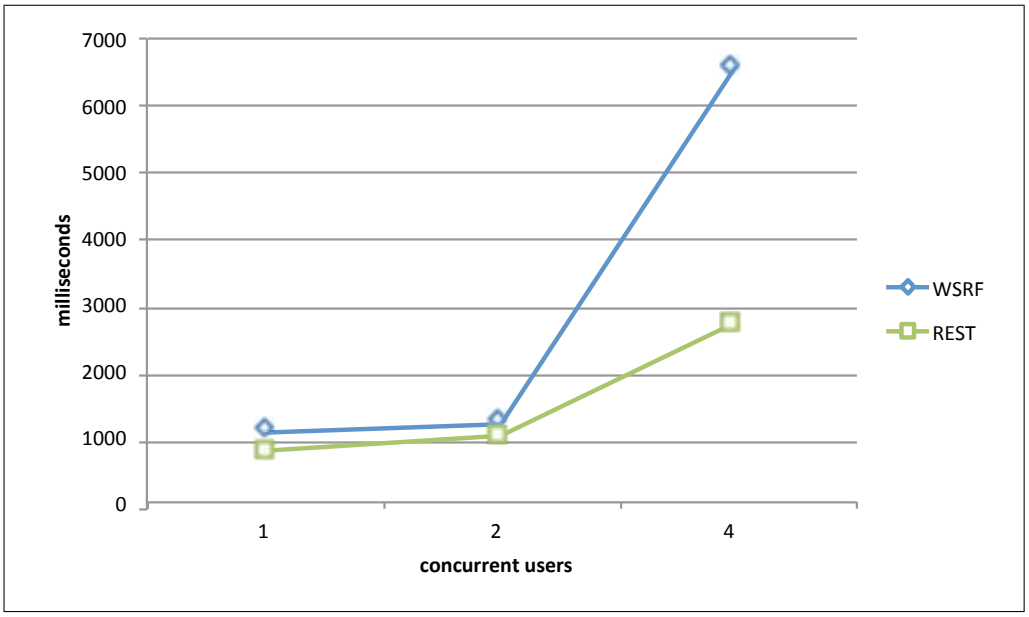

FIGURE 3.6.: Response times for NegotiationScenario

The last figure 3.7 compares response times of all test scenarios proving increased complexity of the last scenario in terms of computation time.

Due to the modularity of WSAG4J, the implementation of functionalities for processing agreement offers, creating agreements, monitoring the service quality, and evaluating agreement guarantees is comprised in the SLA *Engine Module* which is used by both web service stacks: the WSRF as well as the REST. Therefore, the black box approach is adequately, because no separation between performance of internal components like the *Engine Module* and the

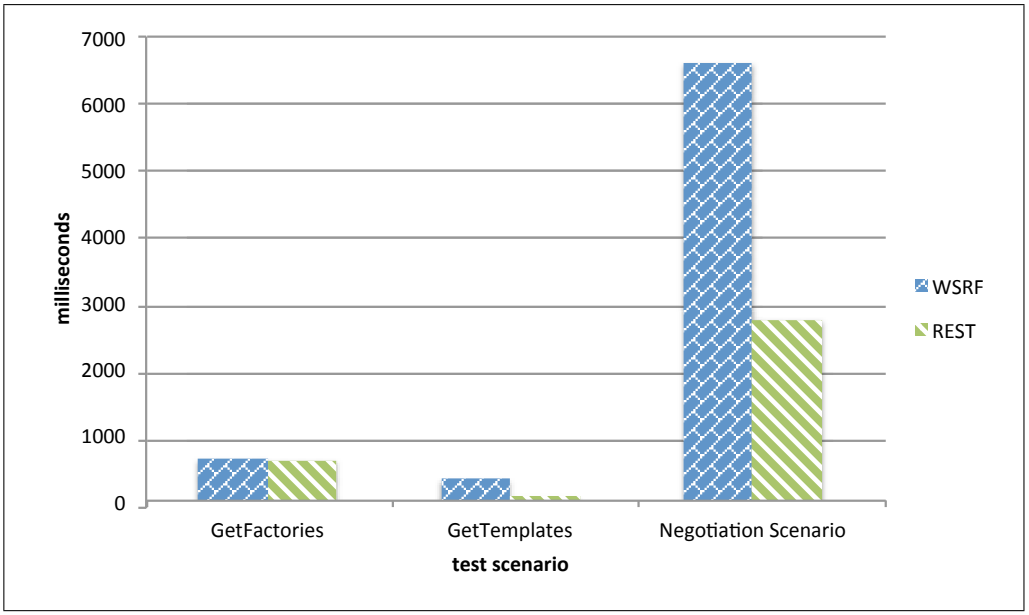

FIGURE 3.7.: Response times of all test scenarios with 4 concurrent users

front-end *Web Service Modules* is applied. The focus of this comparison is on the performance comparison of WSRF to REST where the overhead for parsing the WS-Agreement language, for persistence of agreements or for business logic is equal in both cases. Besides measuring response times, also the amount of network traffic for each solution required during the tests as announced in section 3.4.4 is evaluated. HTTP request and response in the case of *GetFactories* are compared in table 3.2. It shows the request headers as well as the response bodies of both stacks used to communicate about all available factories offered by the service provider. From the last line it becomes apparent that in this sample case REST required nearly one-tenth of WSRF's network traffic by using the very basic media type *text/uri-list* instead of a more complex and verbose XML structure. Both numbers of 3637 and 378 bytes were aggregated over the relevant payload. In order to compare only payloads required for the specific use case, security data such as WS-Security headers or client certificates are neglected.

3.5.5. Conclusion

In terms of performance it becomes apparent with an increasing number of concurrent clients that the RESTful stack of WSAG4J scales better than the WSRF based solution. In the given test infrastructure the *GetFactories* scenario with 24 concurrent REST clients without running into failures while WSRF reported 50% failures with a number of 8 concurrent clients. These results are likely to be influenced by the amount of required network traffic which is significantly larger in the case of WSRF and therefore puts a higher load on the latter's serialization engine.

TABLE 3.2.: Comparison of HTTP traffic

WSRF	**REST**
POST /services/ AgreementFactoryServiceGroup content-type: application/soap+xml user-agent: Axis2	GET /services/factories content-type: application/xml accept: text/uri-list user-agent: Apache CXF 2.6.0 cache-control: no-cache pragma: no-cache
<?xml version='1.0' ...?> <soapenv:Envelope xmlns:...> <soapenv:Header xmlns:...> <wsse:Security> ... </wsse:Security > ... </soapenv:Envelope >	https://... /wsag4j-rest-webapp/services /factories/SAMPLE-INSTANCE-1
3637 Bytes	378 Bytes

It must be emphasized that web services providing WS-Agreement and WS-Agreement Negotiation act as neutral notaries which must by definition always be reachable to both agreement parties enabling 24/7 verification of SLAs. Such availability can be achieved by typical measures such as redundancy and instant provisioning and must scale with an increasing number of clients. As proved by measurements the RESTful implementation of the WSAG4J framework scales better than the WSRF based solution and can therefore reduce operation costs and complexity when using WSAG4J for SLA negotiation and monitoring. The second reason for opting for the REST-based solution is its enhanced interoperability compared with the WSRF variant. When providing a public *Agreement-Mediator* service it is advisable to support as many different clients as possible. Because REST's technological footprint is lighter than the one of WSRF, it is open to more development environments possibly attracting a larger number of users. Therefore, the next section presents further REST-based standards that are considerable for an *Agreement-Mediator* approach or may be a fastening element to WS-Agreement and WS-Agreement Negotiation.

3.6. Further REST-based Cloud Standards

Many companies started working on their own proprietary specifications for virtual machine configurations, their associated file formats, the application packaging and its deployment through proprietary interfaces [8]. This lack of interoperability has been solved by standards like the Open Virtualization Format (OVF) [9], the Topology and Orchestration Specification for Cloud Applications (TOSCA) [11], the Cloud Application Management for Platforms (CAMP) [62], the Cloud Infrastructure Management Interface (CIMI) [63], and the Open Cloud Computing Interface (OCCI) [10].

The OVF [9] is a specification that describes an open format for the packaging and the distribution of software to be executed in virtual machines. This specification was standardized by the Distributed Management Task Force (DMTF) and enables the packaging of distributed applications in a secure, vendor and platform independent, efficient, and extensible format. Applications that are delivered in such a format can be easily migrated from a local cloud to a public cloud or from one public cloud to another independent of the cloud middleware that is used by the provider.

TOSCA [11] is a specification that aims to leverage portability of application layer services by providing a formal description of *Service Templates*, including their structure, properties, and behavior. TOSCA is standardized by the Organization for the Advancement of Structured Information Standards (OASIS) and specifies an XML-based language that is used to orchestrate software components and their relationship as well as their behavior.

These two open standards mentioned above are of high importance, because they build the foundation for further standards and are widely supported by several cloud middleware implementations. However, these standards describe the application or virtual machine configuration and packaging format. They don't specify any kind of operations which can be used to deploy or in general enable the access to a cloud middleware. This kind of interface specifications are specified in the other already mentioned standards, which are described and analysed in the following subsections.

3.6.1. Cloud Application Management for Platforms

The CAMP [62] standard is a RESTful API specification for PaaS cloud middlewares. It describes operations for managing the deployment, running, administration, monitoring and patching of applications in the cloud. CAMP is standardized by the OASIS and specifies artifacts, formats and APIs which enable interoperability among self-service interfaces of PaaS clouds. Use cases that are supported by CAMP are [62]: building and packaging an application in an application development environment, importing a deployment package or uploading

application artifacts into the cloud, as well as run, stop, suspend, snapshot, and patch an application.

The CAMP standard is open and defines also support for multiple endpoints, versions, and extensions that can be used for extending a cloud middleware with additional functionalities. The basic specification has absolutely no support for any SLA functionalities. Furthermore, CAMP is designed only for the PaaS layer and is strictly bounded to HTTP. For this reason, the CAMP standard is not applicable for any *Agreement-Mediator* approach in general.

3.6.2. Cloud Infrastructure Management Interface

The CIMI [63] is a specification that describes a model and a protocol for management interactions between cloud service consumers and cloud service provider. Originally, the Open Cloud Standards Incubator initiative developed cloud management use cases, architectures and interactions which were transitioned to the Cloud Management Working Group (CMWG) that specified the CIMI.

CIMI specifies a RESTful protocol based on HTTP that enables access to an implementation of an IaaS cloud middleware. CIMI may also be adopted to higher cloud layers such as PaaS or SaaS, but is out of scope of this specification. The basic resources for which CIMI is modeled are virtual machines, storage, and network. The media types that are supported by CIMI are either *application/json* or *application/xml*.

The model of CIMI focuses on a single administrated cloud environment where the customer has already established a business relationship to the provider. In particular, financial terms like accounting, billing, or payment are out of the scope of this specification. Furthermore, the model assumes that the customer has already signed up and has to support provider specific authentication and authorisations implementations which are also out of scope of this standard.

The CIMI model has been inspired by the Entity-Relationship model, because it makes use of a tabular representation where each entity is modeled as cloud resource and has its relationships to other entities by using a referential mechanism. The references of this referential mechanism are unique identifiers which are URIs for HTTP. The model should be self-describing in order to support extensibility in two different ways. While the first extensibility mechanism defined by the CIMI model is for consumers, the second is for providers [63]:

Consumers can add additional data to their resources when creating or updating a resource. For this purpose the specification defines an attribute called *property* for each resource. This attribute can store a set of name/value pairs which are stored and returned by the provider.

Providers can define extensions by making use of the *ResourceMetadata* resource for this purpose. It allows for advertising additional attributes, operations, and capabilities. Moreover, constraints can be defined that might need to be understood by customers. This model enables the discovery of any new extension attributes or operations.

The extension model of CIMI is a well designed approach and makes this specification considerable for the application to an *Agreement-Mediator* approach. However, the CIMI is designed for HTTP although the specification explecitly mentioned that "... the underlying model is not specific to HTTP, and it is possible to map it to other protocols as well" [63].

3.6.3. Open Cloud Computing Interface

The OCCI [10] is a set of specifications standardized be the Open Cloud Computing Interface working group (OCCI-WG) within the Open Grid Forum (OGF). It is a RESTful protocol and API for management tasks that was originally designed for IaaS services, but in its current maturity also supports PaaS capabilities. This open standard aims to provide a cloud middleware independent interface to access cloud management functionalities (e.g. instantiation and configuration of virtual machines, setting up associated networks, etc.).

In contrast to the CAMP or the CIMI standard, the OCCI is a specification family that follows a layered approach where the *Core* specification builds the foundation and all others are built thereon. The current version 1.1 consists of three documents which are described in detail and analysed in the following sub-sections.

OCCI - Core

This specification is the foundation for all other OCCI specifications and defines the OCCI core model [64]. This model specifies a representation of instance types which can be accessed and manipulated through operations defined by higher-level specifications. The main goal of this model is its extensibility. In particular, customers are able to explore and to discover all functionalities and extensions at run-time. The extension mechanism that is used here, is based on the *Mixin* concept [65].

The core model can be divided into three categories: classification and identification types, base types, and extensibility types. Three base types are defined in the core model and are depicted in figure 3.8. Here, an *Entity* is an abstract type from which *Resource* type and *Link* type inherit from. The resources are modeled according the REST-style as sub-types of the *Resource* type which could be a virtual machine, a network or any other entity. The *Link* type is used to associate *Resource* instances to other *Resource* instances. In order to identify resources and their classification, each sub-type of *Entity* has a unique *Kind* instance.

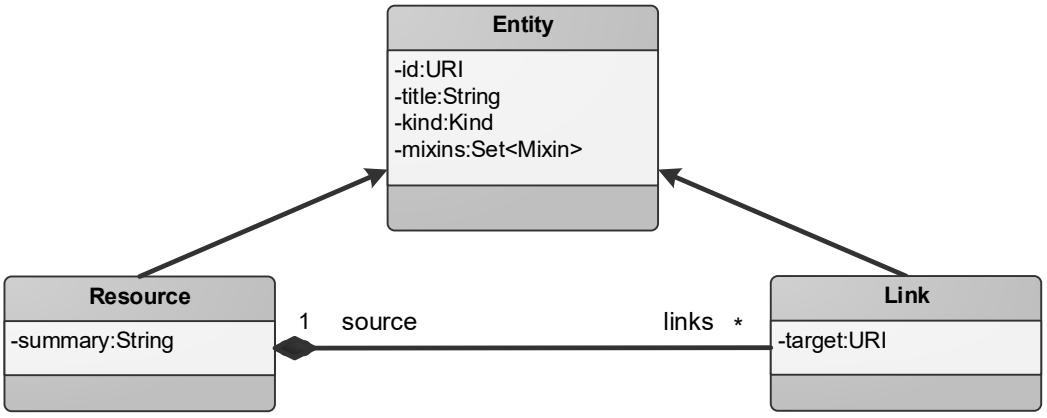

FIGURE 3.8.: OCCI base types of the core model

The fundamental type of the classification and identification system is the *Kind* type. This *Kind* type inherits from the *Category* type which is also the abstract base type of the *Mixin* type and the *Action* type as depicted in figure 3.9. A *Kind* instance is an unique identifier that allows for dynamic discovery of available types. In fact, the *Kind* type is used to expose the functionalities that are supported by a particular implementation. Each *Kind* instance can expose a set of *Actions* that can be invoked on a *Resource* instance that applies to this particular *Kind* instance. In order to expose additional capabilities as extensions at run-time, the core model specifies the *Mixin* type. An instance of this type can expose higher-level functionalities which are optional and domain-specific.

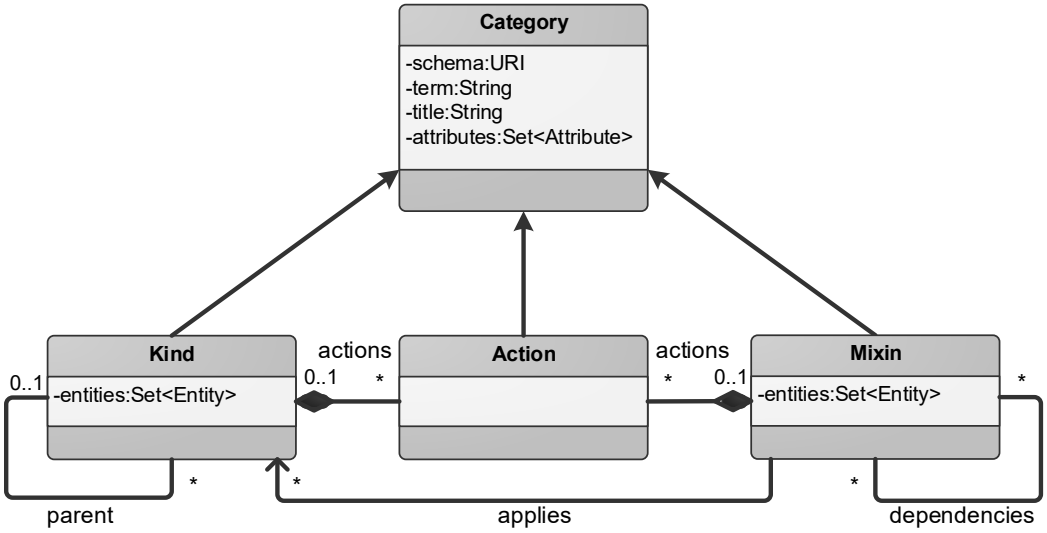

FIGURE 3.9.: OCCI classification types of the core model

Additionally to the core model types described before, the OCCI core specification also defines *Attributes*, *Templates*, and *Collections*. While Attributes specify parameters and their properties, *Templates* are used as mechanisms to provide default values for entity instances. A *Collection* is a set of *Entity* instances associated with the same *Kind* or *Mixin* instance. Such a *Collection* can be used by clients to navigate through resources. Here, a client can retrieve a whole *Collection*, a specific item in a *Collection*, or a subset of a *Collection*.

OCCI - HTTP Rendering

The RESTful HTTP rendering specification describes a RESTful API that enables the interaction with the core model [66]. It specifies the general behavior for all interactions and three content types for the representation of resources: *text/plain*, *text/occi*, and *text/uri-list*. It is based on HTTP and makes use of its standard operations: Create (POST), Retrieve (GET), Update (POST/PUT), and Delete (DELETE). Besides the specification of behavior and access to *Resource* instances, *Kind* collections, *Mixin* collections, and also a query interface is specified. This query interface can be used to determine the capabilities on a specific OCCI implementation. Additionally, a filtering mechanism is described in order to retrieve selected information of interest.

OCCI - Infrastructure

The Infrastructure specification is an extension designed for interactions with an IaaS cloud middleware [67]. It describes three sub-types of the *Resource* type, which are *Compute*, *Network*, and *Storage*, and two sub-types of the *Link* type: *NetworkInterface* and *StorageLink*. The following listing 3.1 illustrates an example for creating a compute resource instance.

```
> POST /compute/ HTTP/1.1
> [...]
> Content-Type: text/occi
> Category: compute;
    scheme="http://schemas.ogf.org/occi/infrastructure#";
    class="kind";
>
< HTTP/1.1 200 OK
< [...]
< Content-Type: text/plain
< Location: http://cloud.cit.tu-berlin.de/users/as/compute/vm1
< Server: cit-occi OCCI/1.1
<
< OK
```

LISTING 3.1: Example request/response for OCCI

4. Agreement Mediation Approaches

Contents

The overall goal of this thesis is the development of an autonomous web service based *Agreement-Mediator* as a third party that sources out SLA management tasks into a neutral zone without taking the consumer's or the provider's side. Thus, the *Agreement-Mediator* adds an additional transitive relationship between service providers and service consumers. Moreover, the introduction of an *Agreement-Mediator* as depicted in figure 4.1 facilitates service discovery mechanisms and allows for comparing services based on their advertised QoS. Furthermore, customers are able to negotiate individual SLAs before they come into effect. In particular, the *Agreement-Mediator's* main purposes are:

- advertising services with individual quality levels and guarantees that a provider is willing and able to deliver;

- giving consumers the opportunity to discover and to easily compare offered services of different providers;
- registering declarations of mutual intentions for business relationships in form of potential SLA offers;
- offering services to feed in monitoring data for participating parties of an agreement for automatic compliance verification of SLA terms;
- notifying participating parties about agreement-related events like creation, violation or termination;
- reducing the overhead for managing SLAs between consumers and providers; and
- establishing higher trust by transferring mediation affiliation into an external and neutral entity.

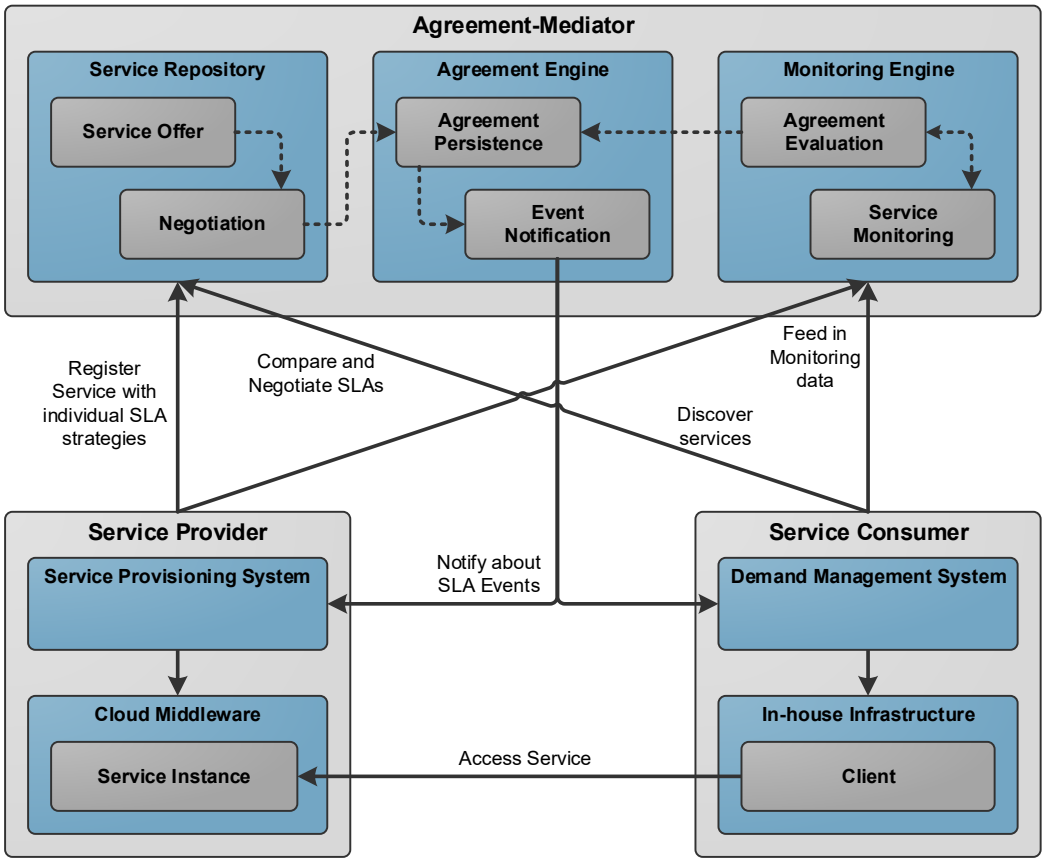

FIGURE 4.1.: Schematical architecture for an *Agreement-Mediator*

The requirement for such an *Agreement-Mediator* is that this approach has to make use of already accepted and widely established standards for SLA and cloud management protocols. Further requirements include the applicability to broader service domains in order to use this approach also for every cloud-based service form the IaaS up to the SaaS layer. Similarly to traditional SLAs, agreements should describe the quality of a service, and what has to be monitored to verify agreement compliance. The *Agreement-Mediator* should represent multiple providers at the same time, i.e. the *Agreement-Mediator* advertises SLA offers of different providers at one central place in a standardized and normative way. This increases the customer's ability to compare these offers of different providers and to negotiate specific terms and conditions with them in order to find the best fitting service. Thus, consumers are able to make better decisions about which provider they want to work with and the providers' competition among each other regarding offers with the best relation between cost and terms is potentially increased. Furthermore, a normative approach for the expression of agreement offers allows for rapidly changing terms, advertisements, and pricing conditions for cloud services offered by provider around the world. Thus, cloud service provider could benefit from this approach, especially in case of cyclically idle times. In particular, providers are able to increase the utilization of their data centers by advertising their prices depending on their capacity. For instance, a virtual machine can be sold for a lower price at night and for a higher price at day.

The following sections of this chapter present different approaches of *Agreement-Mediators* which are evaluated in order to find the best solution for the overall goal of this thesis. While section 4.1, 4.2, 4.3, and 4.4 present the approaches in detail and assess them, section 4.5 presents further related work and approaches. Finally, section 4.6 concludes this chapter and gives an outlook on the last major chapter of this thesis.

4.1. Registry Approach

Service-orientated architecture (SOA) is a software architecture concept that enables the development of loosely coupled, standards-based, distributed, and protocol-independent services [68]. These services can be provided by autonomous organizations and would build a cross-organizational service landscape in this way. Each service can be combined with other services and together construct a higher functionality of a large software application. This paradigm not only facilitates distributed service-orientated computing environments but also reduces development costs because it allows higher flexibility of business processes by reusing existing services.

In order to discover services in a SOA, a kind of service repository has to provide the functionality to retrieve information about the services that are published in a registry. Such information is the location of a service, additional contact and support items, or financial information (e.g. license fees). In order to distinguish between registry and repository the following definitions

are applied for this thesis: A repository represents the business view and a registry the technical view of a service. Thus, a repository contains more information and data than a registry. In particular, while a registry is used primarily to manage the services used at runtime, a repository also contains additional information for considering a service before use (e.g. pricing information, auditing, logging, etc.).

A service registry enables service consumers to find a desired service. One existing standard that specifies the data model and the interfaces for accessing an implementation of a service registry is the Universal Description, Discovery and Integration (UDDI) [69] standard. This specification defines a SOAP-based API for retrieving fundamental information such as the description of a service, a service interface definition, and a description of how to access the service. When a service consumer has found an adequate service in the registry, the web service is accessed directly according to the description retrieved from the registry.

As depicted in figure 2.6, WS-Agreement introduces an additional agreement layer that enables the management of agreements for specific services. In order to discover services with specific service levels, an agreement management component has to look up available services in a public registry [70] [50]. If a service with desired quality characteristics was found, the consumer's agreement management component contacts the agreement management counterpart on the service provider's side. This scenario is illustrated in figure 4.2. Here, both components are able to negotiate the conditions of the service provisioning on the agreement layer. If both sides agree to the negotiated conditions an SLA is created that serves as formal contract. Afterwards, the service provider creates a service instance at the service layer that can be used by the consumer with the agreed quality.

4.1.1. Service Visibility and Discovery

UDDI registries can have two possible visibilities: private and public. Private registries are located in an infrastructure of a single organization and are not accessible from the Internet. Public registries are a collection of peer directories which are available for external customers. Today, only a few public UDDI registries are still available, because they were mostly small from the very beginning and have not been widely used by customers. The reason for this may be the human processes, e.g. the creation of an initial business relationship between customer and provider, the negotiation of price and payment terms, and so on, that need to be performed before the automated processes of service discovery can happen. This fact makes this approach for applying to an *Agreement-Mediator* that allows for discovering services of third-party provider unusable.

However, UDDI is still often used as supporting infrastructure for SOAP web services in a constrained private environment. The field of application of today's UDDI registries is to enable service discovery and to supply additional information missing in WSDL descriptions for web

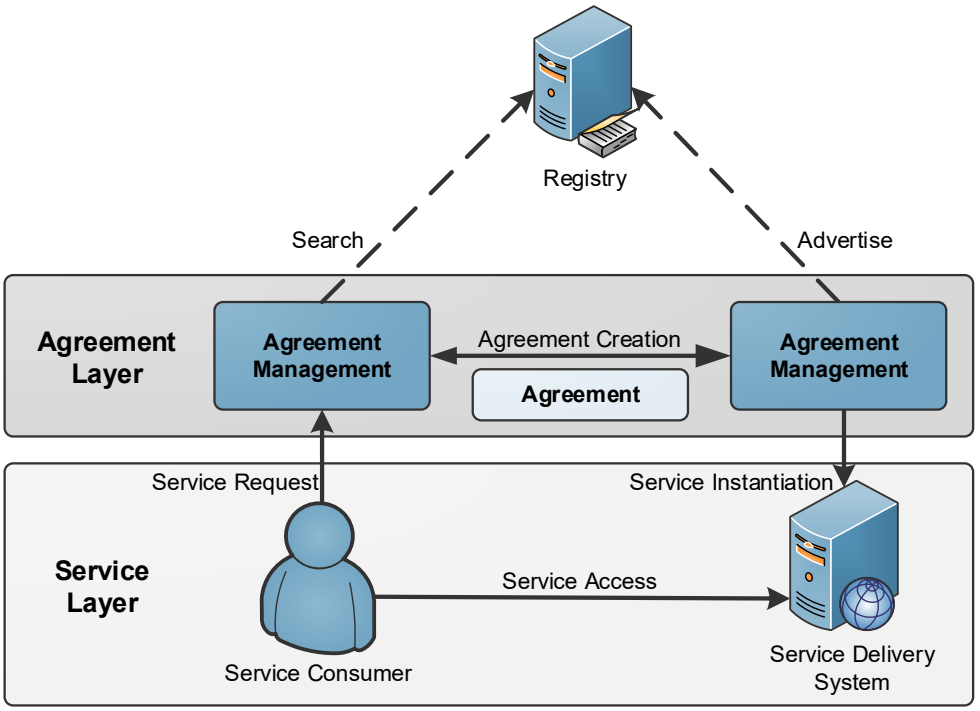

FIGURE 4.2.: SLA aware service discovery

services, which are operating inside a single infrastructure and may be only available for a set of privileged customers. Taking this case for this approach, one of these privileged customers can browse for a desired service, but cannot compare this service to other equal or similar services offered by different providers in the same fashion. In order to compare services based on the advertised service levels, the *Agreement Management* component of the consumer has to search in different registries for equal services and to retrieve agreement templates from all *Agreement Management* components hosted at provider side. These agreement templates include the advertisements of each service exposed by the different providers and build the foundation for the comparison. This makes not only the implementation and the comparison very complex, but also requires the establishment of business relationships to all providers which have to be taken into account for this comparison.

4.1.2. Complexity and Automation

UDDI was not widely used for exposing and discovering web services in a public manner, because of its overambitious complexity, the missing security capabilities, and the difficulty of managing and collecting micro-payments. Besides the use of UDDI, an implementation of a registry for web services requires in any case the development of an *Agreement Management* component for both, the provider and the customer. These components also have to verify the

compliance of an established SLA by monitoring the service during consumption. This fact does not facilitates the SLA management, does not establish higher trust and especially does not actually transfers mediation affiliations into a neutral zone.

4.2. Broker Approach

The approach for an *Agreement-Mediator* acting as a broker between customers and providers requires an additional extension that is based on WS-Agreement and WS-Agreement Negotiation. This protocol extension introduces a new layer located on top and is needed for administration and configuration tasks. Similar to the *Agreement Factory* and the *Negotiation Factory* the Administration layer has to provide a *Service Repository* that allows the definition, the creation, and the configuration of Agreement Terms (*Service Description Terms, Service References, Service Properties, Guarantee Terms*) and Agreement Strategies (*Create Template, Negotiate Offer, Create Agreement*).

This section does not specify the extension itself, but rather identifies the requirements that such a specification should fulfill. Therefore, this section presents details on the functionality of each service and in particular how WS-Agreement (Negotiation) and its reference implementation WSAG4J are exploited in order to provide the necessary services for SLA management.

4.2.1. Advertising SLA Temples

The *Agreement-Mediator* in this approach advertises provider's SLA capabilities with WS-Agreement templates that are delivered by invoking the *GetTemplates* operation from a *Negotiation* instance or an *Agreement Factory*. As described in section 2.3, templates contain guarantees and optional compensation methods referenced to services a provider is likely to accept. Here, a service provider has to use a publicly available term language defined by the *Agreement-Mediator* to describe services or to create references to existing descriptions. The actual SLA is formulated by service properties and guarantees following the standard WS-Agreement format including concretizations made by WSAG4J like the use of JEXL as expression language.

As depicted in figure 4.3, each service provider which cooperates with the *Agreement-Mediator* has one assigned *Service Repository*, one *Negotiation Factory*, and one *Agreement Factory*. This gives the ability to create, configure, and delete its actions by using an interface of the *Agreement-Mediator*. This interface provides a comfortable way to design templates without the requirement to be familiar with the WS-Agreement standard. If a provider is familiar with the WS-Agreement standard though, he can modify the templates directly. Additionally, the *Agreement-Mediator* interface allows inserting placeholders for whole templates or parts of them that are linked to endpoint references of web services. These web services are then

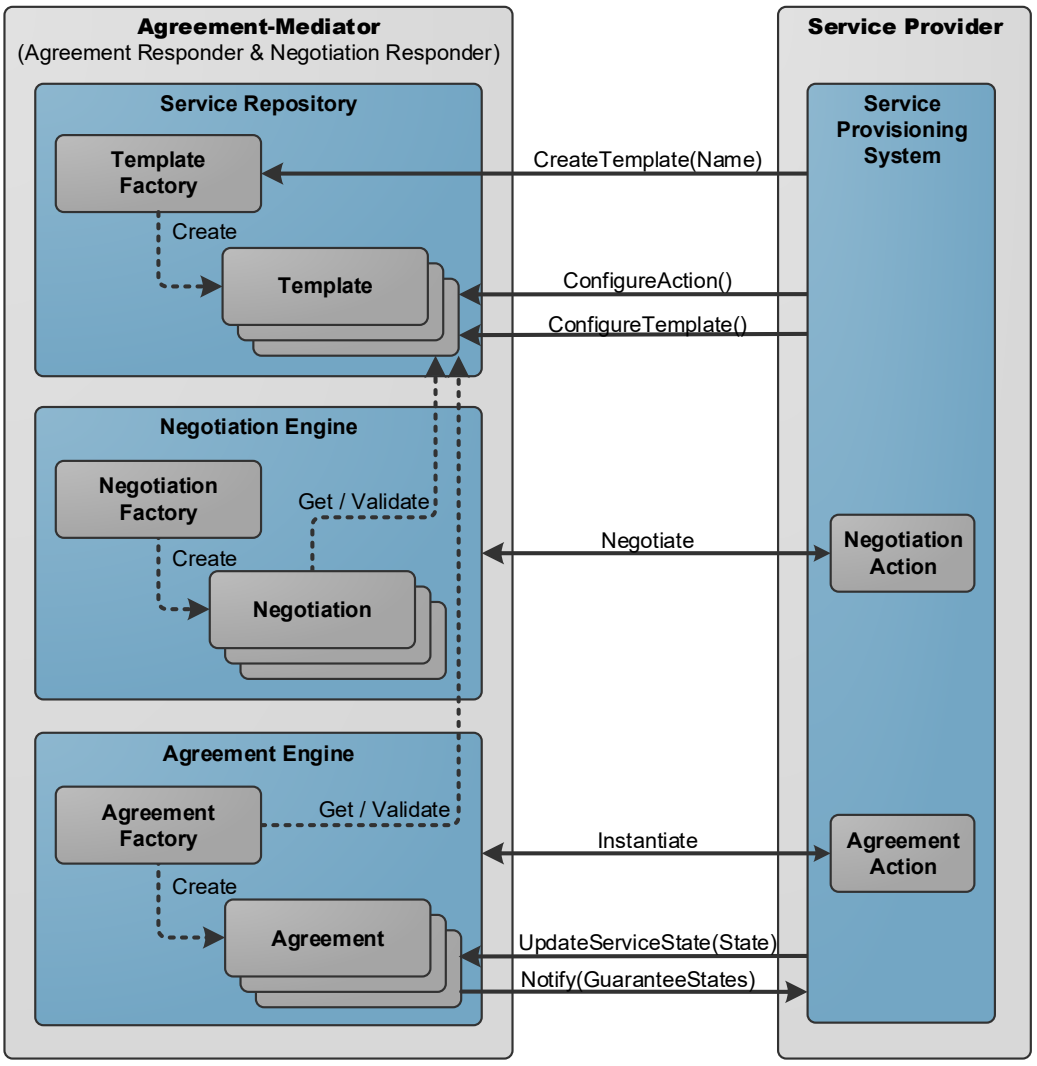

FIGURE 4.3.: Provider's architecture for administration

used by the *Agreement-Mediator* to resolve the placeholders when the template needs to be delivered. As described before a WSAG4J factory action consists not only of a template but also of all decision-making processes that lead to template delivery, negotiation and agreement creation. These strategies can also be freely customized by manipulating them directly or by creating a delegation to an own endpoint reference that serves with the required functionality. Thus, a provider can freely configure the connection between his state of available resources and his advertisements, which is a key requirement for a provider in order to offer the most attractive terms to customers at any time.

4.2.2. Discovering and Comparing service offer

Agreement offers are advertised to and initiated by customers, so in terms of WS-Agreement the customer takes the role of an *Agreement Initiator* and the *Agreement-Mediator*, which is the advertising party that later holds agreements for supervision, takes the role of the *Agreement Responder*. Therefore, a customer gets a set of provider's templates by calling the *GetTemplate* endpoint of the provider's *Agreement Factory* as illustrated in figure 4.4. All templates delivered by the *Agreement-Mediator* follow a standardized format that is publicly available and is partially defined by WS-Agreement and partially by the *Agreement-Mediator* that fills the definition gaps explicitly left open by WS-Agreement. Thus, the customer knows exactly how a template is structured and therefore how to (programmatically) compare two templates even from different providers. This enables the customer to rapidly find a provider's offer that matches his requirements best.

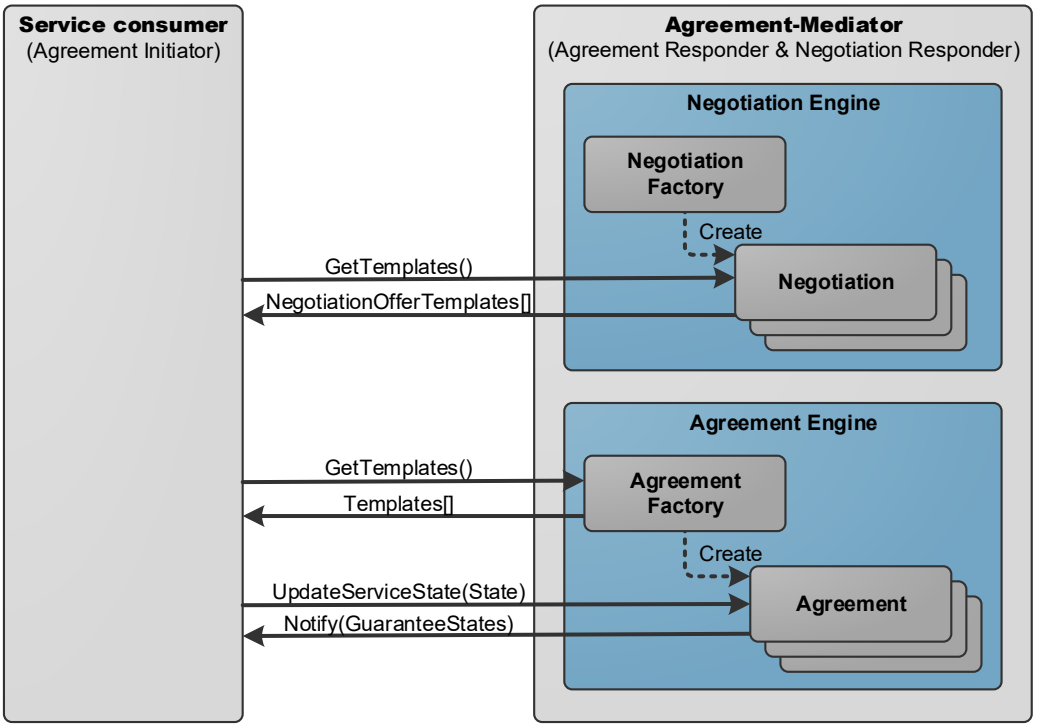

FIGURE 4.4.: Customer's interaction with the *Agreement-Mediator*

If a customer has browsed a desired service and prefers for a service offer of a selected provider, the customer is able to start a negotiation process in order to advance a service offer for his requirements before an agreement is created. Therefore, the customer creates a *Negotiation* instance for a specific provider and performs the negotiation according to the WS-Agreement Negotiation standard.

4.2.3. Feeding in monitoring data

SLOs are formulated by expressions that utilize variables that represent service properties (e.g. availability). The values of these variables need to be determined and set by the participating parties. For that reason the *Agreement-Mediator* provides two endpoints per defined variable of an agreement for both the consumer and the provider to feed in their view of the situation. But the example of a variable representing availability illustrates that in some cases it is impracticable that agreement participants directly determine a variable's value: To calculate a service's availability, one has to constantly check a service's status, store this data and continuously calculate the overall availability. For that reason the *Agreement-Mediator* allows it to express the values of variables as calculations by constructing mathematical formulas using other variables. The agreement participants can then set the variables of these calculations. Doing so, the *Agreement-Mediator* does not take the responsibility for monitoring the services from the agreement participants, but supports them to reduce their overhead.

4.2.4. Getting notified about SLA events

Possible agreement events are instantiation, termination and change of state. The defined endpoints of WS-Agreement allows for inspecting all of these events, so users can use them at any time. But since this is very inefficient, because it requires to constantly poll states and compare them to older ones in order to detect changes, the *Agreement-Mediator* offers users the possibility to register an endpoint reference that is used to get informed about agreement events. Beyond WS-Agreement the *Agreement-Mediator* also offers the possibility to inspect the data, that lead to a certain event or state. That way customers for example not only get informed about an agreement violation, but also get data at hand that might be necessary to request a compensation from a service provider.

4.3. ESB Approach

MO-BIZZ is an ecosystem that provides infrastructure, application, and API services of autonomous organizations. This ecosystem is build for business services, where companies are part of a larger ecosystem and have symbiotic relationships with customers, suppliers and competitors. The applications and services offered by MO-BIZZ are available through a marketplace, which is a web application that is accessible by a web browser. In the MO-BIZZ context, it is essential that specific services are provided according to predefined service levels, e.g. compliance with maximum transaction times or service availability, because a service may be consuming other services offered in this ecosystem. Therefore, the MO-BIZZ system enables

service providers to define service level agreements containing all expectations and obligations of the business relationship.

The four solutions which act together in order to build the MO-BIZZ ecosystem are: the **Marketplace**, the **Service Delivery Broker (SDB)** [71], the **Ontologies Based Enterprise Application Integration (ONAR)** framework [72], and the **SLA Engine** which is based on the Web Service Agreement for Java (WSAG4J) framework [40] customized for MO-BIZZ. The core of this ecosystem is an Enterprise Service Bus (ESB) that provides an integrated solution for enterprises which aims to move their business applications to the cloud. Thus, the platform enables both the access to infrastructure and the access to platform services, in order to develop cloud-based applications. Since this ecosystem provides appropriated mechanisms for SLA management like the definition of transparent SLOs with corresponding guarantees, this MO-BIZZ ecosystem also has to ensure the fulfillment of promised QoS characteristics (e.g. availability, throughput, downtime, bandwidth, response time, etc.). Therefore, SLAs are established, services are monitored, and their compliance is evaluated. Thus, MO-BIZZ is a service platform where cloud services can be discovered, appropriated SLAs are managed, and their compliance is monitored by a third-party entity.

4.3.1. Architecture

The MO-BIZZ architecture joins multiple components as illustrated in figure 4.5. On the bottom, services are provided as registered API services or as external cloud services. Directly above, the following core components of MO-BIZZ are depicted:

- **Marketplace** allows customers to browse for a desired service, to compare this service to other equal or similar services based on SLAs, and to select an appropriated service for consumption according to the advertised SLAs [73].
- **Service Delivery Broker** (SDB) provides an Identity Gateway for bridging between identity consumers and identity providers, a Backoffice for API lifecycle management and configuration, and an API Gateway for processing API requests [71].
- **ONAR** enables semantic conceptualization of cloud-based enterprise applications by using domain ontologies [72].
- **SLA Engine** provides SLA functionalities for defining customized agreement templates, creation of agreements, and evaluation of established SLAs [40].

The SDB API Gateway [71] is not only a core component of MO-BIZZ that exposes fundamental provider services, but is the essential component that strikes together several powerful technologies to this MO-BIZZ platform. In particular, it provides mediation between service

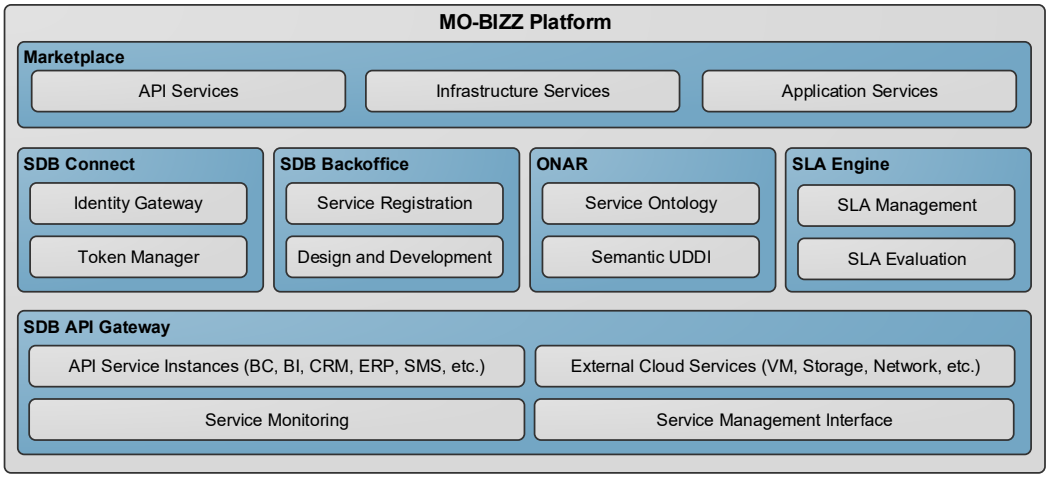

FIGURE 4.5.: MO-BIZZ layered architecture

enablers and their consumers (applications or other service enablers) providing a loosely coupled, highly distributed integration network platform, with enterprise-strength performance, scalability and manageability. Combining messaging exchange, data transformation, intelligent routing and several transversal functionalities, to reliably connect and coordinate service enablers according to the service management decisions.

The SDB Connect is a bridge between identity consumers (e.g. web apps) and identity providers (e.g. Google, LinkedIn, LDAP). Front-channel user authentication is handled by the SDB Identity Gateway, which mediates and manages connections to external identity sources, supporting multiple identity providers, interaction protocols and security token formats. The Identity Gateway relies on several authentication-related support services, which it accesses via SDB API Gateway.

The SDB Backoffice is the system administration application, enabling API life cycle management and configuration of the different SDB components. It is a private web application, built on top of the SDB API Gateway and Support Services. The SDB Support Services are a set of SOA services that support most of the product logic and therefore can be replaced or extended by other SOA services that implement the interface or easily integrate the SDB with other applications. In fact, this SDB Support Services are the interface between the Marketplace, the SDB Backoffice and the SDB Gateway, which makes the MO-BIZZ ecosystem a web service based ESB that can be considered as an approach for an autonomous *Agreement-Mediator*. The APIs exposed on the SDB can be made available as products with different business models. On the Marketplace developers can purchase access to APIs and manage their product usages.

4.3.2. SLA Engine

The SLA Engine of MO-BIZZ provides a set of basic SLA templates for a partner (service provider). This SLA templates are either infrastructure, API, or application specific and guide the service provider through the process of describing an appropriated agreement offer. Here, the provider has to define the SLOs that the services are aiming to achieve. Additionally, the partner can specify compensations methods that come into account if guarantees are violated.

As illustrated in figure 4.6, the Marketplace allows customers to browse for a desired service, to compare this service to other equal or similar services based on SLAs and to select an appropriated service for consumption. After a service consumer has selected an appropriated service, this service is delivered either through the SDB in case of an API service or as external cloud service referenced by the SDB.

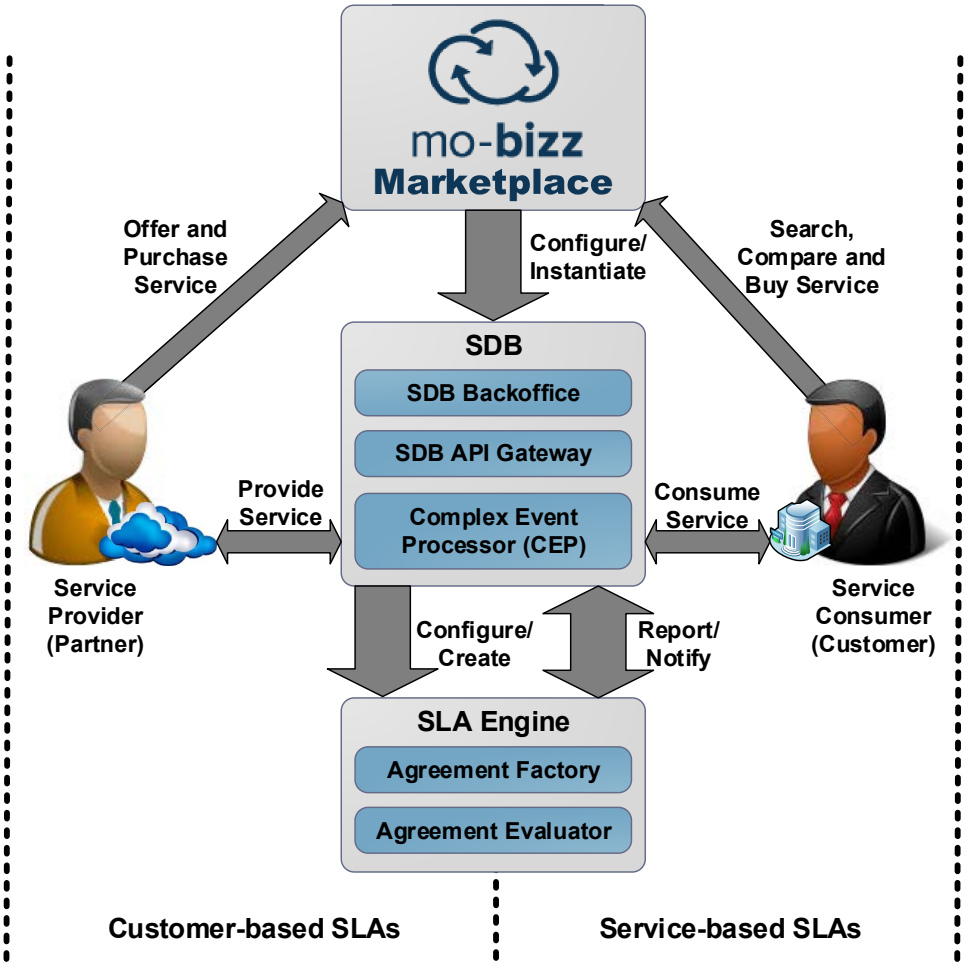

FIGURE 4.6.: SLA Management of MO-BIZZ

The MO-BIZZ platform exposes services that are provided by autonomous organizations, therefore appropriate mechanisms to manage agreements, to define responsibilities, and to advertise SLOs, are required. In order to ensure the fulfillment of QoS guarantees (e.g. availability higher than 99.5%, response time lower than 1.345ms, etc.) SLAs are established for each service that is exposed on the MO-BIZZ marketplace. This SLA serves as a formal contract between the service provider (partner) and the MO-BIZZ platform. Additionally, for each service that is purchased through the Marketplace, SLA metrics are created and are used for an individual evaluation of the established SLA.

The monitoring of a service depends on its nature. In particular, when processing API requests, the SDB Gateway uses internally the SDB Service Monitoring which is a Complex Event Processor (CEP) based on Esper CEP [74] technology to aggregate, count and query event streams resulting from API consumption. Each time an agreement is instantiated metric thresholds, alarms and notifications are configured automatically and are based on the defined SLOs in an agreement offer, which corresponds to the selected KPIs in an agreement template. This enables to process metrics and to create specific SLA events.

In addition to the internal service monitoring of traffic through the SDB Gateway, there is a Service Management Interface (SMI) [75] that has to be supported by each product offer published on the Marketplace. This SMI is especially used for monitoring external cloud services like infrastructure and application services. These external cloud services are directly related to the Infrastructure and Application SLA templates. In general, the SMI has two purposes: It is used for monitoring in order to collect health and performance measurements from instantiated services and secondly to manage the service life cycle of third-party services (e.g. instantiation of a new service or its termination after use).

The SLA Engine that provides the SLA capabilities in MO-BIZZ is based on the WSAG4J [40] framework. It has been customized and integrated as external components in MO-BIZZ. In contrast to the traditional applicability of WSAG4J, the SLA framework acts as an external entity and exposes an interface at the SDB. This interface consists of standardized operations according to the WS-Agreement and the WS-Agreement Negotiation specification and some additional operations for service registration, agreement management, and monitoring functionalities. Considering the SLA Engine as a single component outside the MO-BIZZ ecosystem, then this SLA Engine corresponds to the broker approach of the previous section. However, as SLA Engine inside MO-BIZZ the protocol extension for administration and configuration is of course designed for this particular solution and therefore much more specific.

After the service provider had specified the service offer, integrated the service in the SDB, and checked the functionality, the service offer can be published in the marketplace. With this action an SLA is instantiated and is mandatory for the provider. In particular, the actual formal instance of the SLA is represented as a resource in the WSAG4J-based SLA Engine. Thus, the SLA Engine is able to evaluate agreements with the help of events and measurements coming

from the SDB. Based on this evaluation, the states of SLAs can be requested by the SDB Backoffice and are displayed transparently to the customer and the provider. In case of an SLA violation, an event is triggered at the SDB that notifies the provider about irregularity behaviors. Furthermore, the service provider and the service consumer are able to access historical data about former SLA events in order to clarify conflicts and to initiate compensation requests.

4.3.3. On-Boarding Process

As depicted in figure 4.6, MO-BIZZ differentiates between two SLA structure levels: service-based SLAs and customer-based SLAs. These structure levels have been already defined in section 2.1 as the following:

- A **customer-based SLA** is specific to a customer and can include several services with different service qualities.
- A **service-based SLA** is specific to a service with a specific service quality and has to be accepted by the customer as it is advertised.

Based on these definitions, SLAs are customer-based between the MO-BIZZ platform and the service provider (partner), and service-based between the MO-BIZZ platform and the service consumer (customer). In order to establish such an SLA chain, SLOs and guarantees need to be contracted during the on-boarding process. This on-boarding process for customers to get a service partner is illustrated in the workflow diagram 4.7. Here, the partners have to accept the partner terms of use as basic partnership contract with MO-BIZZ. Within this contract, the service provider has to supply all the required information (e.g.: company name, address, zip code, VAT number, contact email and contact phone number) in order to become verified and consequently accepted as MO-BIZZ partner. Then the service provider has to describe the service offer (e.g.: name, commercial description, technical documentation, required business model and its costs) and all associated services and APIs. In this step, the service provider has to chose an SLA template for the product offer and to select a set of KPIs that the service aims to provide.

After submitting this service offer, the offer is validated and checked for completeness. If this is the case, the service is registered and configured in the SDB. Additionally, the provider is now able to define SLOs for the service and compensation methods that he is willing to pay if guarantees are not fulfilled. In particular, an agreement offer is created based on the selected template with all the information from the previous steps. This agreement offer defines the target SLOs and guarantees of the service provider and is stored until the service offer goes into the certification phase. Here, the functionality is checked, the availability of the management interface, the performance of the service, and whether identity and access management works as expected, according to the integration guidelines. Furthermore, the measurements of the

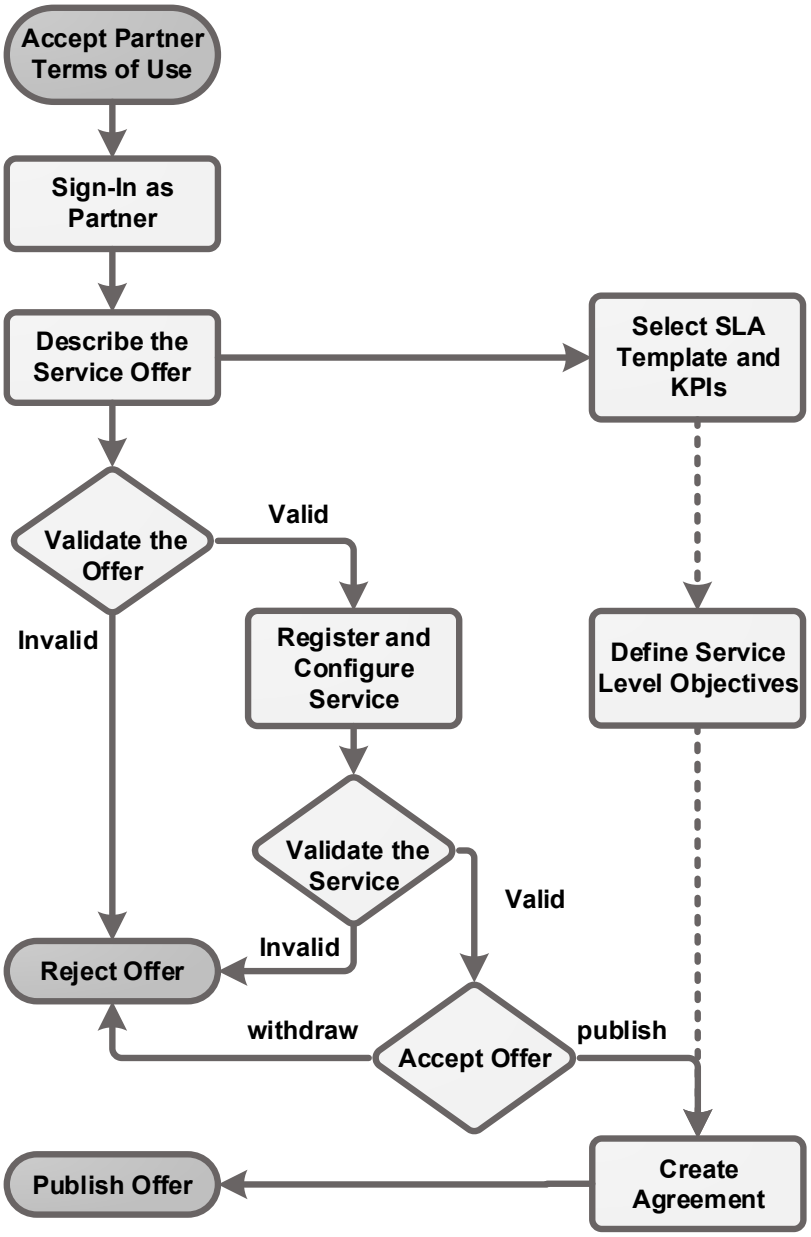

FIGURE 4.7.: MO-BIZZ on-boarding process

performance checks are used to check initially the service quality, which were defined for the agreement offer in the previous step.

If the service is completely configured, ready to offer, and guarantees and SLOs are still fulfilled, the provider is able to withdraw or to publish the product offer. If the partner decides to publish the offer, an SLA is created and the service is published on the marketplace with

the contracted SLA. Based on this SLA, customers can compare this service to other similar or equal services and choose the best fitting service that advertises the most suitable service levels and pricing conditions.

If the customer decides for a service, a service-based SLA (which is a service specific subset of the customer-based SLA) between the MO-BIZZ platform and the customer comes into effect. This SLA is established without any amendments according to the "take-it-or-leave-it" principle.

4.3.4. SLA Inheritance

In the MO-BIZZ ecosystem a service may be consuming other services of third-party partners. Hence, SLA compliance of a single service may rely on SLA compliance of other services. These interdependencies are not visible to the customer, who agreed to an SLA with a matching service quality. The provider that defines the content of an SLA offer has to respect terms and conditions defined in SLAs of services that are being used by this newly composed service. In fact, the new SLA inherits service terms from SLAs defined for its particular third-party services. For example, the availability of an API service can never be higher than the availability of the infrastructure on which the software for this API is deployed. Therefore, advertised SLAs of a single service can be divided into two classes of SLAs which are defined as follows:

Definition 4.1. *A Parent Service Level Agreement denoted as sla_{parent} is established between a customer and a provider, where the provider delivers services that are solely owned and executed by the provider itself.*

Definition 4.2. *A Child Service Level Agreement denoted as sla_{child} is established between a customer and a provider, where the provider delivers services that are based on other services and whose behavior may be influenced by external occurrences.*

Depending on the formal definitions presented in section 2.1, SLAs in MO-BIZZ consist of terms that are the main subjects of an agreement and an agreement offer. These terms can be a set of service descriptions, references to service descriptions, and/or guarantee terms that define SLOs coupled with qualifying conditions that must be met. Thus, terms are defined as follows:

Definition 4.3. *A service term denoted as $term$ with a service identifier i consists of a Service Level Objective denoted as slo, a guaranteed service level threshold denoted as slt, a service property denoted as sp and a qualifying condition denoted as qc.*

$$Term_i := \{slo, slt, qc(slt, sp) | slo, slt, sp \in R, qc \in QCS\} \tag{4.1}$$

The qualifying condition is an operation that is used to evaluate an SLA and to express a guarantee.

$$QCS := \{<, \leq, =, \geq, >\} \tag{4.2}$$

In order to inherit terms of a parent-SLA, the following inheritance operations OPS are defined:

$$OPS := \{\min, \max, \text{sum}, \text{avg}\} \tag{4.3}$$

With the help of these inheritance operations OPS a child-SLA offer can be calculated as follow:

$$\begin{aligned} sla_{child} = \{&(op(slo_1, \ldots, slo_n), op(slt_1, \ldots, slt_n))| \\ &(slo_1, \ldots, slo_n), (slt_1, \ldots, slt_n) \in sla_1 \times \ldots \times sla_n, op \in OPS\} \end{aligned} \tag{4.4}$$

However, this calculation of a proper SLA for a particular product offer is only recommended in MO-BIZZ. In fact, the partner is responsible for the provided service and thus, has to decide which KPIs with associated SLOs and guarantees he is willing to advertise and to accept.

4.4. Federation Approach

Network virtualization is a key technology for deploying next generation networks and applications [76, 77, 78, 79]. Virtual networks can be deployed over a shared network infrastructure, while ensuring isolation of the traffic among concurrent virtual networks. This technology allows companies to specify customized virtual topologies according to their requirements. Software Defined Networking (SDN) complements the network virtualization approach by offering programmability features to the owners of virtual networks [80]. Thus, network functionalities (e.g. routing, firewalls, etc.) which were traditionally implemented by vendors within the hardware, can now be realized in software. This provides a great degree of programmability, since data center providers may develop their own network functions and thus control their virtual networks according to their needs. However, the full range of SDN capabilities is not utilized in today's cloud middlewares. In particular, SDN is used to facilitate future data center networks, but the benefits are often not passed to the paying customer.

Since QoS characteristics can be enforced to the virtual environment of deployed virtual machines [81], a federated provisioning system for autonomous networks fits best for this purpose. Consequently, a provider of virtual networks can apply guarantees in dynamically negotiated and established SLAs to its users. Thus, customers are able to configure the virtual network according to their requirements, which results in an SLA that not only describes the service levels, the objectives, and the prices but can also be used to monitor the fulfillment or violations of advertised guarantees. To achieve this goal a federated *Agreement-Mediator* is introduced that allows for negotiating quality levels for network services that makes use of a provider specific SDN driver to enforce QoS characteristics to the network when all parties have agreed on an SLA. This *Agreement-Mediator* approach is designed not only for managing the networking resources of a single provider, but also for managing the resources within federations of providers, where the end to end user services may involve resources drawn from multiple service providers in a transparent way.

The establishment of a QoS-aware route between two cloud providers involves several independent network providers. Thus an SLA chain has to be discovered, negotiated step by step, and instantiated if all parties come to a common agreement. Here, the customer can define a set of Service Level Requirements (SLR) and would have no additional obligations in this process. In order to facilitate the understanding and to include all information in a single agreement that is presented to the customer, an SLA aggregation for network services is required.

4.4.1. Architecture

The OpenFlow protocol [82] [83] and derivatives enable an innovative mechanism to control extremely large and high performance networks, as often used in cloud service provider facilities [84]. Moreover, OpenFlow provides the opportunity to create a wide range of new applications and the corresponding use cases based on providing the network and the related capacity itself as a service. Configuration and behavior of SDN can be controlled by one entity and changed on demand. The properties of the underlying network and its connected servers can therefore be treated as a resource.

SLAs are established and managed between cloud service provider and consumers and are an instrument to describe a service and its quality. On-demand negotiated and established SLAs in a programmatic fashion for network services do not exist in today's cloud landscape. The architecture described in the following would enhances IaaS middlewares with mechanisms for automated negotiation and creation of SLAs for network services that deliver QoS guarantees (a) between VMs in a single data-center cloud, (b) to external cloud services located anywhere on the Internet, and (c) for interconnections between several heterogeneous cloud environments.

To negotiate, create, and evaluate SLAs in a machine-readable form, a *Service Provisioning System* of a single provider is used. The requirement for interacting with this provider is that the customer already has an established business relationship with the provider. Thus, customers are able to define their requirement and to request services dynamically from trusted providers, which are delivered by an independent service provider. The approach presented in this subsection demonstrates new capabilities based on a shared virtual Ethernet circuit overlay. The basic idea is to enable customers to query and book available network routes between local and external hosts including guaranteed network characteristics. To achieve this goal, a layered architecture has been designed that is depicted in figure 4.8.

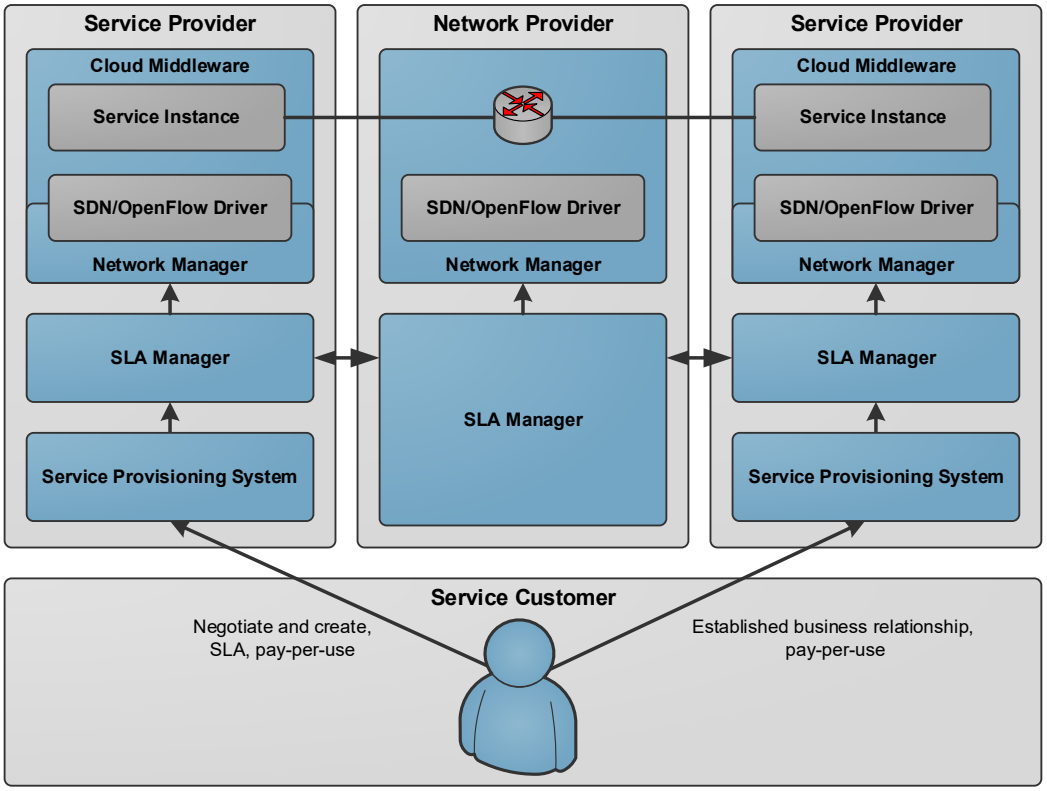

FIGURE 4.8.: Architecture of a federated *Agreement-Mediator*

The middleware software stack of this federated approach is accessible through the *Service Provisioning System*. It can be in the shape of a graphical user interface that allows customers the access in a user-friendly way or a web services-based interface that enables to specify network requirements, configure network topologies and to negotiate SLOs in order to create an SLA and establish an appropriate network environment. Furthermore, this component also provides observation functionalities for detecting SLA violations which occur when network resources have an inadequate behavior and guarantees are not fulfilled. The *Service Provisioning System*, however, is not managing any network resources directly, it just manages aspects regarding the establishment of network services based on SLAs. All properties related to network re-

sources are delegated to the *SLA Manager*, which collects information about the underlying network from the *Network Manager*. Based on this information the *SLA Manager* provides functionalities to negotiate and create aggregated SLAs and then appropriately establish QoS-based network environments in a single data-center and to external services. Furthermore, this *SLA Manager* enables federated networking between several autonomous clouds with heterogeneous infrastructures.

The key components to manage the local and global available network capabilities and its particular QoS allocation is the *Network Manager*. This component builds the backbone of the entire architecture. This architecture addresses mechanisms based on the centralized control plan in SDN substrates and the opportunity to enforce QoS in data-centre as well as comprehensive SDN networks. For pure SDN substrates this architecture considers all opportunities for QoS enforcements based on OpenFlow capabilities (e.g. meter tables) of the latest OpenFlow standards like OF 1.3 and beyond [83].

In contrast to a *Network Manager* installation in a cloud environment, the *Network Manager* in a network provider's infrastructure is an abstraction for the particular external carrier or Internet Exchange Point (IXP) related network and directly exposes the relevant QoS capabilities as external service through the provider's *SLA Manager*. Besides exposing SLA-based networking capabilities to the outside world, the *SLA Manager* checks routes by discovering external *SLA Managers*. To provide a reasonable solution, this approach also has to consider solutions based on direct circuits for cloud data-centres through provider networks or directly connected IXPs.

4.4.2. SLA Aggregation

From a business perspective it is essential that specific services are provided according to predefined service levels (e.g. compliance with minimum bandwidth or service availability). Therefore, this architecture gives providers the opportunity to negotiate SLAs among each other in order to provide a single SLA to the customer. This single SLA contains all expectations and obligations for this particular service provisioning.

Each cloud provider has a relationship and basic contracts with the network provider who is providing the network connectivity to the next IXP and the networks in between. This relationship which is known by both sides is based on a fundamental contract. This contract is for instance also formulating network resources that can be requested by the customer through a provider's *Service Provisioning System*. With this information the *SLA Manager* can be configured in order to provide federated networking.

In the simplest case of a single cloud infrastructure, the *SLA Manager* is waiting for requests from customers and negotiates the conditions for the local network environment. The *Network*

Manager again provides information about the local network utilization and available capacities to the *SLA Manager*. If both the customer and the provider agree to the negotiated conditions, an SLA is created that serves as a formal contract. Afterwards, the network connection is established between the VMs according to the conditions of the agreement and is monitored in order to detect violations. In this scenario, an SLA in a single cloud infrastructure is established between the cloud provider and the customer.

In the complex case of federated networks, an SLA needs to be established for a network service that combines the local network capabilities provided by a local *Network Manager* with network services of an external network provider. In particular, the SLA compliance of a single service may rely on the SLA compliance of other services. Therefore, delivered SLAs of a single provider can be divided into two classes of SLAs which are defined as follows:

Definition 4.4. *A Local Service Level Agreement denoted as sla_{local} is established between a customer and a provider, where the provider delivers services that are solely under the control and only owned by the provider itself.*

Definition 4.5. *An Extended Service Level Agreement denoted as $sla_{extended}$ is established between a customer and a provider, where the provider delivers services that are partially under the control of the provider and depending on third-party services of an external provider.*

Interdependencies of extended-SLAs are not visible to the customer, who negotiated an SLA with its service provider only. Hence, the customer defines the SLRs for a network environment between several VMs. Depending on the entry point, which is in fact the location of the *Service Provisioning System* and its corresponding *SLA Manager*, the provider has to discover all possible routes between specified endpoints. For example, if a customer requests a route between two VMs in different cloud infrastructures, the customer can inquire an offer from both cloud providers or from an external network provider. Thus, the requested provider's *SLA Manager* is responsible for creating a service offer. This *SLA Manager* has knowledge about all *SLA Managers* that are directly connected and responsible for the network connected to the provider's infrastructure.

Figure 4.9 illustrates a situation where the left-hand side provider's data-center is connected with two network providers. Here, two approaches can be applied to discover appropriated routes: First, an directed acyclic graph (DAG) is build consisting of all *SLA Managers* that can be used to establish the requested route. Secondary, the *SLA Manager* requests both federated *SLA Manager* that in turn requests recursively route information from further federated *SLA Managers*. In the first case, the entry point or root *SLA Manager* is responsible for negotiating route characteristics, prices, and all other conditions for agreement offers for each path through

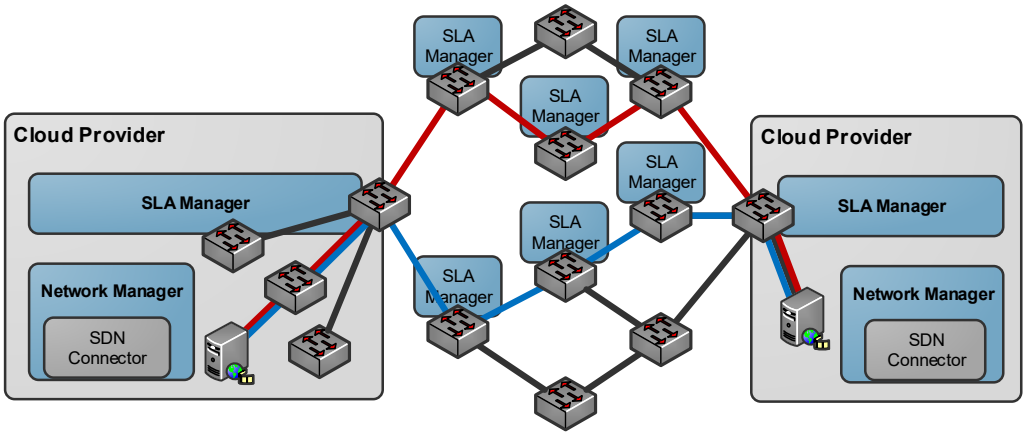

FIGURE 4.9.: Example cloud federation network overlay

the DAG and with each provider of a network on these routes. Moreover, after finding an appropriated route with agreed quality characteristics, an SLA needs to be instantiated and the promised route has to be established. Thus, the *SLA Manager* is also responsible in case of SLA violations and therefore has to monitor the route, to pay penalties and should try to switch to alternative routes if the promised guarantees of an SLA cannot be fulfilled. This is very difficult, because monitoring of external networks and above is impossible in terms of detecting the responsible party on the route. Therefore, the second approach is better and easier for discovering routes over third party provider networks. Here, the *SLA Manager* evaluates a particular SLA with the help of measurements provided by the underlying *Network Manager*. In general, only network providers which have an *SLA Manager* instance up and running can be used to discover a route to an external VM. Furthermore, also the cloud provider on the right hand side has to support this approach in order to establish the last part of that chain between the network edge and the target VM.

If all requirements are met and a set of routes were discovered, the SLA negotiation process can be instantiated. Therefore, the requested *SLA Manager* aggregates its local-SLA offers with the external-SLA offers of both federated network providers. These external-SLA offers are delivered from independent providers, but may use similar or identical network pathes delivered by the same ISPs as illustrated in figure 4.10.

Terms of an SLA are already defined in the previous section 4.3.4. In order to aggregate such terms of a local-SLA offer with an extended-SLA offer, service descriptions or references of third-party provider's SLAs are not relevant for the aggregation in this approach. On the other hand, the number of SLA aggregations is of high importance. This number of aggregations that need to be performed to aggregate all local-SLA offers to a single extended-SLA offer is

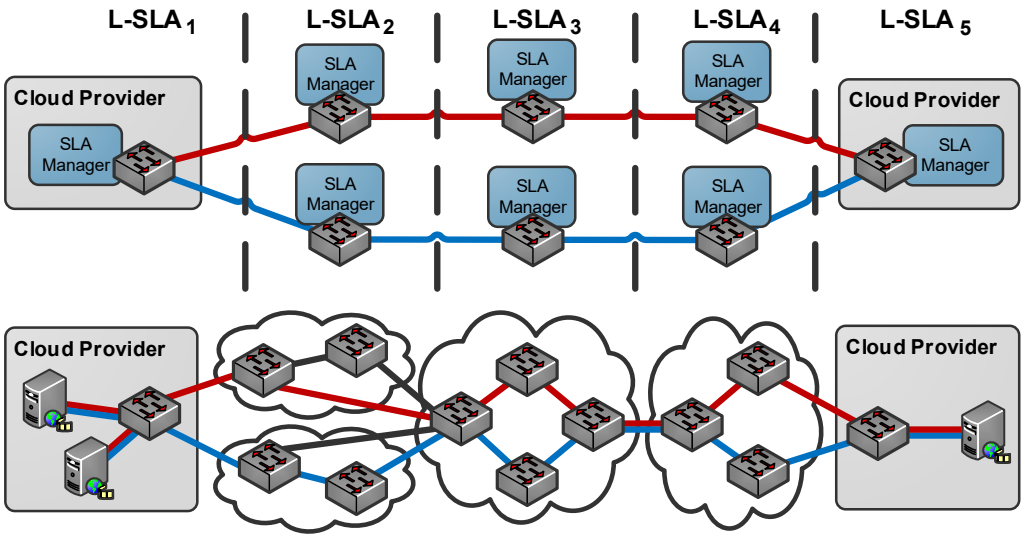

FIGURE 4.10.: Example network SLA aggregation chain

equal to the number of hops of each route. Depending on the kind of service, the aggregation operation has to be applied statically and allows to aggregate all agreement offers recursively by requesting all federated *SLA Managers*. Thus, the aggregation function for each aggregation step is defined as follows:

$$\begin{aligned} sla_{extended} = \{(op(slo_{local}, slo_{external}), op(slt_{local}, slt_{external}))| \\ (slo_{local}, slo_{external}), (slt_{local}, slt_{external}) \\ \in sla_{local} \times sla_{external}, op \in OPS\} \end{aligned} \tag{4.5}$$

Each agreement offer has an *ExpirationTime* element specified in the *Context* element of each local-SLA. This expiration time is an absolute date at which this agreement offer is no longer valid and the resources are no longer reserved. Thus, the customer gets a specific time frame in which a decision has to be made.

4.4.3. Expected Aggregation Count

In order to forecast an expected aggregation count for this approach, an expectation about the distribution of SLA instantiations is presented in the following. Because of the theoretical nature of this approach, the Monte Carlo Simulation was selected as an ideal candidate. In general, the Monte Carlo Simulation is a stochastic method, which performs many random

experiments where the results build the foundation for decision-making. Thereby, it uses the probability theory in order to solve numerical and analytical consumption problems. The goal is to identify the importance of this approach for ISPs in relation to cloud providers.

As already mentioned in the previous section, the number of network hops is not equivalent to the number of the overlay SLA Manager hops. However, the calculation of SLA Manager hops depends on the network hops that are provided by ISPs. As notable research work, the authors in [85] collected hop-count data with a small program that is based on the *traceroute* utility. This program makes use of the time-to-live (TTL) that is set in the packet header. For the Monte Carlo Simulation a maximum hop-count was set to 255, because the TTL in the packet header is an eight binary digit field.

In another research work [86], the authors measured also the hop-count on the Internet but additionally calculated the mean and variance for the total sample. Here, the result is an average hop-count of 16.51 with the variance of 14.96. Based on these results the normal distribution can be applied with these values.

While both measurements present the number of network hops, this approach has just to respect the number of overlay hops. Therefore, the number of network hops resulting from the normal distribution are divided by a random number between one and five, because the average number of hops inside the infrastructure of a single provider is in most cases three [86].

However, the number of hops depends on the position and the target position, therefore an appropriated probability distribution also has to respect not only the geographical position but also the intention of the customer. This intention is mapped in this simulation via a lower bound mapping. In particular, all negative values of the Gaussian distribution are set to zero, thus representing the intention to use the QoS-based networking capabilities just for virtual machines inside a single data center.

The zero for number of aggregations means that no aggregation is performed and just a local-SLA is created for network services of a single provider. Figure 4.11 illustrates the results of 1000 runs. Here, it is very well cognizable that most of the customers would probably use this approach for establishing QoS characteristics in networks of a single cloud environment. Another peak is located at 4.8 aggregations that display the average overlay hop-count of the *SLA Managers* for connections between two data-centers.

4.5. Related Approaches

4CaaSt is a project funded by the European Commission [87]. It aims to develop a PaaS cloud platform that enables the creation of a business ecosystem like MO-BIZZ. Here, applications from different providers can be customized, API services composed, and services traded as

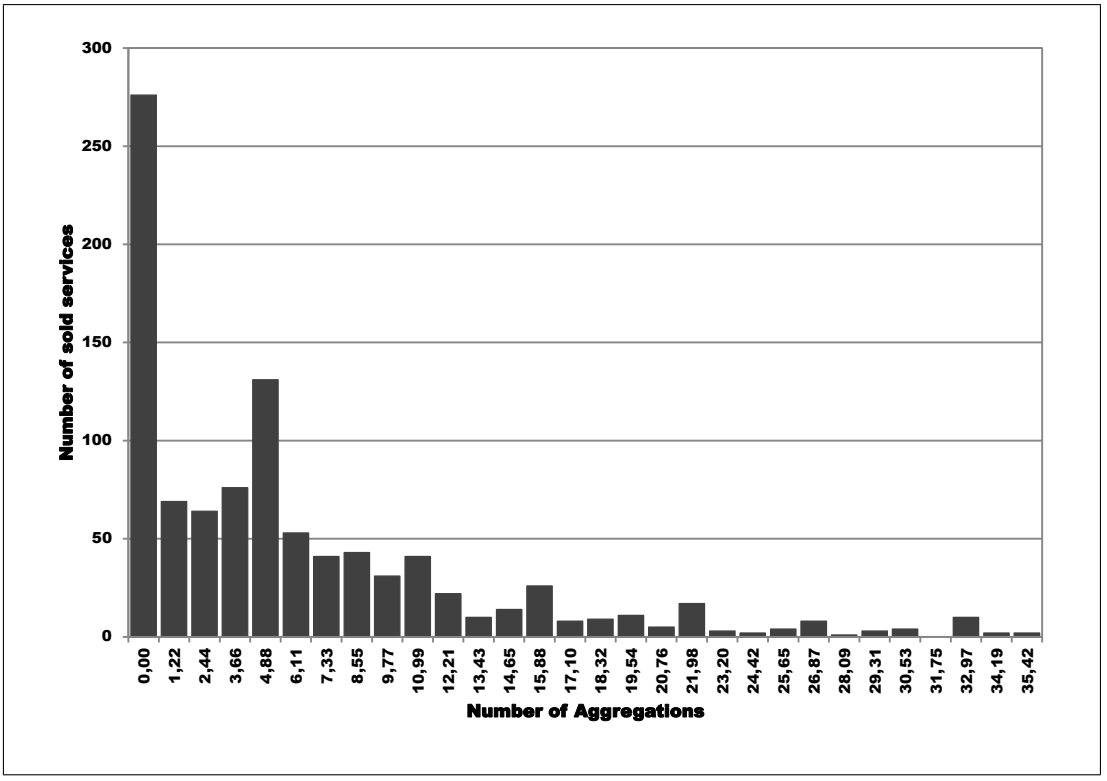

FIGURE 4.11.: Expected probability distribution of SLA aggregations

bundles. This project has developed an eMarketplace as well, where different providers can trade any type of cloud service [88]. The main focus of this marketplace is on SLA support for defining agreement templates, supporting agreement negotiation regarding business and pricing aspects, and SLA enforcement in terms of elasticity management. However, in contrast to 4CaaSt, MO-BIZZ is ready for market launch. Furthermore, MO-BIZZ is not designed only for PaaS offerings, but supports all kinds of cloud services and the monitoring of them as well.

Cloud4SOA provides an interoperable framework for PaaS developer [89]. In this project a REST-based API for any platform access has been developed that builds the foundation layer for three main functionalities: negotiation of SLAs, enforcement of SLAs, and recovery from SLA violations. Furthermore, this project credits oneself by developing a RESTful implementation of WS-Agreement. Besides the overall good results, the actual achievements in this project are the unified monitoring interface and associated definitions for unified platform-independent metrics. In particular, the consortium identified a set of unified metrics for monitoring which are both: application-level and infrastructure-level metrics. The following metrics and corresponding APIs have been developed: response time, application status, memory usage, and CPU usage. Such functionalities are provided in MO-BIZZ by the SMI standard. While the SMI standard is KPI independent, no unified operations for PaaS monitoring are required in MO-BIZZ.

SLA@SOI is an European research project that aims to deliver a SLA Management Framework for supporting SLOs on multiple layers [90]. A reference architecture has been implemented into a SLA management software framework that enables an open, dynamic, SLA-aware market for European service providers [91]. In particular, this SLA management framework is a solution that enables service providers to negotiate SLAs with customers in an automated fashion. SLA(T) is the service description model developed in SLA@SOI. This model follows a hierarchical approach and is language and technology independent. The outcomes regarding SLA negotiation influenced the WS-Agreement Negotiation specification [29]. Besides SLA@SOI, the work on the WS-Agreement Negotiation specification was also supported by the following European and German projects: CoreGRID [92], SmartLM [93], OPTIMIS [94], DGSI, and SLA4D-Grid [95].

The Cloud-TM project developed a data-centric PaaS layer that allows for reducing development and operational costs [96]. The developments of this project are based on the SLA@SOI framework [97]. Besides Cloud-TM, the CONTRAIL project is based on the SLA@SOI framework as well. In particular, it extends the SLA model of SLA@SOI with an OVF descriptor for specifying virtual machine configurations [98]. While SLA@SOI was designed for interoperability between different clouds, CONTRAIL focuses on cloud federations. The main purpose of the CONTRAIL federated approach is to offer elastic PaaS services over federated IaaS cloud resources [99]. In fact, CONTRAIL implements cloud federations, but its architecture is comparable to the broker approach presented in this thesis.

Another project that was also funded by the European Commission was AssessGrid [100]. This project addressed the risk awareness and consideration in SLA negotiation, self-organizing fault-tolerant actions, and capacity planning in the context of Grid computing [101]. AssessGrid's architecture is similar to the broker architecture in this thesis, but only makes use of the WS-Agreement standard. However, AssessGrid like CoreGrid [92], or SmartLM [93] is not actually comparable to the approaches presented before, because their concepts were designed for the Grid domain.

ETICS developed new network control and management services in order to enforce end-to-end QoS characteristics across network service providers [102] [103]. It follows a federated approach with the same purpose as presented in section 4.4. In ETICS static SLAs are established on the atomic service layer between end-user and network provider as well as between two network providers. In contrast to the static SLAs, dynamic SLAs are established for intra-domain network services offered by each of the network providers. The delivered SLA for a particular end-to-end inter-carrier network service is a result of an aggregation of the static SLA and the dynamic SLAs negotiated for each interconnection [104]. The SLA aggregation of ETICS distinguishes from the SLA aggregation presented in this work only in terms of the hierarchical arrangement in which SLA templates are ordered for composite network services.

The first large project that not only used WS-Agreement, but also the standardized WS-Agreement Negotiation is OPTIMIS. The goal of the OPTIMIS project is to suspend the limitation of static QoS offerings of CSPs [94]. Thus, a toolkit was developed that supports infrastructure providers by defining QoS offering within SLA templates, deployment and operation, monitoring of delivered services, and cost and legal issues. Additionally, a multi-cloud broker service was developed for exposing service offerings, negotiation of SLAs, and establishing federated deployments in multiple cloud environments [12]. In fact, OPTIMIS is based on the WSAG4J framework and is definitely the closest solution to the broker approach. However, the essential difference of OPTIMIS is its target service domain which is limited to IaaS offerings. The OPTIMIS approach also introduces a neutral entity that tries to minimize the overhead for managing SLAs between consumers and providers and to raise the trust by mediation of an external entity.

Besides the presented projects above, many more national and international projects (like CloudScale [105], CumuloNumbo [106], GEYSERS [107], Helix Nebula [108], MCN [109], MODACloud [110], mPlane [111], PrestoPRIME [112], SERSCIS [113], or VISION Cloud [114]) aim to design architectures, to specify protocols, and to develop solutions which try to source out SLA affairs in a neutral entity that is realized according one of the *Agreement-Mediation* approaches presented in this section. Not only projects and industries have such a *Agreement-Mediation* approach in their scope of research, but also many academic scientists are working on this topic in order to find a proper solution [115]. For instance, [116] [117] [118] [119] follow a similar approach as presented in section 4.2, [120] [121] follow a similar approach as presented in 4.3, and [122] [123] follow a similar approach as presented in 4.4.

4.6. Conclusion

The sections before presented the most relevant concepts for a neutral and autonomous *Agreement-Mediator* that addresses issues regarding SLA-management, i.e. inflexible term and price conditions, nontransparent verification of advertised service quality, and inflexible expression of SLAs. Furthermore, the biggest challenge in today's cloud market is the discovery of available cloud services and the comparison of them in order to find a desired service with the best relation between service quality and price. Such an *Agreement-Mediator* could create a situation that potentially increases the competition among providers about who offers the most attractive terms to customers, secondly reduces the overall overhead of SLA-management and allows a better binding between a provider's resource state and its offers. Thus, an *Agreement-Mediator* would increase the effectiveness and the efficiency of today's cloud SLAs and service offerings. Furthermore, such an *Agreement-Mediator* would not only establish trust in terms of monitoring (does the provider actually deliver the service as promised) but also trust in the overall service offering of provider where no business relationship has been existed before.

However, each of the presented *Agreement-Mediator* approaches requires an entity that provides the mediation services. In fact, if the provider of an *Agreement-Mediator* solution has a business relationship with a third-party service provider: Does this situation prove that the mediation provider is still autonomous? Furthermore, this mediator only has the knowledge about the registered services offered by providers which have established a business relationship with the provider of the *Agreement-Mediator*. The fact that different projects work on different solutions for these service discovery, service mediation, and SLA management issues shows that the cloud broker/mediation market goes into the same direction as the development of cloud middlewares years before. What is, however, missing is a global intercloud overlay network that interconnects mediators and broker for cloud services around the world. Hence, such an interconnection of cloud service brokers requires a comprehensive understanding between these heterogeneous solutions, a novel intercloud standard is necessary that specifies architectures, protocols, data schemata, and their behavior.

5. Intercloud SLA Management

Contents

Since the cloud paradigm appeared, a lot of research and implementation effort has been invested in order to federate cloud middleware solutions. The main issues in this context are related to resource management, identity management, portability, service discovery, and interoperability in general. Most solutions focus on federations between a private and a public cloud. Thus companies can run a private cloud in their own data center and make use of additional external cloud resources when all local resources are in use. Load peaks can thus be cushioned in order to ensure high quality levels of provided services within the cloud environment. However, most federations are built for a set of public clouds and implicate a vendor lock-in for a particular federation.

Some years ago, the Intercloud project emerged with the goal to overcome the vendor lock-in by building a global federation network of clouds. The vision of this intercloud network is to facilitate interconnections between cloud providers around the world. These providers should be verified and classified for trust in order to establish a global confidentiality cloud

overlay network. For this reason, the IEEE Cloud Computing Initiative [124], the IEEE Cloud Computing Standards Committee, and the IEEE Standards Association [125] established two work groups (WG) and an associated testbed project:

- the **IEEE P2301** work group develop a *Guide for Cloud Portability and Interoperability Profiles (CPIP)* [126],
- the **IEEE P2302** work group develop a *Standard for Intercloud Interoperability and Federation (SIIF)* [127], and
- the **IEEE Intercloud Testbed Project** [128] is a global lab to prove and improve the Intercloud technology according to the development of the standardization initiatives.

These initiatives have the goal to develop standards and a testbed for a world wide cloud provider overlay network. Such an intercloud network would achieve better QoS, higher reliability, increasing flexibility and scalability, and allows for discovering specific cloud resources with individual prices offered by participating providers at a distributed marketplace. The architecture developed in this intercloud context is based on the Extensible Messaging and Presence Protocol (XMPP) [129] [130] and comprises three fundamental service elements as illustrated in figure 5.1.

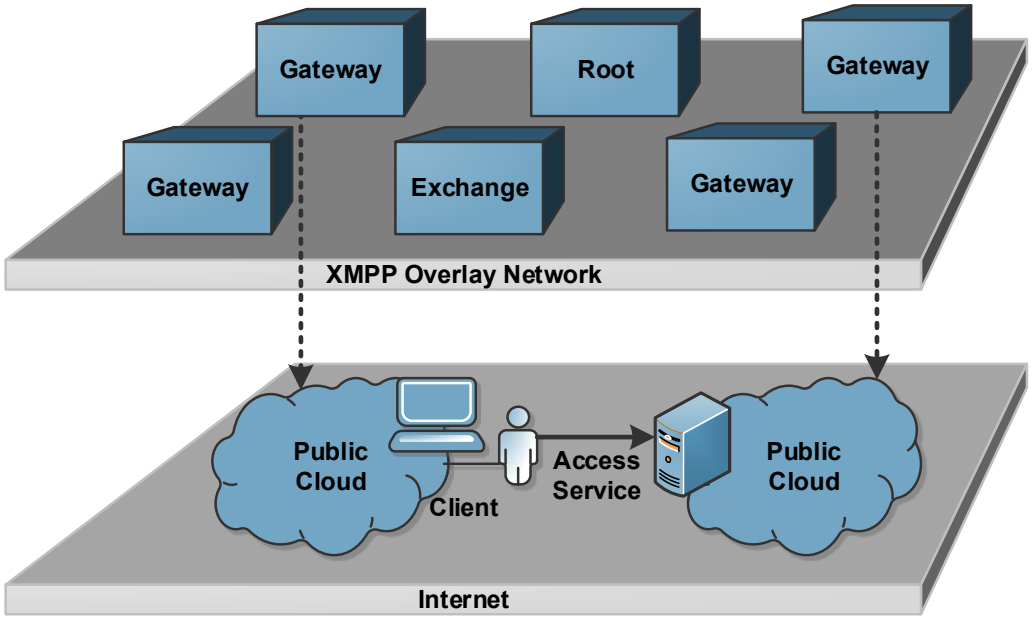

FIGURE 5.1.: Intercloud architecture

The intercloud service elements are governed by autonomous organizations. An *Intercloud Root* element provides "root" services like naming authority, trust authority, directory services, and so forth. It acts as a broker in the intercloud overlay network and hosts a global cloud

resource catalog which can be explored in order to discover desired cloud services. Assuming that such *Intercloud Roots* are distributed around the world which continuously synchronize each other and are permanently updated with current cloud service product offerings advertised by cloud providers, these Root elements provide a global distributed marketplace for cloud services.

An *Intercloud Exchange* provides negotiation and collaboration capabilities among heterogeneous and autonomous cloud environments. Each *Intercloud Exchange* is affiliated with a particular *Intercloud Root* element and hosts second-tier services. Therefore, such *Intercloud Exchange* can be seen as independent notary in the intercloud network. In the approach realized in this thesis, the *Intercloud Exchange* is designed as *Agreement-Mediator* where individual SLAs for cloud service provisions are negotiated, established and evaluated.

The last intercloud element is the *Gateway*. Such an *Intercloud Gateway* represent an interface between a particular cloud and the intercloud network. *Gateways* translate intercloud requests and responses to the individual and customized protocol used by providers internally. Users of a particular cloud provider are able to access the intercloud network as users from the particular cloud provider's domain. When the user found a desired cloud service at another cloud, the access to the service is granted in the traditional way. In particular, the discovery and management of cloud resources are performed through the trusted XMPP overlay network, but the access after provisioning is performed over HTTP, SSH or any other protocol outside the intercloud overlay network.

The establishment of trust, the billing and associated business models as well as the federated identity management of this intercloud initiative are out of scope of this thesis. However, it is considerable to assign these responsibilities to the accountability of each particular cloud provider which participates in the global intercloud network. The rest of this chapter is organized as follows: Section 5.1 explains XMPP fundamentals which are required for the overall understanding of the second part of this thesis. Section 5.2 discusses related work. In particular, this section presents the current P2302 specification and the latest achievements of the intercloud initiatives. Section 5.3 introduces a novel protocol that addresses the requirements of the P2302 to design a service framework that provides similar capabilities as HTTP-based SOAP or REST web services. Section 5.4 presents data models and methods to apply the OCCI Core Model to the Intercloud concept and the newly introduced REST with XMPP protocol. Then the achieved architecture of an autonomous *Agreement-Mediator* that builds the major goal of this work is presented in section 5.5. Last, a set of protocol extensions are defined in section 5.6. They are required in order to realize the presented architecture.

5.1. XMPP

The Extensible Messaging and Presence Protocol (XMPP) [130] was developed by the XMPP Standard Foundation [129]. It is a message-oriented communication protocol that is based on XML and TCP. Even though XMPP is located in the ISO Open Systems Interconnection model on the same layer as HTTP, XMPP also supports extensions to provide HTTP transport over XMPP [131]. In particular, while the XMPP core specification defines protocol methods such as setup and teardown of XML streams, channel encryption, authentication, error handling, communication primitives for messaging, network availability, and request-response interactions, a set of additional XMPP Extensions (XEP) specify protocol add-ons.

XMPP enables the near-real-time exchange of small pieces of structured data which are called "XML stanzas". XMPP specifies three types of XML stanzas: message, presence, and IQ (short for Info/Query). Message stanzas are used to exchange messages among two entities over the XMPP overlay network. Presence stanzas are used to expose the availability of an entity in the XMPP overlay network and IQ stanzas allow for exchange request and response XML stanza. All these stanza types have a kind of header in common. In particular, besides stanza specific attributes the XML root elements (`<message/ >`, `<presence/ >`, and `<iq/ >`) must also have addressing attributes such as a `from` and a `to` attribute. The value of these attributes has to be either a "bare JID" (e.g. localpart@domainpart) or a "full JID" (e.g. localpart@domainpart/resourcepart). Thus, individual addressing is possible and enables a secure, standardized, and fast exchange of XML stanzas over the network.

Additionally to the two addressing attributes an IQ stanza also has a required `type` and a required `id` attribute. While the `id` attribute has the purpose of associating a response to a request, the `type` attribute identifies the kind of the IQ. Four types are specified: `get`, `set`, `result`, and `error`. The IQ types `get` and `set` are operations of a request IQ stanza and return a response IQ stanza of type `result` in case of a successful request processing. If a failure occurs in the request processing, an IQ stanza of type `error` is returned. Examples of IQ stanza processing are presented in the subsequent sections.

XMPP-based services may be distributed at different locations on the Internet. In order to discover desired services within a provider's domain, the XMPP Standard Foundation has specified an XMPP Extension called XEP-0030: Service Discovery [132]. This protocol extension enables to discover information about other XMPP entities. It defines two kinds of information that can be discovered:

- **disco#info** allows for retrieving information about the identity of an entity, the protocols, and the features that this entity supports
- **disco#item** allows for retrieving items associated with an entity, such as the list of components connected to a XMPP server instance.

This section presents only the required fundamentals of XMPP and important extension in order to understand the following concepts and definitions. For further information about how trust, security, or XML streams are established, it is recommended to read the XMPP core specification [130].

5.2. Related Work

The vision of the intercloud project emerged in 2009, but due to the reason that the issues related to this project are very complex and the standardization process has to respect different cloud services, several cloud concepts, existing protocols, interoperability architectures, provider and customer requirements, and related standardization efforts, the current intercloud protocol development is still in its infancy [133].

The first publications regarding the intercloud project were co-authored by David Bernstein who is the founder and chief architect of the IEEE Intercloud Testbed [7] [134]. In these scientific papers the authors introduced the vision, the basic possible protocols, some use cases, and the challenges that go with it. In [135] David Bernstein and Deepak Vij, who is the P2302 working group chair, introduced the intercloud topology components: *Intercloud Root*, *Intercloud Exchange*, and *Intercloud Gateway*. Furthermore, the authors presented the need for a conceptual description and modeling of information similar to the Semantic Web [136] with the help of the Resource Description Framework (RDF) [137].

In [138] the authors propose an approach for a scalable exchange of descriptions about heterogeneous resources. They define an initial ontology for intercloud resources which may be adopted to the resource catalog that should be provided by an *Intercloud Root*. Their idea is to discover available service offerings with corresponding characteristics through SPARQL queries. Such a SPARQL query is able to deliver product offerings and could allow an automatic comparison of these offerings based on further defined object properties. In order to define ontologies, to import or to enhance existing ontologies, and to allow the use of ontologies for resource life-cycle management the authors of [139] propose the Federated Infrastructure Description and Discovery Language (FIDDLE) for this purpose.

Besides the related work above, the authors of [140], [135], [141], [142], [143], [144], and [145] proposed, adopted, and presented solutions which partially influenced the current version of the P2302 specification or are results already specified in there. Thus, the most relevant work in the context of the IEEE Intercloud initiative is of course the latest P2302 specification [146] itself. This specification defines the responsibilities of the intercloud topology components and the protocol basis (i.e. XMPP) as described before. Here, it also specifies that the supported encryption mechanisms (i.e. Transport Layer Security (TLS) and Simple Authentication and Security Layer (SASL) [130]) are used to restrict the access and the communication

between XMPP server-to-server and client-to-server connections. Based on this, the specification defines that a service framework layer has to be introduced which will be able to support SOAP or REST services analogous to the HTTP-based ones. However, the specification also explicitly says:

> *"... the intrinsically synchronous HTTP protocol is unsuitable for time-consuming operations, like computationally demanding database lookups or calculations, and server timeouts are common obstacles.*
>
> ...
>
> *XMPP based services, on the other hand, are capable of asynchronous communication. This implies that clients do not have to poll repetitively for status, but the service sends the results back to the client upon completion. As an alternative to RESTful or SOAP service interfaces, XMPP based services are ideal for lightweight service scenarios."* [146]

To address the issue, the P2302 specification aims to develop a series of XMPP extensions (XEP) and proposes one extension as a candidate: XEP-0244 IO Data [147]. This extension was prototypically implemented in the XMPP Web Services for Java (XWS4J) framework [148] and is primarily used for long time consuming jobs in bioinformatics [149]. It allows for discovering specific commands that can be executed. These commands have generic input and out types which are in fact XML documents. The schemata for the input and output documents can be discovered as well and retrieved on-demand. This enables to automatically build client stub codes. However, the IO Data extension was designed for a closed system which may fit the requirements in the domain of bioinformatics [149], but has the following drawbacks that keep it from being the foundation XEP in the intercloud project:

- The entity that intends to invoke a remote method has to know on which entity the method is available.
- The entity that intends to invoke a remote method has to know which method is provided and how this method is named.
- The generated client stub code has to be compiled and linked at runtime in order to use the generated client.
- The extension supports "Schema Discovery" but not service/operation discovery.
- No existing cloud computing standard (see section 3.6) can be applied to this protocol extension.

The requirement to support time-consuming operations that are not effected by HTTP time-outs is basically fulfilled by the fundamental characteristic of XMPP. However, most of the tasks and operations for managing cloud computing services are not very time-consuming. For example, time consuming tasks may be virtual machine image uploads or migrations. Not time-consuming operations are the pre-defined virtual machine instantiation, termination, re-configuration and so on. Of course, the processes in background may consume a longer time than the triggering of the process, but this is dissembled in the provider's specific cloud middleware implementation behind a *Intercloud Gateway* anyway.

Beside an ontology-based resource catalog hosted on *Intercloud Root* services that will be governed by organizations such as the Internet Society (ISOC) [150] or Internet Corporation for Assigned Names and Numbers (ICANN) [151] and an appropriated trust model, the P2302 specification also describes an approach for SLA management. Here, the authors of the SLA related part of the specification identified the need for machine-processable SLA terms, their deployment, and their automated evaluation. However, this SLA management approach mixes the purpose of SLAs in general with automated provisioning capabilities of particular cloud services. Furthermore, this initial SLA management approach enforced additional functionalities that a partner has to provide. These requirements were domain-specific and may discourage potential customers and providers to get a partner within this intercloud network:

> *"...focusing on intercloud-specific SLA ... assumes (1) all intercloud services are defined in terms of their APIs, (2) all cloud service implementation and management details are not exposed to the service consumers, and (3) all cloud services support policy-based auto-scaling."* [146]

The initial approach in this specification completely leaves open where and which SLA and monitoring services are provided. Moreover, the specification provides no details about SLA management workflows, life-cycles, interfaces and how to access them as well as how this initial approach was fit into the overall intercloud concept. This thesis solves all these issues and is entirely cloud service and provider's domain-independent.

5.3. REST with XMPP

As presented in section 3.4 and 3.5, REST is an architectural style that aims at simplifying component implementations, reducing the complexity of distributed software elements, improving the performance, and increasing the scalability. In relation to the definition of an RESTful application programming interface (API) the uniform interface constraint is of high importance. It simplifies and decouples the architecture and makes REST components independent. The

constraints for a uniform interface can be reduced to: the identification of resources, the self-descriptive representation of resources, and the self-descriptive manipulation of resources.

REST systems typically communicate over HTTP and are gaining large acceptance due to their growing support and their simplicity for implementation. In this context RESTful web services are a simpler and efficient alternative to SOAP and WSDL-based web services which are specified for the use with XMPP in XEP-0072: SOAP Over XMPP [152] and also a more powerful alternative to XML-RPC [153] which is specified as XMPP extension in XEP-0009: Jabber-RPC [154]. Since all existing cloud computing standards (see section 3.6) are based on REST, an REST-based protocol extension for XMPP is needed. This extension has to address the requirements of P2302 to introduce a service framework that provides similar capabilities as HTTP-based SOAP or REST web services. However, no XMPP extension for REST with XMPP exists until now. Therefore, this section defines how the REST architectural style can be applied to pure XMPP entities. It specifies an XMPP protocol extension for accessing resources and transporting resource metadata and XML-REST encoded requests and responses between two XMPP entities. This protocol extension has the purpose to be the intercloud XEP foundation and thus close the lack of a missing service framework.

The XEP-0332: HTTP over XMPP transport [131] protocol extension allows for designing REST services in the context of XMPP, but requires an implementation of both protocols: XMPP and HTTP. Furthermore, HTTP was selected in the past because of its degree of popularity, but has some drawbacks like the lack of discovery for services. The REST with XMPP extension defined in this section is a powerful protocol for cloud services that has several advantages in contrast to the traditional HTTP-based REST approach:

- services are discoverable and explorable,
- dynamic generation of clients stubs on the fly
- the state-oriented programming paradigm is applied to resources [155], and
- multiple input and output types definitions are possible.

The REST with XMPP protocol makes use of the IQ stanza in order to enable access, to create, to delete, or to modify resources of an XMPP entity. Furthermore, this protocol also allows to define individual actions that can be invoked on a resource. For this purpose this specification defines two XML Schema files: one for exploring the capabilities of a resource and one for transfering representations to a method or performing actions on a resource. In order to achieve these goals, this specification has been designed with the following requirements in mind:

- REST with XMPP should be easy to implement such as it is with REST over HTTP.
- This specification should apply the REST architectural style to XMPP and should eliminate limitations of HTTP.
- Resources should be linkable in terms of static connections as well as link targets used for resource access and modifications.
- The number of parameters of an action and the number of supported media-types of a method should be unbounded.
- The number of operations should be unlimited as in contrast to HTTP's GET, POST, PUT, DELETE methods.
- Methods and actions should be dynamically available for invocations according to the state-oriented programming model [155].

5.3.1. Resource Exploration

In order to explore the capabilities of a resource, the IQ stanza type `get` has to be used. The returned IQ stanza is either of type `error` or `result`. If it is of type `result`, the returned content has to comply with the XMPP Web Application Description Language (XWADL) schema of this specification. The XWADL schema has been designed for providing a machine processable description of a resource. It was inspired by the WADL standard and can be found in the appendix A.1.

An IQ stanza of type `get` returns an IQ stanza of type `result` that describes all methods and actions which the requesting party can perform. The following example in listing 5.1 shows an exploration request for a cloud provider's REST based interface that manages compute services.

```
<iq type='get'
    from='requester@company-b.com/rest-client'
    to='company-a.com/openstack'
    id='rest1'>
  <resource_type xmlns="urn:xmpp:rest-xwadl" path="/compute" />
</iq>
```

LISTING 5.1: Exploration of an interface for managing compute services

In order to explore a resource, only the path to a resource is required. The counter party has to answer such a request with a response that exposes all possible methods and actions which can be performed on the resource located at the specified path. The following example shown in listing 5.2 illustrates a response that exposes all methods for this resource.

```
<iq type="result"
   from="company-a.com/openstack"
   to="requester@company-b.com/rest-client"
   id="rest1">
  <resource_type xmlns="urn:xmpp:rest-xwadl" path="/compute">
    <documentation title="Summary">
      This resource allows for managing compute instances, e.g.
      creating virtual machines.
    </documentation>
    <method type="POST">
      <request mediaType="text/occi">
      ...
      </request>
      <response mediaType="text/uri" />
    </method>
    <method type="GET">
      <response mediaType="text/uri-list" />
    </method>
  </resource_type>
</iq>
```

LISTING 5.2: Result of an exploration for handling compute services

This response exposes two methods that can be performed on the resource located at entity `company-a.com/openstack` at resource path `/compute`. The first method of type `POST` can be used to create virtual machines. This method expects a representation of media-type *xml/occi* and returns a representation of media-type *text/url*. A detailed example of how to access this method is illustrated later in the Resource Access subsection. Based on this overview, the following subsections describe each element of the designed XWADL schema and its assigning documents in detail.

Resource Type

The `resource_type` element forms the root element of an XWADL document and may comprise one or more of the following sub-elements: `documentation`, `grammars`, `action`, and `method`. This *REST with XMPP* specification distinguishes between two kinds of operations: actions and methods. Methods are used to transfer state representations according to predefined media types among two entities. Actions in turn are used to perform operations on a resource in order to retrieve primitive values or to change the state of a resource. In fact, while methods are used to transfer representations, actions are used to invoke operations on an existing resource with primitive data type as parameters or return value similar to XML-RPC. Figure 5.2 depicts the possible elements of an XWADL document schematically.

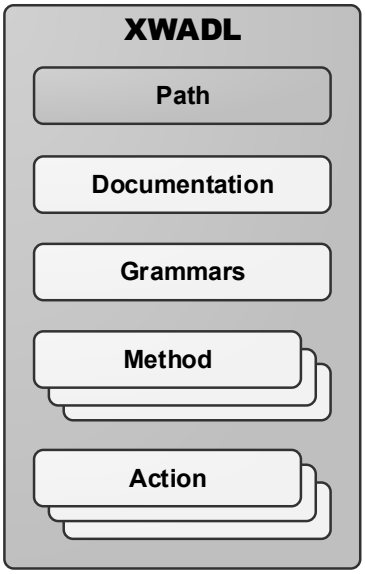

FIGURE 5.2.: Structure of an XWADL document

Documentation

Each XWADL element down to the `request` and the `response` elements can have one child `documentation` element that can be used to append a human-readable explanation of that element. The `documentation` element has a `title` attribute which is a short plain text description of the element being documented. The `documentation` element itself can have string represented content and may contain text, HTML, zero, or more child elements.

Grammars

The `grammars` element acts as a container for definitions of the format of data exchanged during execution of the protocol described by the XWADL document. It should be loosely based on the XML Schema definition and specify data structures used for data exchange. The following listing 5.3 illustrates an example of a grammars element.

```
...
<resource_type xmlns="urn:xmpp:rest-xwadl"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" path="/address-book">
  <grammars>
    <documentation title="Person List"/>
    <xs:element name="PersonList" type="MyStructType"/>
    <xs:complexType name="MyStructType">
      <xs:sequence>
       <xs:element name="Person" type="MyPersonType"
           maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="MyPersonType">
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="age" type="xs:integer"/>
      </xs:sequence>
    </xs:complexType>
  </grammars>
...
```

LISTING 5.3: Example XWADL document with a grammars element

Method

A method element describes a specific operation that can be performed on a resource targeted by the `path` attribute of the `resource_type` element. A method element is a child of a `resource_type` element and has a `type` attribute that identifies the kind of such a method. In order to achieve simplicity for implementation and having uniform interface constraints, methods are of a specific type. The types defined for a method are `GET`, `POST`, `PUT`, and `DELETE` like the CRUD methods of HTTP. This has the purpose to gain a feeling of familiarity for the developer and increases his intention to use. Additionally, each method can have one `request` and one `response` element which can be empty or be used to expose optional templates for representations expected by this method.

Request and Response

The `request` and the `response` elements of a method describe the input and output representation data to be transferred among an entity and a resource of another entity. While a request describes the input to the method, a response describes the output from the method.

Both elements have a required `mediaType` attribute that specifies the media type of the representation expected by the method. A `request` element in contrast to a `response` element can also define a set of representation templates as illustrated in listing 5.4.

```
...
<request mediaType="xml/occi">
  <template>
    <Category xmlns="urn:xmpp:occi-representation">
      <Kind>
        <term>compute</term>
        <schema>http://schema.ogf.org/occi/infrastructure#</schema>
        <title>Compute Resource</title>
        <attribute name="occi.compute.memory">
          <DOUBLE>4.0</DOUBLE>
        </attribute>
        <attribute name="occi.compute.cores">
          <INTEGER>2</INTEGER>
        </attribute>
      </Kind>
    </Category>
  </template>
  <template>
    <Category xmlns="urn:xmpp:occi-representation">
      <Kind>
        <term>compute</term>
        <schema>http://schema.ogf.org/occi/infrastructure#</schema>
        <title>Compute Resource</title>
        <attribute name="occi.compute.memory">
          <DOUBLE>8.0</DOUBLE>
        </attribute>
        <attribute name="occi.compute.cores">
          <INTEGER>4</INTEGER>
        </attribute>
      </Kind>
    </Category>
  </template>
</request>
...
```

LISTING 5.4: Templates for creating a virtual machine

In the example above, the XWADL document of a resource offers two flavor types for the creation of a virtual machine. It is the representation template list expected by the `POST` method shown in listing 5.2. The schema used for the `xml/occi` media type is presented later in section 5.4.

Action

An action element describes a specific action operation that can be performed on a resource targeted by the `path` attribute of the `resource_type` element. An action element is a child of a `resource_type` element and has a `name` attribute that identifies a specific invokable operation of a resource. The number of actions and the amount of parameters of an action is unlimited. This eliminates limitations of HTTP and allows for defining APIs similar to SOAP and XML-RPC. In combination with detailed descriptions of each individual element, this achieves self-descriptive, efficient, and fractional resource capabilities for self-descriptive manipulation of resources as illustrated in listing 5.5.

```
...
<resource_type xmlns="urn:xmpp:rest-xwadl" path="/compute/vm1">
  ...
  <action name="stop">
    <documentation title="Stop this virtual machine"/>
    <parameter name="method" type="STRING">
      <documentation title="The method used for stopping this vm">
        Accepted values are: graceful, acpioff, or poweroff
      </documentation>
    </parameter>
    <result type="BOOLEAN">
      <documentation title="Returns true if the vm has been stopped successfully"/>
    </result>
  </action>
</resource_type>
...
```

LISTING 5.5: Result of an exploration for controlling a virtual machine

Each action can have one `result` and zero or more `parameter` elements which specify the types of expected parameters and the return value. A `parameter` element describes a parameterized value and can be identified by its `name` attribute. Additionally, it must have a required `type` attribute that declares the type of this parameter. The schema defines a static set of possible primitive data types: `STRING`, `INTEGER`, `DOUBLE`, `BOOLEAN`, and `LINK`. These primitive data types are also required to specify the return value in the `result` element as illustrated in listing 5.6. A `parameter` element can also have an optional `default` attribute that exposes a default value which is applied if this parameter is not stated. Therefore, if this attribute is specified, the overall `parameter` element is optional when accessing a resource.

```
...
<resource_type xmlns="urn:xmpp:rest-xwadl" path="/compute/vm1">
  ...
  <action name="start">
    <documentation title="Start this virtual machine"/>
    <result type="BOOLEAN">
      <documentation title="Returns true if the vm has been started successfully"/>
    </result>
  </action>
</resource_type>
...
```

LISTING 5.6: Result of an exploration for controlling a virtual machine

When retrieving an XWADL document for a specific resource, this XWADL document comprises all methods and actions that are available for this particular resource at a specific moment in time. It is like a snapshot of capabilities and the state of this resource. In fact, the actions illustrated in listing 5.5 and 5.6 are never available at the same time and thus can never be found in the same XWADL document. This methodology enables state machine implementations for resources thus eliminating further limitations of HTTP-based REST.

5.3.2. Resource Access

In order to access a resource, the IQ stanza type `set` has to be used. The returned IQ stanza is either of type `error` or `result`. If it is of type `result`, the returned content has to be in compliance with the XML-REST schema that can be found in appendix A.2. The XML-REST schema has been designed for providing an XML-REST encoded payload for accessing a resource. An IQ stanza must contain either one method element or one action element with an individual request and response media-type or a corresponding sequence of parameters. The following example in listing 5.7 illustrates how the `POST` method of the previous example is requested. Here, the client creates a VM which is configured according to a previously advertised flavor.

```
<iq type="set"
    from="requester@company-b.com/rest-client"
    to="company-a.com/openstack"
    id="rest2">
  <resource xmlns="urn:xmpp:xml-rest" path="/compute">
    <method type="POST">
      <request mediaType="xml/occi">
```

```
      <Category xmlns="urn:xmpp:occi-representation">
        <Kind>
          <term>compute</term>
          <schema>http://schema.ogf.org/occi/infrastructure#</schema>
          <title>Compute Resource</title>
          <attribute name="occi.compute.memory">
            <DOUBLE>4.0</DOUBLE>
          </attribute>
          <attribute name="occi.compute.cores">
            <INTEGER>2</INTEGER>
          </attribute>
        </Kind>
      </Category>
    </request>
    <response mediaType="text/uri" />
  </method>
 </resource>
</iq>
```

LISTING 5.7: Access of an interface to create a virtual machnine

In order to make sure that both parties have a common understanding, the requester specifies also the expected responds type which has been exposed during the exploration step. The counter party has to answer such a request with no request element and an extended complement of the response element as illustrated in the example in listing 5.8. The identification of a response to an associated request is accomplished by the `id` of the IQ stanza.

```
<iq type="result"
   from="company-a.com/openstack"
   to="requester@company-b.com/rest-client"
   id="rest2">
 <resource xmlns="urn:xmpp:xml-rest" path="/compute">
   <method type="POST">
     <response mediaType="text/uri">
       <representation>
         xmpp://company-a.com/openstack#/compute/vm1
       </representation>
     </response>
   </method>
 </resource>
</iq>
```

LISTING 5.8: Result of a state transfer to create a virtual machnine

The `resource` element forms the root of an XML-REST document and must comprise only a single `method` or `action` as sub-element. In contrast to the XWADL description, no further documentation or grammars are allowed in order to keep the number of bytes as low as possible. In order to specify the resource to access, the `path` attribute is required identical to XWADL.

The `method` element can have one request element and/or one response element. Additionally, the `type` attribute is required in order to identify the operation that has to be performed on the resource. The `request` and the `response` elements of a method are in the XML-REST schema similar to the definition in XWADL. Both have a `mediaType` attribute in order to define the input and output data of the method. The `request` attribute in XML-REST has no templates in contrast to XWADL. Here only XML with namespace definitions are allowed or the predefined `representation` element for plain text can be used as child element.

Actions are encoded appropriated. An `action` element can have one `result` element and one or more `parameter` elements. In contrast to XWADL, both have no type attributes. The type is encoded as child element with an assigned XML type definition. Thus types are able to verify by validating an XML-REST document with its appropriated XML-REST schema.

A link targets at a single location locally or remotely within an XMPP network overlay. A link is expressed according RFC 2396 [156] that specifies the syntactic constraints of an URI string, e.g. `xmpp://company-a.com/openstack#/compute/vm1`. Here, the authority and path elements constitute the JID and the fragment element is the *Path* of the resource at that entity identified by its JID. This enables a multi-dimensional resource placement. The examples listed in table 5.1 show how different resources can be placed within a single server entity. Each of these links points to another resource.

TABLE 5.1.: Multi-Dimensional Resource Placement

JID	**Path**
company-a.com	/
company-a.com/resource	/
responder@company-a.com	/
responder@company-a.com/resource	/
company-a.com	/resource
company-a.com/resource	/resource
responder@company-a.com	/resource
responder@company-a.com/resource	/resource

Service Discovery

If an entity supports the REST with XMPP protocol, it should advertise that fact in response to XEP-0030: Service Discovery [132] information (*diso#info*) requests by returning an identity of *automation/rest* and the features `urn:xmpp:rest:xwadl` and `urn:xmpp:rest:xml`. The listing 5.9 illustrates the corresponding response.

```
<iq type='result'
    to='requester@company-b.com/rest-client'
    from='responder@company-a.com/rest-server'
    id='disco1'>
  <query xmlns='http://jabber.org/protocol/disco#info'>
    <identity category='automation' type='rest'/>
    <feature var='urn:xmpp:rest-xwadl'/>
    <feature var='urn:xmpp:xml-rest'/>
  </query>
</iq>
```

LISTING 5.9: A disco#info query for REST with XMPP

5.3.3. Implementation Concept

The implementation concept for REST with XMPP provides a portable API for developing, exposing and accessing resources designed and implemented in compliance with the specification defined in this section. The API of this implementation concept is designed similar to the JAX-RS API that is part of the Java Enterprise Edition (JavaEE). This gives the developer the feeling of familiarity and rises the overall acceptance of this concept.

There are three ways to develop XMPP applications in order to make them available within an XMPP overlay network:

- **User Service:** A service provided by a regular user that is logged in. This service starts after log in and is only available until the user is present.
- **XMPP Server Plug-in:** An XMPP server specific plug-in is part of the server itself. Such a plug-in is mostly a package (like a jar file) that has to be linked to the server's plug-in architecture. The disadvantage of this approach is that this plug-in is XMPP server specific and may not be portable across to other servers.
- **XMPP Component:** An XMPP component is a stand-alone server instance that provides a service within a subdomain of the XMPP server to which this component is connected. The protocol between the XMPP server and the external component is standardized in

XEP-0114: Jabber Component Protocol [157] and makes such components implementation XMPP server independent and thus portable across to other servers.

Resources are deployed inside a *Resource Container* in order to be available for exploration and access. Therefore, the implementation concept makes use of the container pattern, which is also applied to JavaEE applications. This pattern has several advantages: resources can be added and removed from the container during run-time, developers of resources only have to care about business logic and functionality inside the resource. All other required interfaces to other resources and container functionalities are provided by the container. The *Resource Container* itself in turn is part of an *XMPP External Component* in order to make this implementation server independent. However, the *Resource Container* design is also applicable to all other XMPP application deployments as well. Additionally, a *Marshaller* that implements fundamental functionalities for service discovery and processing of IQ stanza complements the architecture of this implementation concept that is depicted in figure 5.3.

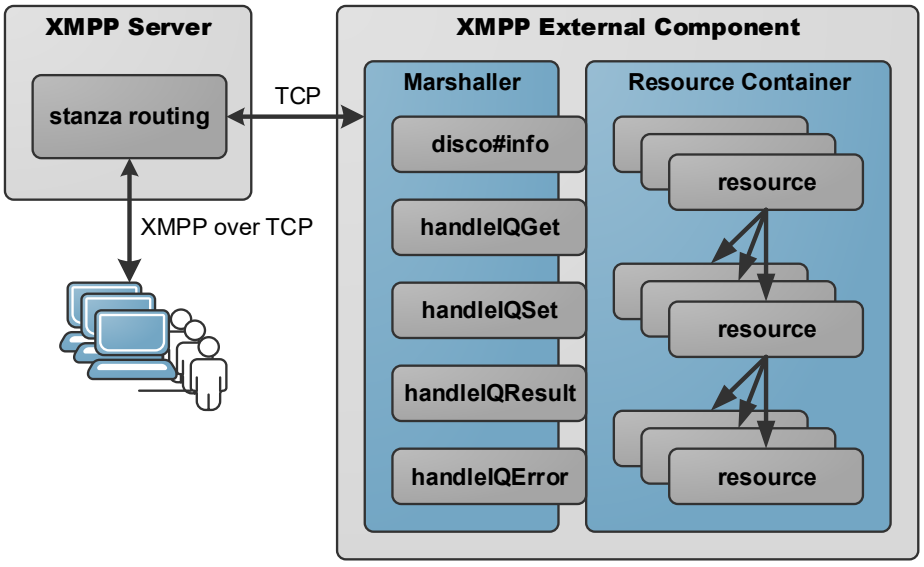

FIGURE 5.3.: Architecture of the implementation concept

The resources in the *Resource Container* are stored in a tree data structure. Each node of this tree in turn has a thread-safe HashMap that stores its child nodes as values and the sub-path of each node as key. Thus, collisions are prevented and this data structure provides an averaged time cost of $\mathcal{O}(1)$ for the basic operations. Also taking the tree structure into account, the complexity for finding resources is $\mathcal{O}(p)$ while p is the number of path elements.

A resource in this implementation concept has to inherit from *ResourceInstance* which implements functionalities of a tree node. These functionalities comprise methods for requesting its path, sub-path, and parent node and adding, removing, and getting child resources. These

functionalities are used by the container for instantiation, exploration, and resource methods invocation. There are two ways for deploying resources within the container: with a static sub-path or a dynamically generated UUID as sub-path. For this purpose two annotations are defined which are applicable to resources classes as illustrated in listing 5.10.

```
@Path("/compute")
public class ComputeManager extends ResourceInstance {
    ...
}

@PathID
public class VirtualMachine extends ResourceInstance {
    ...
}
```

LISTING 5.10: Resource class annotations

Classes annotated with `@Path` specify a static sub-path to which this resource is deployed. In contrast, classes annotated with `@PathID` are deployed under a sub-path consisting of a generated UUID. JAX-RS allows to specify path annotations also for methods and several path elements per annotation. The implementation concept applied here follows the approach that each resource is present only under a specific path. The reason for this is that these resources can have XMPP over REST methods and actions implemented for different representations or type sets. All these definitions belong to only a single resource and are separated in that way. Furthermore, a state machine can be applied to each resource instance which could affect the visibility and behavior of methods if they would have a similar path annotation as in JAX-RS.

XMPP resource methods are defined with the `@XmppMethod` annotation. The kind of this method is passed as parameter like illustrated in figure 5.11. Additionally, the input and output of this method have to be defined by the `@Consumes` and the `@Produces` annotation. Both annotations require definitions of which media types is consumed or produced by this method. Optionally a serializer can be defined that has to implement a default constructor and is used to serialize the representation passed to or from the method. A serializer allows for defining representation templates that are passed into XWADL documents and to provide additional functionalities by objects assigned to the method by the *Marshaller*.

```
@Path("/compute")
public class ComputeManager extends ResourceInstance {

   @XmppMethod(XmppMethod.POST)
   @Consumes(value = OcciXml.MEDIA_TYPE, serializer =
      FlavorMixin.class)
   @Produces(value = UriText.MEDIA_TYPE, serializer = UriText.class)
   public UriText createVM(FlavorMixin flavor) {
      VirtualMachine vm = new VirtualMachine(flavor);
      String path = this.addResource(vm);
      return new UriText(path);
   }
}
```

LISTING 5.11: Resource method annotations

Equivalent to XMPP resource methods, also annotations for XMPP resource actions are developed. An action has to be annotated with `@XmppAction` and an assigned action name. While the action name is obligated, the documentation can be assigned optionally as it is the same case for `@XmppMethod`. The primitive data types for the result element and the parameters are identified automatically by the *Marshaller*. However, only the following Java data types are permitted in this implementation: *java.lang.String*, *java.lang.Integer*, *java.lang.Double*, *java.lang.Boolean*, and *java.net.URI*. Moreover, for each parameter an unique name has to be defined as illustrated in listing 5.12. This allows for having different names among the REST with XMPP representation and its implementation.

```
@PathID
public class VirtualMachine extends ResourceInstance {

   @XmppAction(value = "stop", documentation = "Stop this virtual
      machine")
   @Result(documentation = "Returns true if the vm has been stopped
      successfully")
   public Boolean stop( @Parameter(value = method, documentation =
      "The method used for stopping this vm") String meth) {
      ...
   }
}
```

LISTING 5.12: Resource action annotations

The state-oriented programming model can be applied to a resource instance in two ways: Either the resource instance has to inherit from the `StateOrientedResourceInstance`

class or via an annotation. While the inherited variant allows to retrieve state information on-demand from a remote instance (e.g. a cloud middleware) the annotated variant defines its state in a field attribute. Here, the field has to be annotated with `@State` as depicted in listing 5.13. In order to define which methods are available at which state, the `@Condition` annotation has to be declared for a method or action. If a method is not annotated with `@Condition` this operation is visible and accessible at any time.

```
public class VirtualMachine extends ResourceInstance {

   public enum state { Running, Suspended }

   @State
   State state;

   @Condition("Running")
   public Boolean stop(String method) { ... }

   @Condition("Suspended")
   public Boolean start() { ... }
}
```

LISTING 5.13: State-oriented programming annotations

The *Marshaller* is the key component in this implementation concept. It translates incoming and outgoing IQ stanza into XWADL or REST-XML documents. This is only possible via well annotated methods as presented before.

5.3.4. Performance Evaluation

The specified REST with XMPP extension is a promising approach as alternative to HTTP-based RESTful web services. Therefore, this section compares the performance between these two approaches. For this purpose the test workflow depicted in figure 5.4 has been designed and is performed for this comparison. This workflow comprises different methods which are `GET`, `POST`, and `DELETE`. The workflow depicted in figure 5.4 illustrates the HTTP test case on the left hand side in the graphic and the REST with XMPP calls on the right hand side. The flavors that are available for creating a virtual machine are retrieved in the first step of the test workflow. The first flavor of these possible virtual machine configurations is selected and transferred in the second step. Here a virtual machine resource is created that serves the virtual machine representation. In fact, no actual virtual machine in the backend is created, because that would mean an ambiguous overhead which would distort the measurements. After the virtual machine resource has been created, the URI to the new virtual machine is returned. This URI is then used in the third step, where the virtual machine is destroyed.

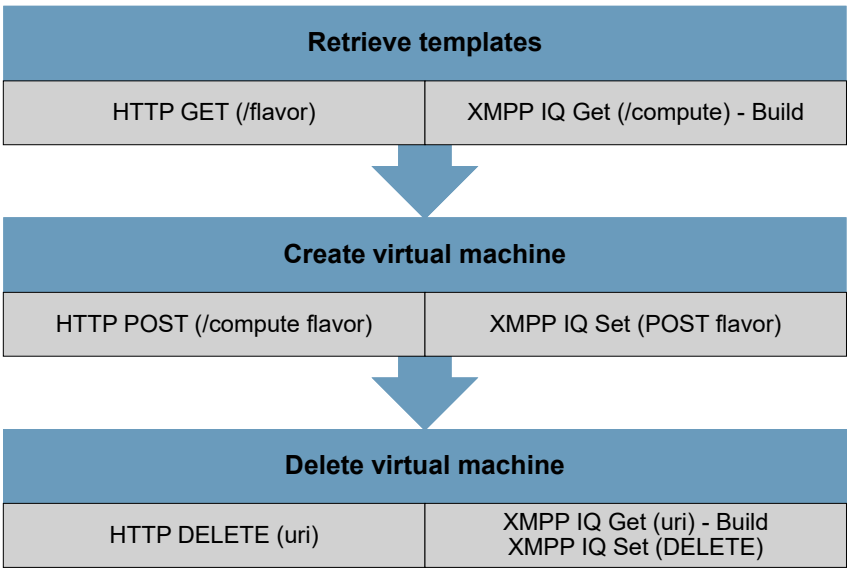

FIGURE 5.4.: Workflow of the performance test

The load tests measured 50 test runs for each test step of the workflow with both HTTP and REST with XMPP code bases. After each test run of the overall workflow, the number of resources is increased in order to measure the transversal scalability as a function of the amount of available resources.

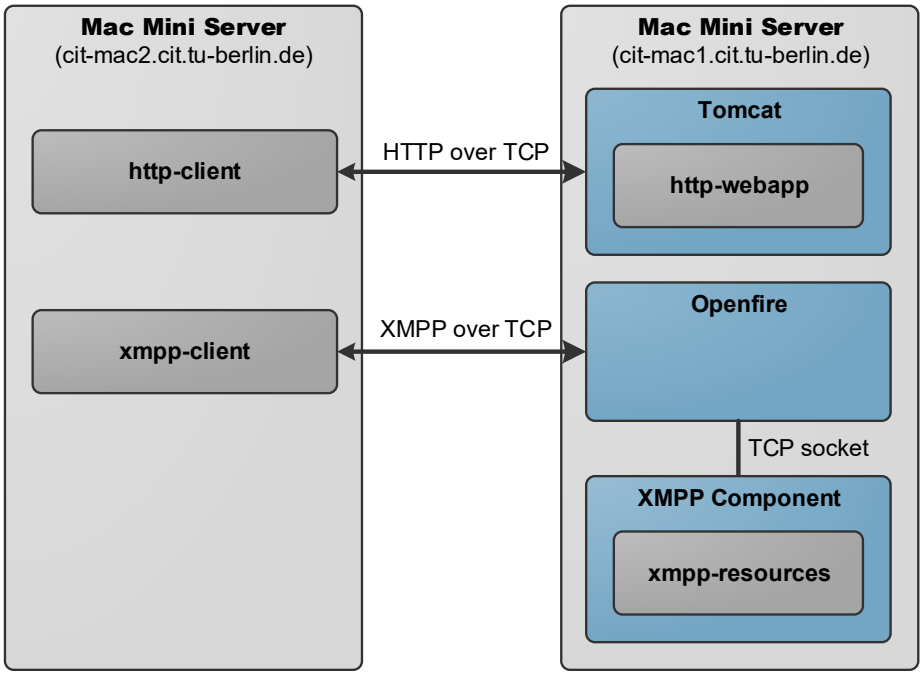

FIGURE 5.5.: Experimental setup for the performance test

The performance tests used two Mac Mini servers as shown in figure 5.5. Each server is equipped with one Intel Core 2 Duo 2.66 GHz CPU and 4 GB DDR3 SDRAM. The nodes are connected via regular Gigabit Ethernet links and run Mac OS X Yosemite version 10.10.3 (Darwin kernel version 14.3.0). Both nodes run Java version 1.8.0_40 (Oracle Java SE Development Kit 8u40). The HTTP web application is deployed as war file in an Apache Tomcat installation in version 8.0.23. The XMPP server used for the performance tests is an Openfire in version 3.10.0. Both are installed on the same physical host. Additionally, the *XMPP External Component* that hosts the *Resource Container* is executed on the same host as well. This component is connected to the Openfire server via a local TCP socket.

Figures 5.6, 5.7, and 5.8 show response times of all test cases. It is important to note that the scale of each diagram's response time axis is adapted to fit the measured values.

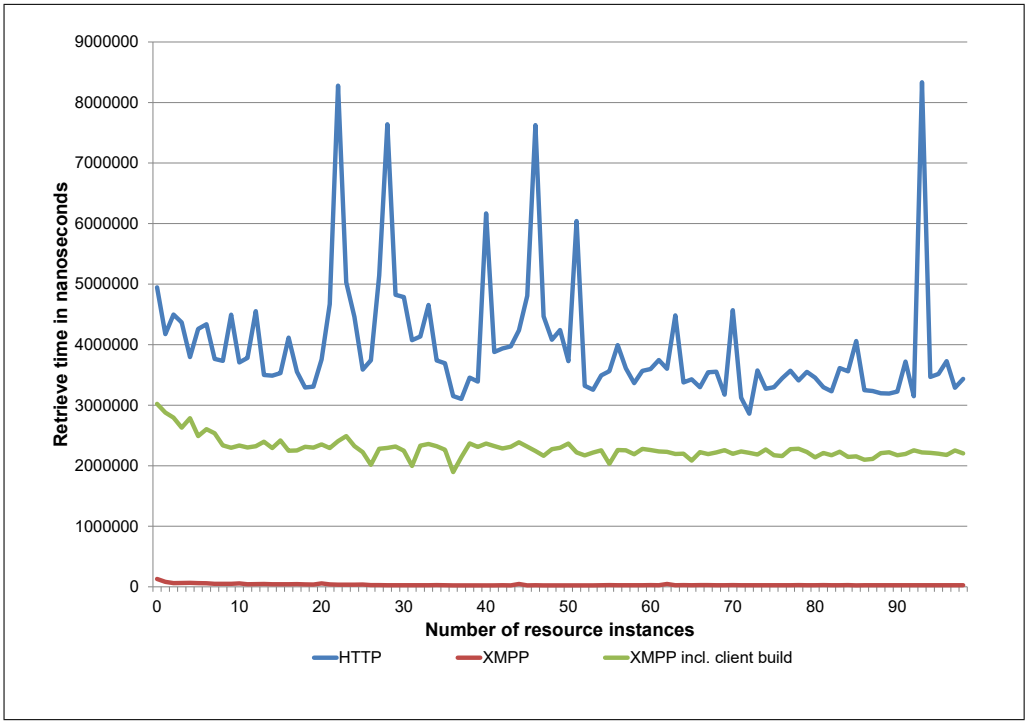

FIGURE 5.6.: Performance of retrieving a flavor template

Figure 5.6 illustrates the performance for the first step of the test workflow. Here the REST with XMPP approach performs better in any case. While for the HTTP test case only one curve (blue) is depicted, for the XMPP test case two curves (red and green) are displayed. The green curve illustrates the measured response time for retrieving the XWADL document that already includes all acceptable templates for the second step of the test workflow. The red curve shows the retrieve time for getting the flavors from the client that has been build dynamically based on the XWADL document. Because of the fact that the pure retrieve of flavors from a local client instance is executed in the same JVM and does not require any additional network transmission, the retrieve time tends to zero.

Figure 5.7 illustrates the performance for creating a virtual machine resource. Here in both code bases, the REST with XMPP and the HTTP, a `POST` method with the same complexity is invoked. This equivalent method invocations clearly show the differences in achievement levels.

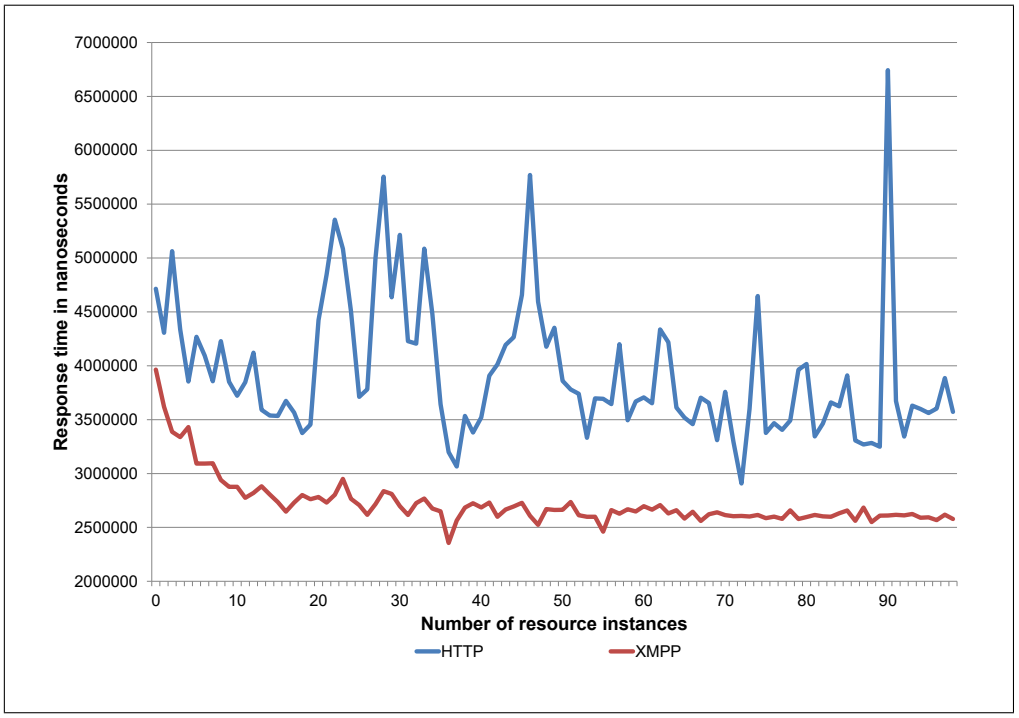

FIGURE 5.7.: Performance of creating a compute resource

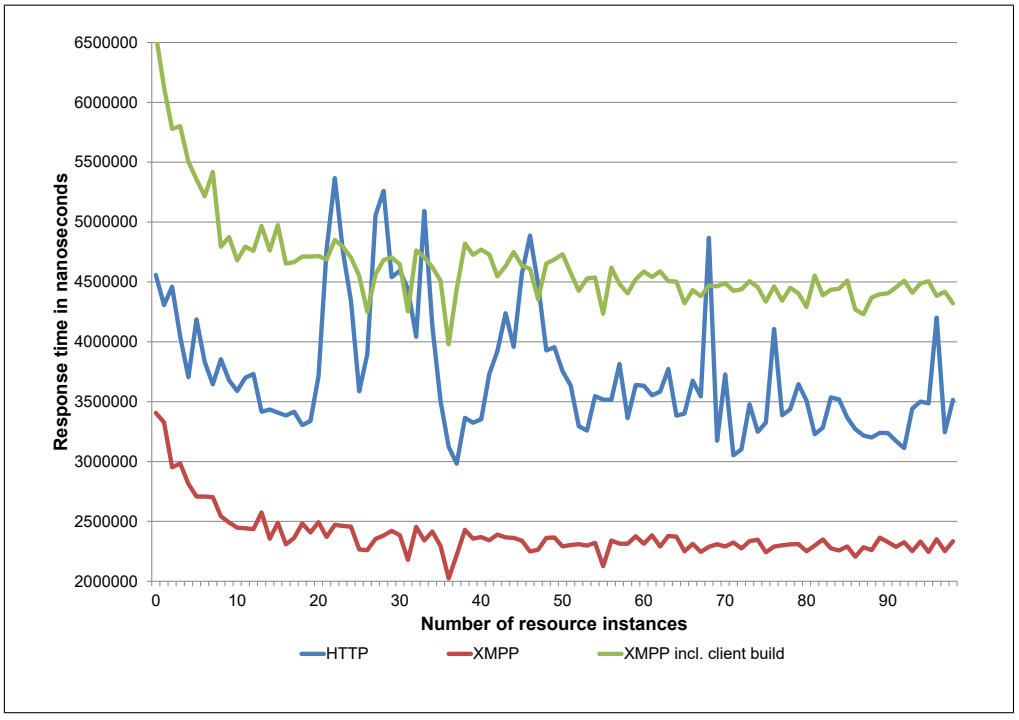

FIGURE 5.8.: Performance of deleting a compute resource

Figure 5.8 compares response times for deleting the virtual machine that has been created in the previous step. This figure has three curves similar to the first step, but in this scenario the green curve depicts the entire process time for requesting the resource's XWADL document, creating the client based on that XWADL, and invoking the `DELETE` method that terminates the virtual machine resource at the given URI. The red curve in contrast only shows the method invocation, i.e. only two network transmissions: one for the request and one for the response.

The performance measurements show significant improvements. Depending on the implementation and the scenarios the REST with XMPP API is designed for, this approach is not only an alternative to HTTP but rather a serious competitor.

5.4. REST with XMPP Rendering

As described in section 3.6 the OCCI standard consists of three specifications: The OCCI-Core [64], the OCCI-HTTP Rendering [66], and the OCCI-Infrastructure [67]. Since May 2015 the OCCI working group published the new version 1.2 for public comments. This version is not standardized until today, but will be in this form or with less modifications in the near future. The proposed version 1.2 consists of eight specifications built on each other. This new specification family is quite a bit more modular than the standardized version 1.1. For example, the OCCI-HTTP Rendering v1.1 has been split into Text Rendering v1.2, JSON Rendering v1.2, and HTTP Protocol v1.2. Additionally three further OCCI extensions have been introduced to become a part of this specification family: The OCCI Platform Model, the OCCI SLA Model, and the OCCI Monitoring Model. The following table 5.2 shows a comparison of required implementations between these two versions:

TABLE 5.2.: Comparison of required OCCI specification implementations

Specification	**OCCI v1.1**	**OCCI v1.2**
Core Model	MUST	MUST
Text Rendering	MUST	MUST
JSON Rendering	MAY	MUST
HTTP Protocol	MUST	MUST
Infrastructure Model	SHOULD	SHOULD
Platform Model	—	MAY
SLA Model	—	MAY
Monitoring Model	—	MAY

Both OCCI specification families in version 1.1 and in version 1.2 define the fundamental OCCI Core Model that is renderable and applicable for the REST architecture style. However, only the applicability to HTTP is defined. Though version 1.2 specifies renderings for `application/occi+json`, `text/uri-list`, `text/plain`, and `text/occi` media types which can also be used for this approach, the REST with XMPP protocol provides more capabilities to express the classification and identification model. Moreover, XMPP is XML-based and an XML rendering would not only fit better in the XMPP concept but also cloud increase the performance, because stanzas are transmitted as XML streams and are also interpreted in that way. For these reasons the application of the REST with XMPP protocol and an XML rendering for the OCCI Core Model is presented in the following. In particular, the classification model is expressed in XWADL documents and two novel media types for XML rendered representations, namely `xmpp/occi` and `xmpp/occi-list` are introduced.

5.4.1. Classification Rendering

The OCCI-HTTP Rendering specification defines a query interface located at the path `/-/` of the root of the OCCI implementation [66]. This query interface has the purpose to expose types and capabilities of an OCCI implementation. Here, all defined types including *Kinds*, *Mixins*, *Attributes*, *Links*, and *Actions* are exposed. These definitions are extended with a location attribute that advertises the location for which the type can be applied to. A filter mechanism is also defined and must be supported by an implementation, because the amount of data for such a query interface can be quite quickly become very huge.

The classification rendering for REST with XMPP follows another approach. As specified before, an XWADL document advertises just the capabilities of a specific resource instance at a specific path. Thus not all types must be retrieved but just the types which belong to this particular resource. Furthermore, the location attribute is not required, because resources are discovered and explored by the nature of the REST with XMPP protocol. In contrast to the HTTP rendering, the REST with XMPP rendering also exposes all methods which are available for a particular resource. Moreover, *Actions* are exposed and its parameterization is explorable in the same fashion. The *Kind* of a resource instance with its related *Mixins*, *Links*, and *Attributes* are defined in the grammars section of XWADL documents as illustrated in listing 5.14.

```
...
<resource_type path="/compute" xmlns="urn:xmpp:rest-xwadl">
  <documentation title="Summary">This resource allows for managing
      compute instances, e.g. creating virtual machines.
  </documentation>
  <grammars>
    <urn1:Classification xmlns:urn1="urn:xmpp:occi-classification">
      <urn1:KindType> ... </urn1:KindType>
      <urn1:MixinType> ... </urn1:MixinType>
      <urn1:LinkType> ... </urn1:LinkType>
    </urn1:Classification>
</grammars>
...
```

LISTING 5.14: Example grammars element for classification rendering

The classification of a resource exposed in the grammars section of an XWADL document has to be in compliance with the classification schema that can be found in appendix A.3. Here, the `Classification` element forms the root element of a classification document and may comprise exactly one *Kind* classification and an unbounded set of *Mixin* and *Link* classifications as sub-elements.

Category Classification

The `CategoryClassification` type is the base type of all other classification types: `KindType`, `MixinType`, and `LinkType`. This `CategoryClassification` type specifies the `schema`, the `term`, the `title`, and a set of `attributeClassification` of any inherited category type. This classification enables an unique assignment of representations and types according the OCCI Core Model. While the `schema` and the `term` element specifies the assignment, the `attributeClassification` defines the attributes that are able to apply to this particular category. The sub-elements of an `attributeClassification` are the same as specified in the OCCI Core Model and are exemplarily depicted in listing 5.15.

Each `attributeClassification` must have one `name`, one `type`, one `mutable`, one `required`, zero or one `default`, and zero or one `description` element. These sub-elements specify a particular attribute exactly, its type, and how this attribute is handled. The schema defines a static set of possible types that can be assigned to an attribute which are: `STRING`, `ENUM`, `INTEGER`, `FLOAT`, `DOUBLE`, `BOOLEAN`, `URI`, `SIGNATURE`, `KEY`, `DATETIME`, `DURATION`, `LIST`, and `MAP`.

```
<KindType>
  <term>compute</term>
  <schema>http://schema.ogf.org/occi/infrastructure#</schema>
  <title>Compute Resource</title>
  <attributeClassification>
    <name>occi.compute.cores<name>
    <type>INTEGER</type>
    <mutable>true</mutable>
    <required>false</required>
    <default>1</default>
    <description>Number of virtual CPU cores assigned to the
       instance</description>
  </attributeClassification>
    ...
</KindType>
```

LISTING 5.15: Example kind and attribute classification rendering

Mixin and Link Assignment

In contrast to the `KindType`, the `MixinType` and the `LinkType` extend the `Category Classification` type each with one required additional sub-element as illustrated in listing 5.16. The `LinkType` has a child element called `relation` that defines the classification type which this link can point to. And the `MixinType` has a child element called `applies` that defines the classification type which this mixin can applied to. While all these category types are listed flat, a mixin could be applied to a listed kind, to a listed link, to another listed mixin, or to the basis category schema. If a mixin applies to the `http://schema.ogf.org/occi/core#category` type, then this mixin can be applied to any other kind or link.

```
...
<MixinType>
  <term>ipnetworkinterface</term>
  <schema>http://schema.ogf.org/occi/infrastructure/networkinterface#
    </schema>
    ...
  <applies>http://schema.ogf.org/occi/infrastructure#networkinterface
    </applies>
</MixinType>
<LinkType>
  <term>networkinterface</term>
  <schema>http://schema.ogf.org/occi/infrastructure#</schema>
    ...
  <relation>http://schema.ogf.org/occi/infrastructure#network
```

```
    </relation>
</LinkType>
...
```

LISTING 5.16: Example mixin and link classification rendering

5.4.2. Representation Rendering

The representation of a resource has to be in compliance with the classification expressed in a XWADL document's grammar section and with the representation schema that can be found in appendix A.4. While a classification document has a `Classification` element as its root and a `KindType` element, a set of `MixinType` elements, and a set of `LinkType` elements as child elements, a representation document has a `Category` element as its root and a `Kind` element, a set of `Mixin` elements, and a set of `Link` elements as child elements. Hence, the `Kind`, the `Mixin`, and the `Link` elements still have a `schema` and a `term` element in order to identify their classification, no additional declarations of attributes, relations, and possible applications are included in a representation. An attribute is identified by its name and has a particular sub-element that can be validated against the schema and has to be analogous to the classification declarations as illustrated in listing 5.17. This allows to transmit values in a representation which are string-based but are parsed in a specific number encoding.

```
...
<request mediaType="xml/occi">
  <Category xmlns="urn:xmpp:occi-representation">
    <Kind>
      <term>compute</term>
      <schema>http://schema.ogf.org/occi/infrastructure#</schema>
      <title>Compute Resource</title>
      <attribute name="occi.compute.memory">
        <DOUBLE>4.0</DOUBLE>
      </attribute>
      <attribute name="occi.compute.cores">
        <INTEGER>2</INTEGER>
      </attribute>
    </Kind>
      ...
  </Category>
</request>
...
```

LISTING 5.17: Example representation of a kind

While the classification of kinds, mixins, and links is flat, the representation of them is sorted. In particular, mixins which are defined and applied to a kind are listed as a set of elements under the `Category` element and mixins which are defined and applied to a link are listed as child elements of that `Link` which they are applied to.

5.4.3. Implementation Concept

Annotations are ideal to add metadata to classes, fields, and methods. Therefore, the implementation concept for the OCCI rendering makes also use of annotations similar to the implementation concept for REST with XMPP. Since the data model of a representation and its classification have to be assigned to Java classes, two base classes are introduced: `Category` and `LinkCategory`. Four additional annotations are defined which allow to assign classification metedata to classes and fields: `Kind`, `Mixin`, `Link`, and `Attribute`. Listing 5.18 illustrates how these annotations are used. Here, the *ComputeKind* is defined.

```
@Kind(schema = "http://schema.ogf.org/occi/infrastructure#",
      term = "compute")
public class ComputeKind extends Category {

   @Attribute(name = "occi.compute.cores",
              type = AttributeType.INTEGER,
              mutable = true,
              required = false,
              value = "1",
              description = "Number of virtual CPU cores assigned
                             to the instance")
   public Integer cores = 1;
   ...
}
```

LISTING 5.18: Compute *Kind* declaration

The annotations which can be assigned to a class (i.e. `Kind`, `Mixin`, `Link`) require a definition of a schema and a term that identifies that particular classification type. In contrast to the `Kind` annotation, the `Mixin` and the `Link` annotation need more information for their definition as illustrated in listing 5.19. The `Link` annotation has a `relation` field that defines the classification type which this link can point to. And the `Mixin` annotation has a `applies` field that defines the classification type which this mixin can be applied to. However, both fields have the `http://schema.ogf.org/occi/core#category` definition as default value which makes the fields optional.

```
@Link(schema = "http://schema.ogf.org/occi/infrastructure#",
      term = "networkinterface")
      relation =
   "http://schema.ogf.org/occi/infrastructure#network")
public class NetworkInterfaceLink extends LinkCategory {
   ...
}

@Mixin(schema =
   "http://schema.ogf.org/occi/infrastructure/networkinterface#",
       term = "ipnetworkinterface",
       applies =
   "http://schema.ogf.org/occi/infrastructure#networkinterface")
public class IpNetworkInterfaceMixin extends Category {
   ...
}
```

LISTING 5.19: Network interface *Link* and *Mixin* declaration

Additionally, three base classes are introduced: `Collection`, `Resource`, and `Link`. These base classes provide basic functionalities according to the OCCI Core Model specification. The sub-classes which inherit from one of these base classes have to be annotated with an optional summary that explains the purpose of this resource and a `Classification` annotation that specifies the *Kind*, *Mixins*, and *Links* that are assigned to this resource. Listing 5.20 illustrates their use by the example of a virtual machine which is used in the previous sections as well.

```
@Summary("This resource allows for managing a particular virtual
   machine.")
@Classification(kind = ComputeKind.class,
                mixins = {IpNetworkInterfaceMixin.class},
                links = {NetworkInterfaceLink.class})
public class VirtualMachine extends Resource {
   ...
}
```

LISTING 5.20: Compute resource classification annotation

5.5. Intercloud Agreement-Mediators

This thesis aims to develop an autonomous web service based *Agreement-Mediator* as a third party that sources out SLA management tasks into a neutral zone without taking the consumer's

or the provider's side. Thus, the *Agreement-Mediator* adds an additional transitive relationship between service providers and service consumers. Figure 5.9 depicts the developed architecture and service distribution in the intercloud overlay network. This architecture facilitates service discovery mechanisms and allows for comparing services based on their advertised QoS. Furthermore, customers are able to negotiate individual SLAs before they come into effect. In particular, the goals achieved with this approach are:

- services are advertised with individual quality levels and guarantees that a provider is willing and able to deliver;
- consumers are able to discover and to easily compare service offerings of different providers;
- providers are able to publish service offerings of mutual intentions for business relationships in the form of SLA offers;
- monitoring services are supported in order to feed in monitoring measurements for automatic compliance verification of SLA terms;
- participating parties are notified about agreement-related events like creation, violation or termination;
- the overhead for managing SLAs between consumers and providers is minimized by intercloud developed ready-to-use solutions; and
- higher trust is established by transferring mediation affiliation into an external and neutral entity.

The *Intercloud Agreement-Mediator* is basically provided by the *Intercloud Exchange*. However, services from all intercloud topology elements are used to provide the promised functionalities as depicted in figure 5.9. For instance, the *Intercloud Root* hosts a global service repository which dynamically changes, based on advertisements that provider published in the service catalog all along. This service catalog is a global marketplace to which all participating organizations have access to. The *Intercloud Exchange* hosts a service catalog as well. This local service repository hosts in its service catalog only the service offering of the provider which this *Intercloud Exchange* belongs to. The customers are not allowed to access the global service repository directly, but have to browse for a desired service in the local service repository of its provider. Thus, providers are able to decide which product offering they pass on to their customers. One reason for this design decision is its facilitation for providers to apply filter mechanisms on product offer which they want to pass on to their customer. Another reason is a business related aspect that providers would like to sell their services first. After all their resources are in use further third-party service are offered. Of course, if the providers do not

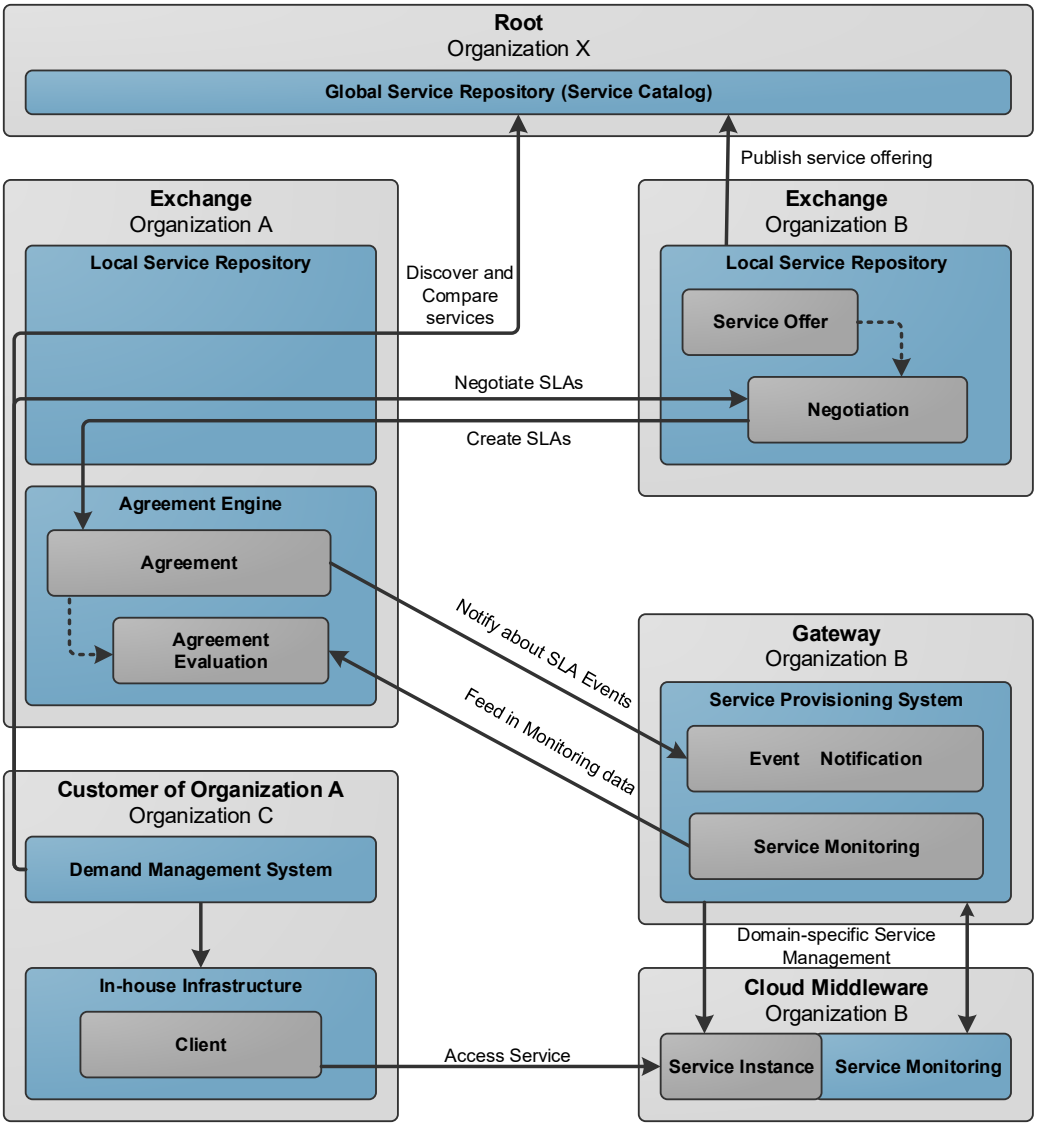

FIGURE 5.9.: Architecture of the Intercloud *Agreement-Mediator*

have a desired service or any services at all, the provider plays the role of a broker and supports the user with advanced functionalities to find, compare, and use a service of a third-party provider. In general, if a provider has its own local service repository and appropriated access to the global service repository, each provider is able to design his own customized marketplace available over HTTP.

When a customer has found a potential product that is considerable for purchase, the customer can buy the service as advertised or can start a negotiation. However, not all service offerings in a service repository are negotiable. Whether a service offer is negotiable or not depends on the provider who offered that service. If the customer buys a service without any negotiation, an agreement is created at the *Intercloud Exchange* of the provider that enables its customer

the access to the intercloud network. Thus, this provider is an autonomous notary in this case. If the service offer is negotiable, a negotiation instance is created at the *Intercloud Exchange* of the provider that offers the service. If both parties agree to conditions negotiated in multiple rounds, an agreement is created as in the case of no negotiation. Thus, the agreement instance is always fully under the control of an independent party. Hence, providers can also offer their own services to customers, the agreement is created at the *Intercloud Exchange* of the same provider. This would obviously destruct the impartiality of the *Intercloud Agreement-Mediator*.

If an agreement is established, this agreement serves as formal contract between the customer and the provider of the service. The content of this agreement reflects the business relationship and the service reference, conditions, and guarantees that are related to this particular transaction. If the customer buys more than one service or adds additional services to this business relationship with the same provider later on, the established agreement is updated and no additional agreements are created. In fact, each customer has a single agreement with a particular provider to which a business relationship exists. Each service with appropriated guarantees and further declarations is expressed as a link, i.e. *ServiceDescriptionTerm*, *ServiceReference*, or *GuaranteeTerm*.

The evaluation of an agreement is based on monitoring measurements that have to be provided either by the provider of the service or by the customer. These measurements have to be passed to an event stream provided by the *Intercloud Exchange* where the agreement instance is located. The data model used to transmit measurements has to be in compliance with the XEP-0337: Event Logging over XMPP protocol extension [158]. Thus, any kind of information can be reported and evaluated. If an agreement violation is detected or any other event occurs which is well worth to inform, the basic capabilities of XMPP are used. In particular, a message stanza is sent to the provider of the service as well as to the customer who is affected by this event.

5.6. Protocol Extensions

The previous section described the *Intercloud Agreement-Mediator* solution. However, such an architecture requires a set of protocol extensions and adjustments which are not specified in the OCCI specification family. Therefore, this section presents and specifies the missing extensions in order to establish such an *Intercloud Agreement-Mediator* with its capabilities and behavior.

5.6.1. Monitoring Model

The OCCI Monitoring Model specification based fully on the OCCI Core Model version 1.1. The purpose of this specification is to define a standardized interface that enables users to request the creation of a distinct cloud monitoring infrastructure, to configure this infrastructure according to pre-defined requirements, and to access quantitative measurements of the performance collected by this infrastructure for a specific set of provisioned cloud services. The OCCI Monitoring Model introduces two new entity types which are: the *Sensor* and the *Collector*. The *Collector* type represents a link that defines timings used for the measurement. The *Sensor* type represents a resource that defines how to process measurement results.

However, this specification is still under development and contains some bugs (e.g. a reference to further OCCI extensions like OCCI Notification Extension which do not exist, duplicated attributes in the *Collector* link and the *Sensor* resource, etc.). Therefore, an advanced and idealized specification for the Monitoring Model is presented in this subsection. This model also has a *Sensor* and a *Collector* type, but with different attributes, different behavior, and in general a different intended purpose. The model described in the following distinguishes between two natures of sensors: an active sensor and a passive sensor. A passive sensor is defined as an instrument that receives data and measures a provisioned service without sampling or polling information from a process. In contrast, an active sensor has to continuously sample the performance of a service and has to actively poll measurements from a process. The nature of these sensors is schematically depicted in figure 5.10.

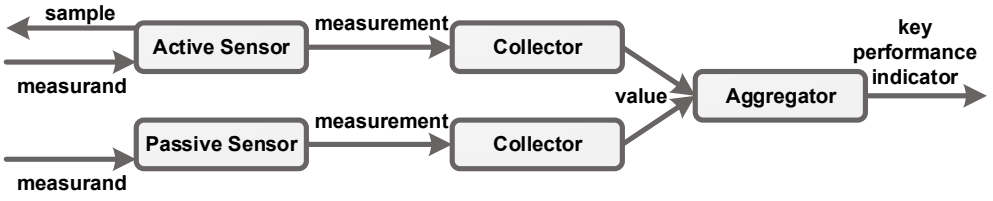

FIGURE 5.10.: Schematical monitoring infrastructure

Figure 5.10 illustrates a schematical composition of instruments of a measurement chain. Here, a passive and an active sensor collect raw information of the performance, e.g. a service is available or not (true/false). This raw information is processed as measurement with a specific unit (e.g. Bolean) to a collector. The collector gathers the measurements for a specific time period and calculates a value based on pre-defined metrics, e.g. an availability of 99% in the last 24 hours. This value can then simply be processed as it is or aggregated with other values, e.g. to calculate the availability for a set of services. The result is a Key Performance Indicator (KPI) for a particular monitoring infrastructure.

In order to support a similar measurement chain with this monitoring model, three types are introduced: the *Sensor* and the *Meter*, and the *Collector* type. Figure 5.11 gives an overview of these key types involved in this specification.

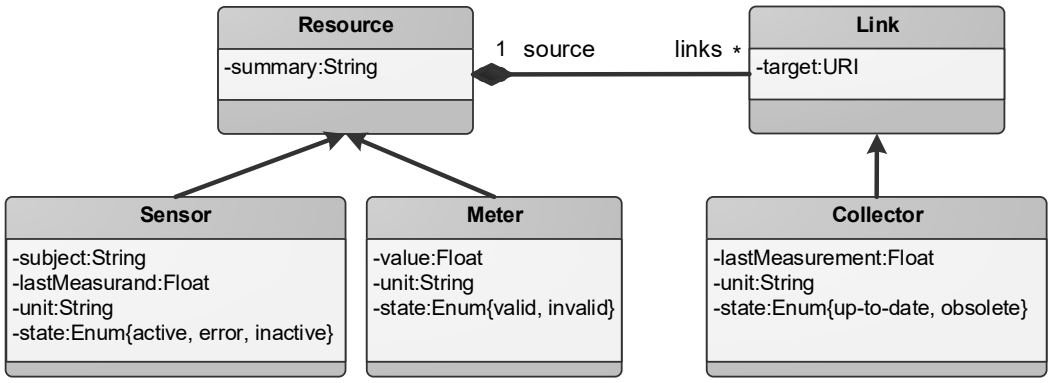

FIGURE 5.11.: Overview diagram of monitoring model types

The monitoring types inherit the OCCI Core Model *Resource* and *Link* base type and all their attributes. Table 5.3 describes the *Mixin* instances defined for each of the monitoring *Resource* or *Link* sub-types. The base URL *http://schemas.cit.tu-berlin.de/occi* has been replaced with <*schema*> in this table for better readability. These types can be extended according to the OCCI Core Model [64].

TABLE 5.3.: *Mixin* instances defined for the monitoring sub-types.

Term	Scheme	Title	Parent *Kind*
sensor	<schema>/monitoring#	Sensor *Resource*	<schema>/core#resource
meter	<schema>/monitoring#	Meter *Resource*	<schema>/core#resource
collector	<schema>/monitoring#	Collector *Link*	<schema>/core#link

The following subsections describe the *Sensor*, *Meter* and *Collector* types in detail. Furthermore the *Attribute*s, *Actions* and states definition for each of them, including type-specific mixins, are specified.

Sensor

The *Sensor* type represents a generic sensor resource, e.g. an error scanner or a reachability sampler like *ping*. The *Sensor* inherits the *Resource* base type defined in OCCI Core Model [64]. It is assigned to the *Mixin* instance *http://schemas.cit.tu-berlin.de/occi/monitoring#sensor* which has to be exposed by each instance of *Sensor*.

TABLE 5.4.: *Attributes* defined for the *Sensor* type.

Attribute	Type	Multi-plicity	Mutability	Description
occi.sensor.subject	String	0...1	Immutable	A unique identifier that specifies the subject under test.
occi.sensor.lastmeasurand	Float	0...1	Immutable	Current measurand of the last sample.
occi.sensor.unit	String	0...1	Immutable	The unit of the measurand.
occi.sensor.state	Enum {active, inactive, error}	1	Immutable	Current state of the instance.

Table 5.4 describes the attributes defined by the *Sensor* through its *Mixin* instance. These attributes should be exposed by an instance of the *Sensor* type depending on the "Multiplicity" column in the aforementioned table.

TABLE 5.5.: *Actions* defined for the *Sensor* type

Action Term	Target state	Attributes
start	active	–
stop	inactive	–

Table 5.5 describes the *Actions* defined for *Sensor* by its *Resource* instance. These *Actions* must be exposed by an instance of the *Sensor* type of the monitoring model implementation. Figure 5.12 illustrates the state diagram for a *Sensor* instance.

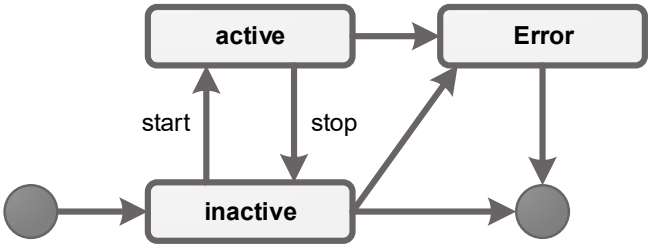

FIGURE 5.12.: State diagram for a *Sensor* instance

A sensor is in the "inactive" state after creation. In this state the sensor does not sample the performance of a service and does not accept any changes to the last measurand. The user or the provider have to deliberately change the state to "active" by invoking the *start* action.

The purpose for this design is justified by the influence of a monitoring overhead that may have impact on the performance of the subject under test. The user or the provider can stop sampling processes by invoking the *stop* action that changes the state of a sensor to "inactive". If an error occurs, the sensor should be in the "error" state. The state of a sensor has direct impact on all other instances of *Collector* links and *Meter* resources in a measurement chain that has a particular sensor as a source.

ActiveSensor Mixin

An *ActiveSensor* mixin introduces an attribute which is required for configuring instances of an active sensor resource. An active sensor in contrast to a passive sensor actively samples periodically the performance of a service to be monitored. The *ActiveSensor* mixin is assigned the schema *http://schemas.cit.tu-berlin.de/occi/monitoring/sensor#* and the "term" value *activesensor*.

TABLE 5.6.: *Attributes* defined by the *ActiveSensor* mixin

Attribute	Type	Multiplicity	Mutability	Description
occi.sensor.samplerate	Time (ISO8601)	1	Mutable	Sampling rate with which the sensor scans a subject periodically.

Table 5.6 defines the attribute introduced by the *ActiveSensor* mixin. This mixin must be related to the *Sensor* mixin by setting the *applies* attribute to *http://schemas.cit.tu-berlin.de/occi/monitoring#sensor*.

Figure 5.13 illustrates an example UML object diagram where a *Sensor* is associated with an *ActiveSensor* mixin when both are instantiated. It shows an example for an active sensor that checks the availability of a virtual machine every 5 minutes if the virtual machine was reachable at the last sample.

PassiveSensor Mixin

A *PassiveSensor* mixin introduces an attribute which is required for configuring instances of a passive sensor resource. A passive sensor in contrast to an active sensor does not samples periodically, but is event driven. The *PassiveSensor* mixin is assigned the schema *http://schemas.cit.tu-berlin.de/occi/monitoring/sensor#* and the "term" value *passivesensor*.

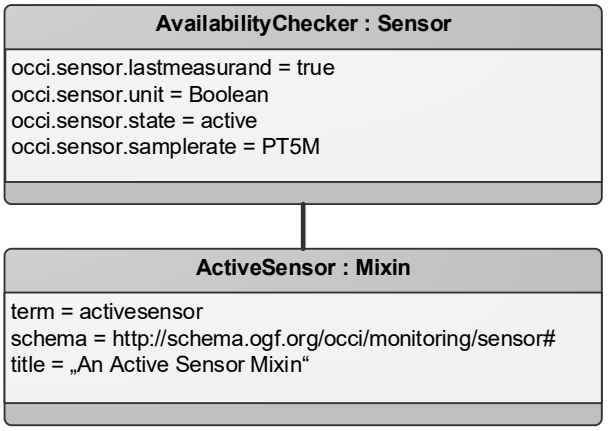

FIGURE 5.13.: Example of a *Sensor* instance associated with an *ActiveSensor* mixin

TABLE 5.7.: *Attributes* defined by the *PassiveSensor* mixin

Attribute	Type	Multiplicity	Mutability	Description
occi.sensor.lastoccurrence	Time (ISO8601)	1	Immutable	Timestamp of the last occurrence of a sampling event.

Table 5.7 defines the attribute introduced by the *PassiveSensor* mixin. This mixin must be related to the *Sensor* mixin by setting the *applies* attribute to *http://schemas.cit.tu-berlin.de/occi/monitoring#sensor*.

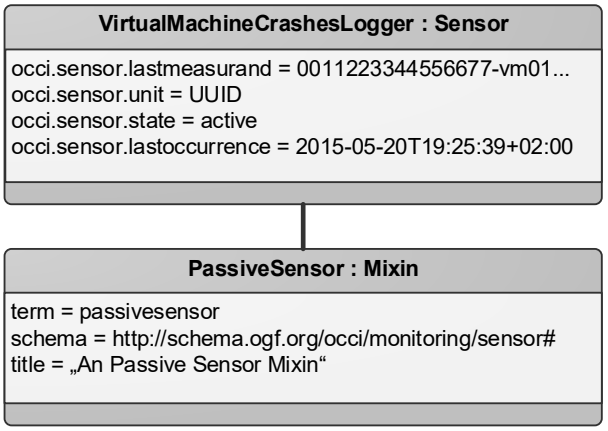

FIGURE 5.14.: Example of a *Sensor* instance associated with a *PassiveSensor* mixin

Figure 5.14 illustrates an example UML object diagram where a *Sensor* is associated with a

PassiveSensor mixin when both are instantiated. It shows an example for a passive sensor that logs virtual machine crashes. In the example, the last virtual machine that had been crashed and logged by this sensor was the machine with UUID "...vm01 ...". This event occurred on May 20, 2015, 07:25:39 pm, Central European Time (CET).

Meter

The *Meter* type represents a generic meter resource, e.g. a service availability meter or a network traffic meter like *nload* or *iftop*. The measurements provided by such a meter can be collected individually or sampled by a *Sensor* resource instance, gathered by *Collector* link instances and processed within the *Meter* resource instance. The *Meter* inherits the *Resource* base type defined in OCCI Core Model [64]. It is assigned to the *Mixin* instance *http://schemas.cit.tu-berlin.de/occi/monitoring#meter* which has to be exposed by each instance of *Meter*.

TABLE 5.8.: *Attributes* defined for the *Meter* type

Attribute	Type	Multiplicity	Mutability	Description
occi.meter.value	Float	0...1	Immutable	Current value of the service performance.
occi.meter.unit	String	0...1	Immutable	The unit of the value.
occi.meter.state	Enum {valid, invalid}	1	Immutable	Current state of the instance.

Table 5.8 describes the attributes defined by the *Meter* through its *Mixin* instance. These attributes should be exposed by an instance of the *Meter* type depending on the "Multiplicity" column in the aforementioned table.

The *Meter* type has no *Actions*, but a state that indicates whether the value is "valid" or "invalid". These states must be exposed by an instance of the *Meter* type of the monitoring model implementation. Figure 5.15 illustrates the state diagram for a *Meter* instance.

A meter is in the "invalid" state after creation. The meter persists in this state until a valid value is exposed via the "occi.meter.value" attribute. When a valid value is exposed, the meter changes its state to "valid". If an instance of the *Meter* type receives its measurements from an instance of a *Collector* link type, the state of the collector has direct impact on the state of this meter. In fact, if the collector provides obsoleted measurements, the value of the meter is invalid and should advertise this state through the "invalid" state of the meter instance.

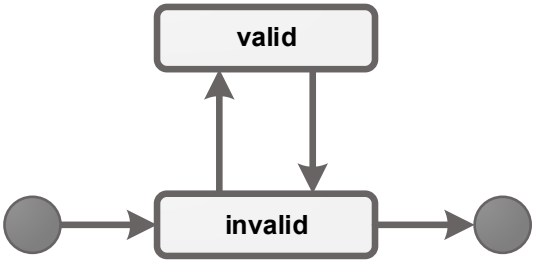

FIGURE 5.15.: State diagram for a *Meter* instance

Collector

The *Collector* type is responsible for gathering measurements from a sensor and to calculate a value based on an applied metric. The *Collector* inherits the *Link* base type defined in OCCI Core Model [64]. It is assigned to the *Link* instance *http://schemas.cit.tu-berlin.de/occi/monitoring#collector* which has to be exposed by each instance of *Collector*.

TABLE 5.9.: *Attributes* defined for the *Collector* type

Attribute	Type	Multiplicity	Mutability	Description
occi.collector.measurement	Float	0...1	Immutable	Current measurement based on a calculated aggregation of samples.
occi.collector.unit	String	0...1	Immutable	The unit of the measurement.
occi.collector.state	Enum {up-to-date, obsolete}	1	Immutable	Current state of the instance.

Table 5.9 describes the attributes defined by the *Collector* through its *Link* instance. These attributes should be exposed by an instance of the *Collector* type depending on the "Multiplicity" column in the aforementioned table.

The state of a *Collector* instance indicates whether the value is "up-to-date" or "obsolete". These states must be exposed by an instance of a *Collector* type of the monitoring model implementation. Figure 5.16 illustrates the state diagram for a *Collector* instance.

Whether a measurement is up-to-date or not depends on the source of the *Link* instance. A *Link* instance has to be in the state "pending" after instantiation. The collector has to persist in this state until a calculation with minimum one measurement is possible. Afterwards the state of a

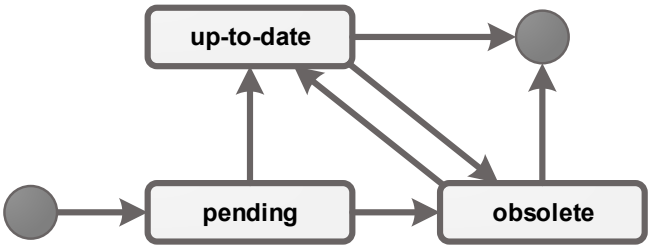

FIGURE 5.16.: State diagram for a *Collector* instance

Collector instance has to be switched into the state “up-to-date” or “obsolete”. During its life time, a *Collector* instance can switch between “up-to-date” and “obsolete”. In fact, the state of a collector depends on the up-to-dateness of the measurement that it exposes. If no calculation can be performed in an exposed time period, the state of a collector should be in the “obsoleted” state.

5.6.2. Service Level Agreement Model

The OCCI SLA Model specification is based on the OCCI Core Model version 1.2. The goal of this specification is to define a standardized SLA interface that enables users to create and manage SLA related resources. In particular, these SLA related resources realize agreements between an OCCI-enabled cloud service provider and potential consumers. The OCCI SLA Model introduces two new entity types which are: the *Agreement* and the *AgreementLink*. The *Agreement* type represents a resource that defines an agreement. This agreement resource is used for negotiation, service instantiation, and evaluation during the whole life-cycle of a particular business relationship. The *AgreementLink* type represents links of an agreement resource to provisioned service resources after both parties have agreed to an particular agreement.

However, this specification is defined for an interface of a single cloud installation. Thus, resources cannot be distributed among different systems which is the case in the intercloud architecture and a requirement in order to transfer mediation affiliations to an external neutral entity. Therefore, an advanced and comprehensive specification for the SLA Model is presented in this subsection. This model also has an *Agreement* and an *AgreementLink* type, but with different attributes, different behavior, and in general a different intended purpose. Additionally, an *Offer* and a *Negotiation* resource is introduced as well as a *NegotiationLink*, a *ServiceReference* link, a *ServiceDescriptionTerm* link, and a *GuaranteeTerm* link. This extended SLA Model allows for distributing SLA and monitoring resources over different entities as depicted in figure 5.17.

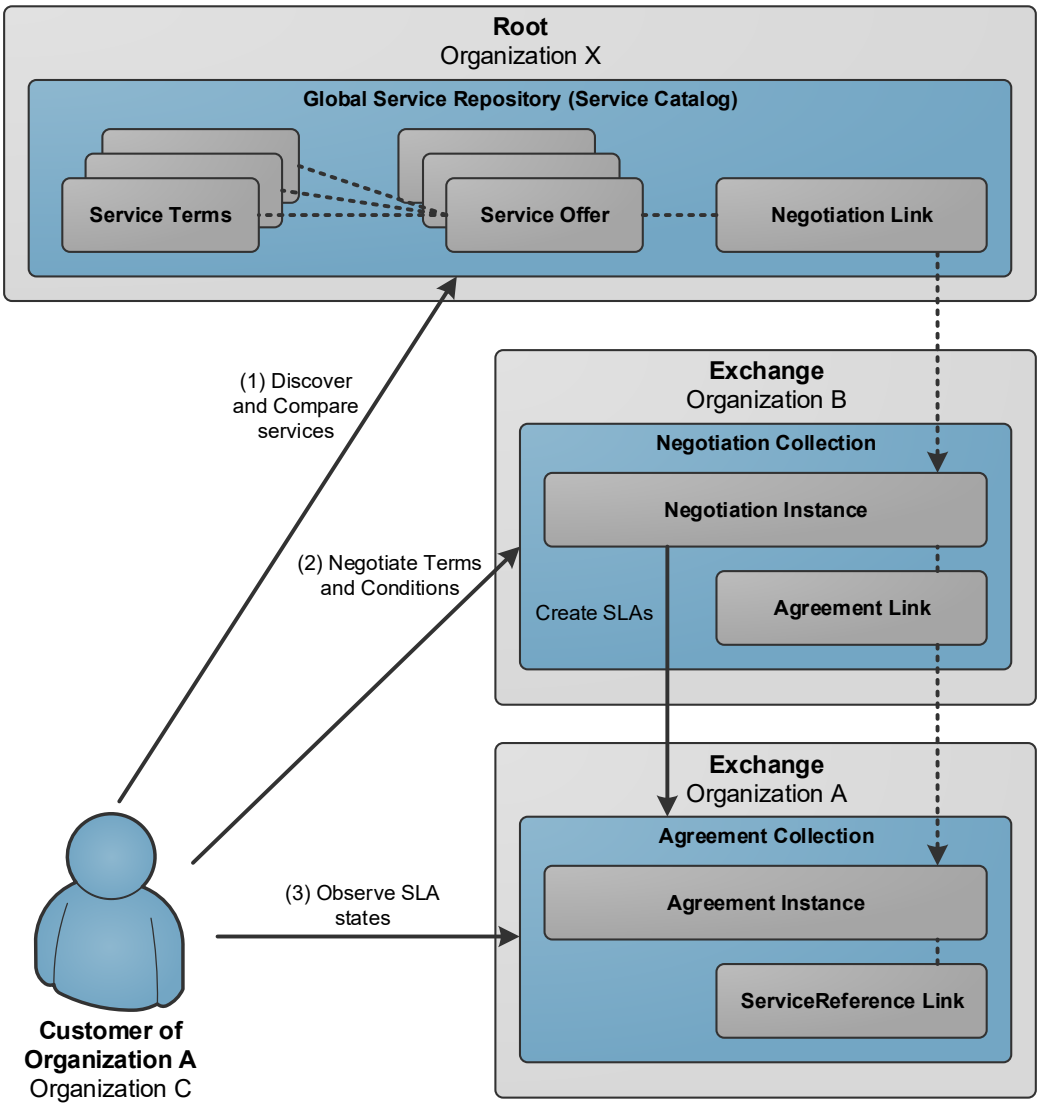

FIGURE 5.17.: Schematical architecture of the SLA model

Figure 5.17 illustrates a possible distribution of resources. In this scenario (1) a customer searches a desired service and retrieves potential offers from the global service repository hosted at an *Intercloud Root*. The representations retrieved from the service catalog include pre-defined terms expressed as links but without a target definition. Additionally, one *NegotiationLink* is part of the representation and points to a *Negotiation* collection hosted at an *Intercloud Exchange*. If the user initiates a negotiation, the customer has to create a *Negotiation* instance with the offer representation retrieved before. After a *Negotiation* instance has been created, (2) the customer is able to negotiate specific terms and conditions in multiple rounds. If both parties achieve a common agreement, an *Agreement* instance is created at a third-party *Intercloud Exchange*. The provider of the third-party *Intercloud Exchange* is independent in this transaction. This *Intercloud Exchange* hosts the agreement instance for its whole life-time.

When the agreement has been created successfully and all links are instantiated and configured, the SLA is evaluated continuously. (3) The customer and the provider are able to observe the state of the established agreement. Furthermore, if an unexpected event occurs, all parties are informed by a basic XMPP message stanza. Figure 5.18 gives an overview of these key types involved in this specification.

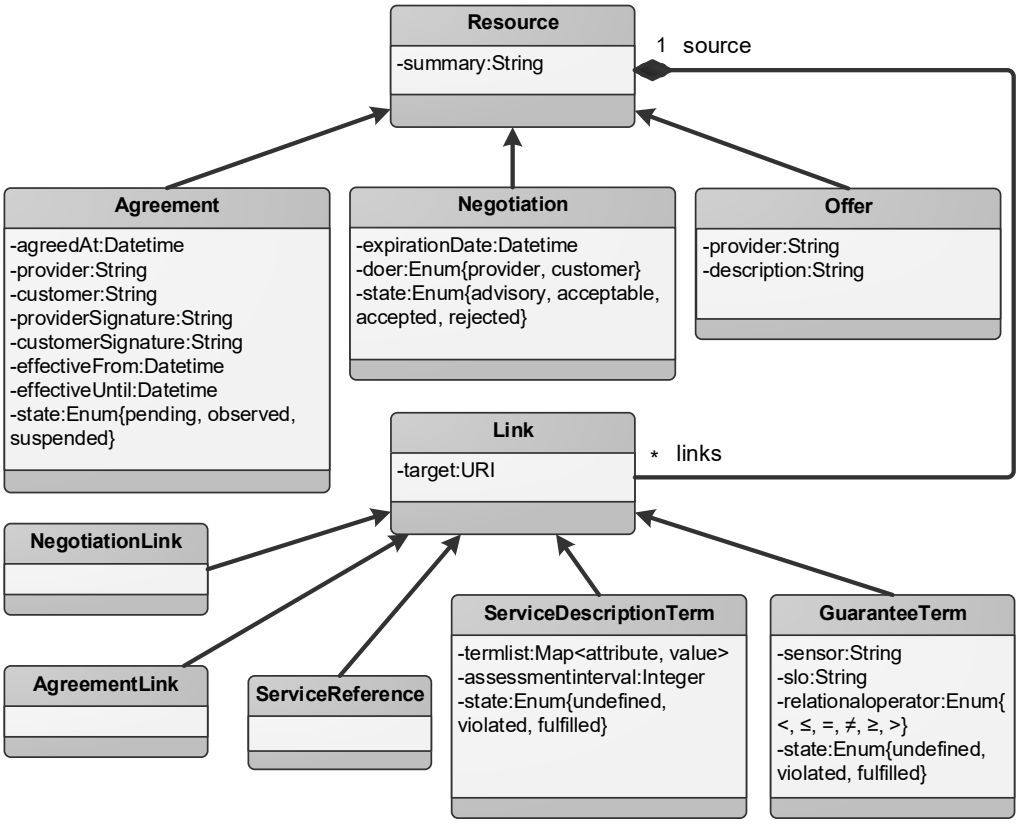

FIGURE 5.18.: Overview of SLA types

Offer

The *Offer* type represents a generic service offer resource that may be exposed in a local service repository or in the global service catalog. The *Offer* inherits the *Resource* base type defined in OCCI Core Model [64]. It is assigned to the *Kind* instance *http://schemas.cit.tu-berlin.de/occi/sla#offer* which has to be exposed by each instance of *Offer*. Table 5.10 describes the attributes defined by the *Offer* through its *Kind* instance.

TABLE 5.10.: *Attributes* defined for the *Offer* type

Attribute	Type	Multi-plicity	Mutability	Description
occi.offer.provider	String	0...1	Immutable	The domain that identifies the provider.
occi.offer.description	String	0...1	Immutable	A textual description of the service.

Negotiation

The *Negotiation* type represents a generic negotiation resource that is used to negotiate an SLA based on an exposed *Offer* instance. The *Negotiation* inherits the *Resource* base type defined in OCCI Core Model [64]. It is assigned to the *Kind* instance *http://schemas.cit.tu-berlin.de/occi/sla#negotiation* which has to be exposed by each instance of *Negotiation*.

TABLE 5.11.: *Attributes* defined for the *Negotiation* type

Attribute	Type	Multi-plicity	Mutability	Description
occi.negotiation .expirationdate	Time (ISO8601)	1	Immutable	The date time at which this negotiation expires.
occi.negotiation .doer	Enum {provider, customer}	1	Immutable	The party of this negotiation who have to carries out the next negotiation step.
occi.negotiation .state	Enum {advisory, acceptable, accepted, rejected}	1	Immutable	The current state of the negotiation instance.

Table 5.11 describes the attributes defined by the *Negotiation* through its *Kind* instance. Table 5.12 describes the *Actions* defined for *Negotiation* by its *Resource* instance. These *Actions* must be exposed by an instance of the *Negotiation* type of an SLA model implementation.

The state of a *Negotiation* instance indicates the progress of the negotiation are either “advisory”, “acceptable”, “accepted”, or “rejected”. These states must be exposed by an instance of a *Negotiation* type of the SLA model implementation. Figure 5.19 illustrates the state diagram for a *Negotiation* instance.

TABLE 5.12.: *Actions* defined for the *Negotiation* type

Action Term	Target state	Attributes
agree	acceptable	Customer's Signature
reject	rejected	–
sign	accepted	Provider's Signature

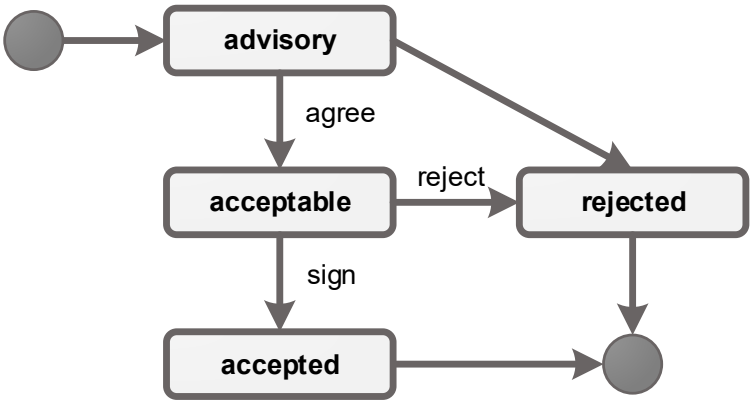

FIGURE 5.19.: State diagram for a *Negotiation* instance

A *Negotiation* instance has to be in the state "advisory" after instantiation. The *Negotiation* instance has to persist in this state during the whole negotiation process. If the service provider offers an acceptable counter-offer, the customer is able to accept this offer by invoking the "agree" *Action*. Thus, this *Negotiation* instance changes its state to "acceptable". If the doer in this negotiation step is the customer and no acceptable counter-offers has been delivered by the provider in the past, the customer is able to submit a final counter-offer and to invoke the "agree" *Action*. In particular, if the doer is the provider and the state of a *Negotiation* instance is "acceptable", the provider has to decide to accept this agreement offer or to reject it. This allows to force a faster decision if both parties fail to agree. If the *Negotiation* instance is in the "acceptable" state, the customer as well as the provider are able to invoke the "reject" *Action*. This will abort the negotiation process. If the customer has agreed and the *Negotiation* instance is in the "acceptable" state, the provider is able to agree, too. Therefore, the provider has to invoke the "sign" *Action* which changes the state of the *Negotiation* instance to "accepted". With this step an obligated *Agreement* instance is created which an *AgreementLink* points to. The *Negotiation* instance is in any case or any state only available until expiration date.

Agreement

The *Agreement* type represents a generic agreement resource that has either been negotiated with an instance of *Negotiation* or is based on an exposed *Offer* instance. The *Agreement* inherits the *Resource* base type defined in OCCI Core Model [64]. It is assigned to the *Kind* instance *http://schemas.cit.tu-berlin.de/occi/sla#agreement* which has to be exposed by each instance of *Agreement*.

TABLE 5.13.: *Attributes* defined for the *Agreement* type

Attribute	Type	Multiplicity	Mutability	Description
occi.agreement .provider	String	0...1	Mutable	The domain that identifies the provider.
occi.agreement .customer	String	0...1	Mutable	The JID that identifies the customer.
occi.agreement .customersignature	Signature	0...1	Mutable	The customer's signature.
occi.agreement .providersignature	Signature	0...1	Mutable	The provider's signature.
occi.agreement .agreedat	Time (ISO8601)	0...1	Mutable	The date time at which the provider and the customer agreed.
occi.agreement .agreedfrom	Time (ISO8601)	0...1	Mutable	The date time when the agreement starts to be effective.
occi.agreement .agreeduntil	Time (ISO8601)	0...1	Mutable	The date time when the agreement seizes to be effective.
occi.agreement .state	Enum {pending, observed, suspended}	1	Immutable	Current state of the agreement instance.

Table 5.13 describes the attributes defined by the *Agreement* through its *Kind* instance and should be exposed by an instance of *Agreement*. The state of an *Agreement* instance indicates whether the SLA is "observed" or "suspended". These states must be exposed by an instance of an *Agreement* type of the SLA model implementation. Figure 5.20 illustrates the state diagram for an *Agreement* instance.

An *Agreement* instance has to be in the state "pending" after instantiation. The agreement has to persist in this state until the first service is instantiated and the first term *Link* instance is configured with a valid target. In general, an instance of *Offer* or *Negotiation* may have

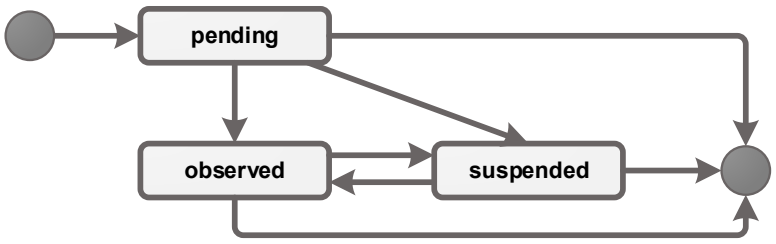

FIGURE 5.20.: State diagram for an *Agreement* instance

several *ServiceReference*, *ServiceDescriptionTerm*, and *GuaranteeTerm* links, but all of them do not point to a specific target. The term *Link* instances are not configured before they are instantiated in an *Agreement* instance and the appropriated service is up and running. If one term *Link* instances points to an actual service, the agreement is activated and changes its state to "observed". If all service instances are canceled and no term *Link* instances are longer available, the *Agreement* instance has to switch into the state "suspended". During its life time, an *Agreement* instance can only switch between these two states.

NegotiationLink

In order to associate an *Offer* instance with a *Negotiation* collection that the customer can use to instantiate a *Negotiation* instance related to a particular *Offer* instance. The *NegotiationLink* inherits the *Link* base type defined in OCCI Core Model [64] and points to a *Negotiation* collection hosted at the exchange of the provider of the service offering. It is assigned to the *Link* instance *http://schemas.cit.tu-berlin.de/occi/sla#negotiationlink* which has to be exposed by each instance of *NegotiationLink*.

AgreementLink

In order to associate a successfully finished *Negotiation* instance with an *Agreement* instance that has been created after both parties have agreed on the terms and conditions, the *AgreementLink* link is introduced. The *AgreementLink* inherits the *Link* base type defined in OCCI Core Model [64] and points to an existing *Agreement* instance in order to provide both parties with a link to the instantiated agreement. It is assigned to the *Link* instance *http://schemas.cit.tu-berlin.de/occi/sla#agreementlink* which has to be exposed by each instance of *AgreementLink*.

ServiceDescriptionTerm Link

The *ServiceDescriptionTerm* type allows to express a set of service terms and evaluates their compliance. The *ServiceDescriptionTerm* inherits the *Link* base type defined in OCCI Core Model [64] and points to a service that is evaluated. Furthermore, it not only links to and describes the service, but it also evaluates the target in an assessment interval based on the attributes that describes the service within this link. It is assigned to the *Link* instance *http://schemas.cit.tu-berlin.de/occi/sla#servicedescriptionterm* which has to be exposed by each instance of *ServiceDescription*.

TABLE 5.14.: *Attributes* defined by the *ServiceDescriptionTerm* link

Attribute	Type	Multiplicity	Mutability	Description
occi.servicedescriptionterm .termlist	Map<attribute, value>	0...1	Immutable	Service Description Terms that the instance target have to fulfill.
occi.servicedescriptionterm .assessmentinterval	Integer	1	Immutable	Assessment interval to check the SDT compliance in seconds.
occi.servicedescriptionterm .state	Enum {undefined, violated, fulfilled}	1	Immutable	The current state of the term.

Table 5.14 describes the attributes defined by the *ServiceDescriptionTerm* through its *Link* instance and should be exposed by an instance of the *ServiceDescriptionTerm*. The state of a *ServiceDescriptionTerm* instance indicates whether the service terms are “fulfilled” or “violated”. These states must be exposed by an instance of a *ServiceDescriptionTerm* type of the SLA model implementation. Figure 5.21 illustrates the state diagram for a *ServiceDescriptionTerm* instance.

A *ServiceDescriptionTerm Link* instance has to be in the state “undefined” after instantiation. The service term has to persist in this state until the first assessment has been performed. Afterwards the state of the *ServiceDescriptionTerm* instance has to be switched into the state “fulfilled” or “violated”. During its life time, a *ServiceDescriptionTerm* instance can only switch between these two states.

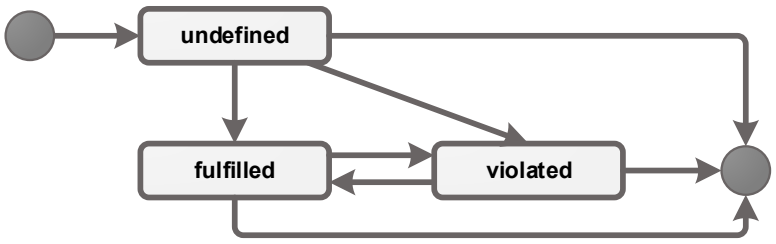

FIGURE 5.21.: State diagram for a *ServiceDescriptionTerm* link

GuaranteeTerm Link

The *GuaranteeTerm* type allows to express a guarantee term and evaluates its compliance. The *GuaranteeTerm* inherits the *Link* base type defined in OCCI Core Model [64] and points to a service that is evaluated. Furthermore, it not only links to and describes the service, but it also evaluates the target based on the attributes that describes the service within this link. It is assigned to the *Link* instance *http://schemas.cit.tu-berlin.de/occi/sla#guaranteeterm* which has to be exposed by each instance of *GuaranteeTerm*.

TABLE 5.15.: *Attributes* defined by the *GuaranteeTerm* link

Attribute	Type	Multi-plicity	Mutability	Description
occi.guaranteeterm .sensor	String	1	Immutable	The sensor from which the link retrieves its measurements, i.e. the object.
occi.guaranteeterm .slo	String	1	Immutable	The service level objective.
occi.guaranteeterm .state	Enum {undefined, violated, fulfilled}	1	Immutable	The current state of the term.
occi.guaranteeterm .relationaloperator	Enum {LESS_THAN, LESS_THAN_OR_EQUAL_TO, EQUAL_TO, NOT_EQUAL_TO, GREATER_THAN_OR_EQUAL_TO, GREATER_THAN}	1	Immutable	The relational operator to describe the guarantee.

Table 5.15 describes the attributes defined by the *GuaranteeTerm* through its *Link* instance and should be exposed by an instance of the *GuaranteeTerm*. The state of a *GuaranteeTerm* instance indicates whether the guarantee is "fulfilled" or "violated". These states must be exposed by an instance of a *GuaranteeTerm* type of the SLA model implementation. Figure 5.22 illustrates the state diagram for a *GuaranteeTerm* instance.

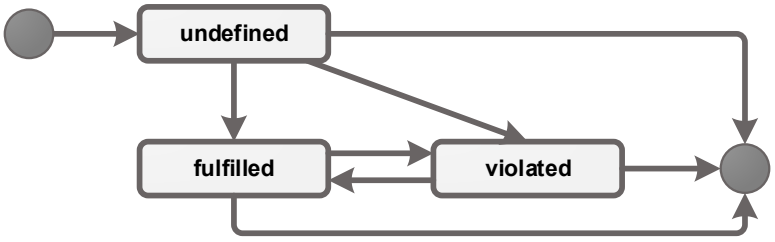

FIGURE 5.22.: State diagram for an *GuaranteeTerm* link

A *GuaranteeTerm Link* instance has to be in the state "undefined" after instantiation. The guarantee term has to persist in this state until the first assessment is possible. Afterwards the state of a *GuaranteeTerm* instance has to be switched into the state "fulfilled" or "violated". During its life time, a *GuaranteeTerm* instance can only switch between these two states.

ServiceReference Link

In order to associate an *Agreement* instance with an existing service resource instance, the *ServiceReference* link is introduced. The *ServiceReference* inherits the *Link* base type defined in OCCI Core Model [64] and points to a service in order to link instantiated services with appropriated agreements. It is assigned to the *Link* instance *http://schemas.cit.tu-berlin.de/occi/sla #servicereference* which has to be exposed by each instance of *ServiceReference*.

BusinessValue Mixin

A *BusinessValue* mixin introduces attributes listed in table 5.16 which are required for describing the price of a service or a penalty that has to be paid in case of agreement violations. This *BusinessValue* mixin is assigned the schema *http://schemas.cit.tu-berlin.de/occi/sla#* and the "term" value *businessvalue*.

This mixin can be related to any other kind, mixin, or link and thus *applies* to *http://schemas .ogf.org/occi#category*.

TABLE 5.16.: *Attributes* defined by the *BusinessValue* mixin

Attribute	Type	Multiplicity	Mutability	Description
occi.businessvalue.type	Enum {Reward, Penalty}	1	Mutable	The type of the business value.
occi.businessvalue.price	Double	1	Mutable	The price for the business value.
occi.businessvalue.currency	String	1	Mutable	The currency of the price.
occi.businessvalue.billingincrements	String	1	Mutable	The billing increments of the price, e.g. per GB, per hour, per second, etc.

5.6.3. Event Processing Model

While the previous sections defined kinds, mixins, and links for monitoring and SLA management, this sub-section defines mixins which can be applied to any resource or link definition and complements their application. These mixins can be related to any other kind, mixin, or link and thus *applies* to *http://schemas.ogf.org/occi#category*.

EventLog Mixin

If monitored measurements are transferred between many independent entities, the *EventLog* mixin is used to identify a category of events. The *EventLog* mixin introduces an attribute listed in table 5.17 that is required for separating event streams sent according to the XEP-0337: Event Logging over XMPP protocol extension [158]. This *EventLog* mixin is assigned the schema *http://schemas.cit.tu-berlin.de/occi/cep#* and the "term" value *eventlog*.

TABLE 5.17.: *Attributes* defined by the *EventLog* mixin

Attribute	Type	Multiplicity	Mutability	Description
occi.eventlog.eventid	String	1	Immutable	The event id for this particular event stream.

Aggregation Mixin

An *Aggregation* mixin introduces an attribute listed in table 5.18 which is required to expose an aggregation operation. This operation should have to be applied in order to aggregate mea-

surements of a single source or from different sources. The *Aggregation* mixin is assigned the schema *http://schemas.cit. tu-berlin.de/occi/cep#* and the "term" value *aggregation*.

TABLE 5.18.: *Attributes* defined by the *Aggregation* mixin

Attribute	Type	Multi-plicity	Mutability	Description
occi.aggregation.operation	Enum {min, max, sum, avg}	1	Immutable	The aggregation operator to specify how measurements are aggregated.

LengthWindowMetric Mixin

A *LengthWindowMetric* mixin introduces attributes listed in table 5.19 which are required for describing a metric where the calculation of measurements is based on the number of events. This *LengthWindowMetric* mixin is assigned the schema *http://schemas.cit.tu-berlin.de/occi/cep#* and the "term" value *lengthwindowmetric*.

TABLE 5.19.: *Attributes* defined by the *LengthWindowMetric* mixin

Attribute	Type	Multi-plicity	Mutability	Description
occi.metric.windowtype	Enum {SlidingWindow, BatchWindow}	1	Mutable	The type of the window, i.e. either a periodically assessed window (BatchWindow) or a continuously assessed window (SlidingWindow).
occi.metric.windowlength	Integer	1	Mutable	The length of the window, i.e. the number of events that this window assesses.

TimeWindowMetric Mixin

A *TimeWindowMetric* mixin introduces attributes listed in table 5.20 which are required for describing a metric where the calculation of measurements is based on a time window in which events occur. This *TimeWindowMetric* mixin is assigned the schema *http://schemas.cit.tu-berlin.de/occi/cep#* and the "term" value *timewindowmetric*.

TABLE 5.20.: *Attributes* defined by the *TimeWindowMetric* mixin

Attribute	Type	Multiplicity	Mutability	Description
occi.metric.windowtype	Enum {SlidingWindow, BatchWindow}	1	Mutable	The type of the window, i.e. either a periodically assessed window (BatchWindow) or a continuously assessed window (SlidingWindow).
occi.metric.windowduration	Integer	1	Mutable	The time length of the window, i.e. the duration that this window assesses.
occi.metric.durationunit	Enum {years, months, weeks, days, hours, minutes, seconds, milliseconds}	1	Mutable	The unit in which time length of the window is specified.

6. Intercloud Prototyping and Evaluation

Contents

This chapter describes the prototypical implementation of the features described in the chapter before and its evaluation in terms of efficiency and scalability. The requirements that such a software architecture should fulfill are: efficiency, scalability, flexibility, and reusability. In order to achieve these sustainability requirements, a modularly architecture has been developed as depicted in the dependencies tree diagram illustrated in figure 6.1. The modules depicted in this figure 6.1 provide the following functionalities:

xmpp-rest: The xmpp-rest module implements the functionality described in section 5.3.

xmpp-occi: The xmpp-occi module implements the functionality described in section 5.4.

smack: Smack is an open source XMPP client library developed by the Igniterealtime open source community [159].

xmpp-core: The xmpp-core module is the basis module for all components. It is a fork of Whack [160] that implements the XEP-0114: Jabber Component Protocol extension.

xmpp-component: The xmpp-component module combines the xmpp-core functionalities with the xmpp-rest and the xmpp-occi resource deployment. It implements the request/ response model of containers and the particular marshaller for resource access.

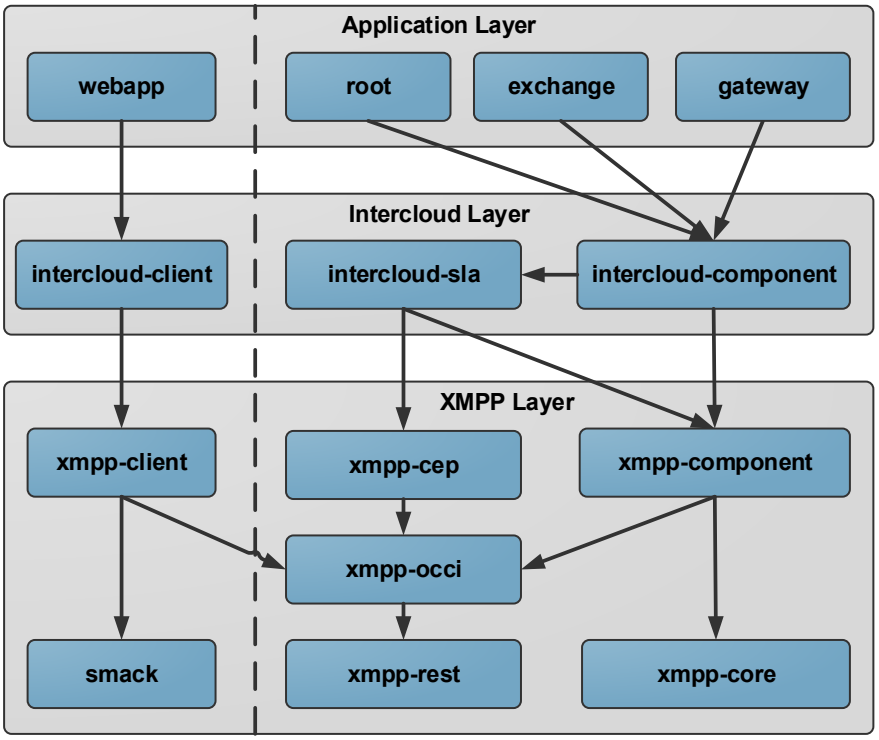

FIGURE 6.1.: Module dependencies of the intercloud prototype

xmpp-client: The xmpp-client module implements functionalities for xmpp-rest and xmpp-occi IQ translation based on the Smack communication model. Additionally, the client stub generation is supported in this module which do not require any recompilation of the client's code base.

xmpp-cep: The xmpp-cep module implements the complex event processor used for monitoring and SLA evaluation. It defines a set of predefined event types and mixins as decribed in section 5.6.1 and 5.6.3. It is based on the Esper open source event stream analysis and event correlation engine.

intercloud-sla: The intercloud-sla module implements the intercloud SLA-Engine described in section 5.5 and 5.6.2.

intercloud-client: The intercloud-client module provides intercloud specific fuctionalities for clients.

intercloud-component: The intercloud-component module provides intercloud specific fuctionalities for the intercloud topology elements including discovery and so on.

root: The root module implements the *Intercloud Root* topology component as an executable jar archive.

exchange: The exchange module implements the *Intercloud Exchange* topology component as an executable jar archive.

gateway: The gateway module implements the *Intercloud Gateway* topology component as an executable jar archive. The current version available at *github* supports OpenStack's API model.

webapp: The web application module is based on Apache Wicket [161] and Bootstrap [162]. It is a HTTP-based web application that renders the Intercloud REST with XMPP and OCCI XML Rendering specification in order to provide users a human-friendly access and management platform for the intercloud network. Furthermore, it prints each request and responds with syntax highlighting building thus a well-designed debugging tool as well.

The modules described above and specified in the previous chapter build the protocol stack depicted in figure 6.2. Here, the XMPP Core specification is the foundation. Its basic functionalities are extended with the XEP-0030 and the XEP-0114 XMPP extension. Build on that, the REST with XMPP specification is added on top. Then the OCCI rendering is applied and allows for applying all existing OCCI Extension Models as well as novel extensions specified in this thesis and future extensions which will be specified in the intercloud initiative.

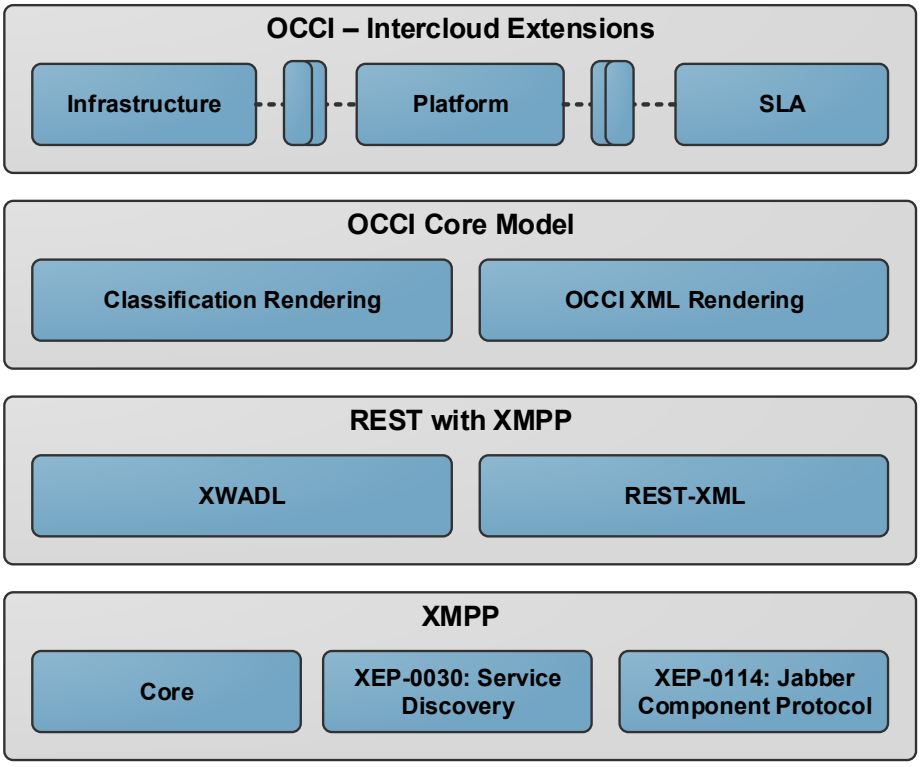

FIGURE 6.2.: Protocol stack of the intercloud prototype

6.1. Implementation

This section presents the four essential design concepts which are not specified or described in the previous sections but are implemented in the intercloud prototype. However, these design concepts are partially reflected in the specifications in the previous chapter, but are not further explained in detail. Thus, this section not only presents the developed implementation concepts, but also illustrates use cases to which the extension specifications can be applied.

6.1.1. Communication Pattern

The communication pattern implemented for an XMPP component is package-driven according to the event-driven programming paradigm. Here, the flow of the program is determined by received packages that are sent from any entity to the particular XMPP component. For this purpose, the XMPP component implements a thread executor with a thread pool and a task queue that makes use of the `java.util.concurrent.ThreadPoolExecutor`. Each package that is received by a component is wrapped in a `PacketProcessor` task and processed by the thread executor. When a package has to be processed, this package is analyzed and handled according to its nature: `<message/ >`, `<presence/ >`, and `<iq/ >`. While message and presence stanzas are connection-independent, IQ stanzas follow a structured request-response mechanism. The REST with XMPP protocol extension presented in section 5.3 uses IQ stanza for exchanging XWADL and REST-XML documents. Thus, all IQ stanzas of type `get` or `set` require an appropriated response of type `result` or `error`.

This pattern is ideal in cases where no further interaction with other intercloud topology elements is required. However, in the approach presented before interactions between *Intercloud Gateways*, *Intercloud Exchanges*, and *Intercloud Roots* are necessary. Therefore, an additional socket-based communication pattern has been developed. This socket-based communication model allows any resource instance to create a socket which can be used to send messages, to transfer *LogEvents*, or to request XWADL documents and to invoke appropriated remote methods. The sockets cannot be created without any relationship to the XMPP component communication, therefore, a singleton socket manager has to be used to create a socket and provide the required mapping for IQ requests and responses. While the package-driven nature of XMPP components expect IQ stanza of type `get` or `set`, the *Marshaller* identifies response packages via the IQ stanza type `result` or `error`. This packages are passed to the socket manager that maps response IQ stanza to sockets by their IQ-IDs.

The socket supports two types of methods: synchronous and asynchronous package processing. Presence and message related methods are always processed asynchronously, but IQ related methods are not. The IQ related methods such as `requestXWADL` or `invokeRestXML` are synchronous and block the resource instance thread until the IQ response has been received.

However, a result listener can be used in order to invoke a REST-XML method asynchronously. This connection pattern concept not only allow to process packages extremely loosely coupled and well distributed, but also enables sequentially controlled processes between intercloud topology components.

6.1.2. Service Discovery

When a new intercloud component joins the intercloud network, the new component may synchronize its data sets with other intercloud components. For this purpose, the component has to discover other existing intercloud components in order to interact with them. Furthermore, if an *Intercloud Exchange* wants to publish a service offering in the service catalog hosted at an *Intercloud Root*, the component needs its JID in order to create a socket for this communication. For this discovery mechanism the XEP-0030: Service Discovery [132] protocol extension is used. This extension defines two types of discovery: *disco#info* and *disco#item*. While the *disco#item* discovery allows for retrieving information about a target entity's identity, the *disco#info* discovery allows for retrieving the features offered and protocols supported by the target entity.

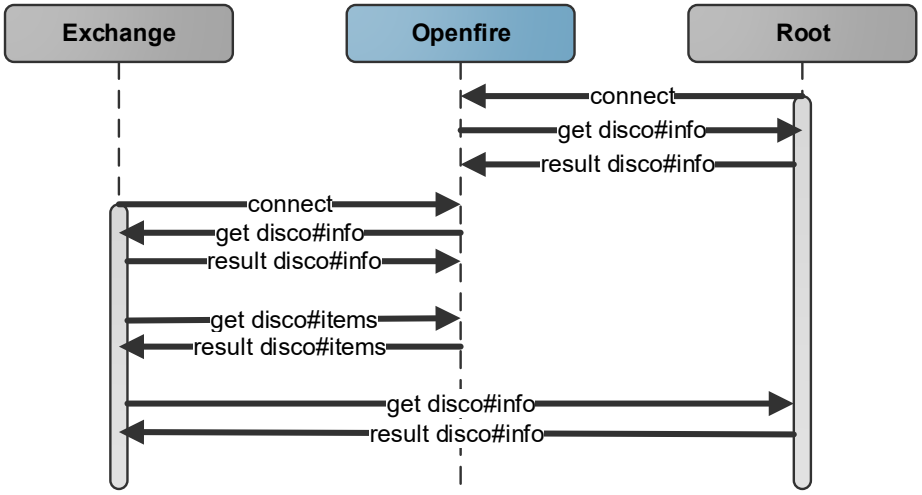

FIGURE 6.3.: Root discovery by an *Intercloud Exchange*

In the following, the discovery of intercloud topology components is presented. Figure 6.3 depicts a sequence diagram of the discovery mechanism for components which are connected to the same XMPP server instance. Here, a *disco#item* is requested that lists all connected XMPP components and services. The prototypical implementation identifies intercloud topology elements by their names which are:

- `RootComponentName = 'Intercloud Root'`
- `ExchangeComponentName = 'Intercloud Exchange'`
- `GatewayComponentName = 'Intercloud Gateway'`

If the *Intercloud Root* component is hosted by an external XMPP server in another domain, a static root server whitelist is used as a basis to find the JID of a corresponding *Intercloud Root*. The last step of the discovery process (i.e. *disco#info* for retrieving the features) is then executed in any case in order to check the protocol compliance.

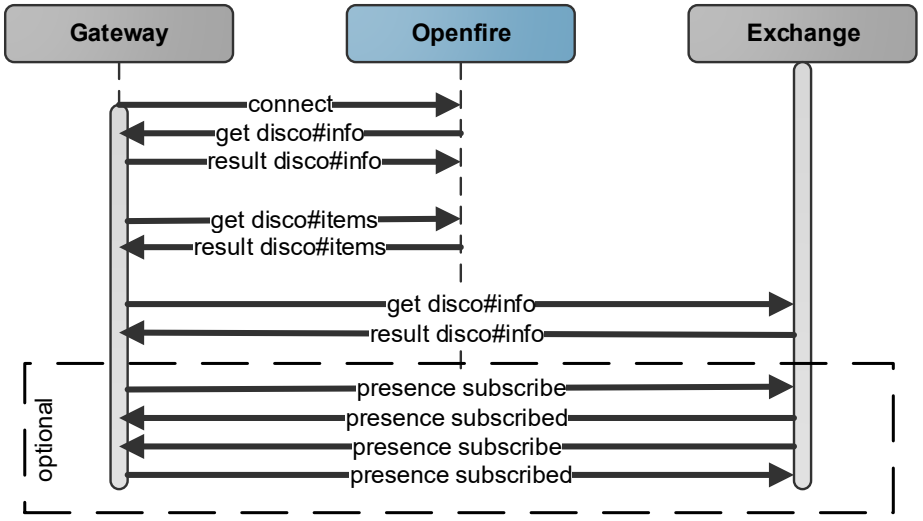

FIGURE 6.4.: Exchange discovery by an *Intercloud Gateway*

If an *Intercloud Gateway* hosts monitoring services as described in section 5.6.1 and 5.6.3, these *Intercloud Gateway's* sensor resources have individual link configurations which specify the JID of *Intercloud Exchanges* to which the monitoring measurements should be sent. In this case the sensor resource instance that wants to create a socket for this purpose does not have to discover the JID of this *Intercloud Exchange* but has to discover the features provided by the counter-party. However, assuming that the *Intercloud Gateway* starts-up the first time and want to advertise its service supply to other *Intercloud Exchanges* of the same domain in order to enable service life-cycle management, the *Intercloud Gateway* has to discover all available *Intercloud Exchanges* and to advertise its availability. This discover procedure is depicted in figure 6.4. Here, the use of presence subscription is considerable in order to exchange presence information between *Intercloud Gateways* and *Intercloud Exchanges*.

6.1.3. Service Catalog

The service catalog is a structured set of service offerings. Depending on the location of the service catalog, this service catalog is either a local service repository hosted at an *Intercloud Exchange* of a particular provider or a global service repository hosted at *Intercloud Roots* around the world. While the service repository of an *Intercloud Exchange* has to follow provider specific requirements, the service repository hosted on *Intercloud Roots* has to be extremely decoupled, distributed, efficient, and scalable. Therefore, the service catalog is partitioned similar to the Domain Name System (DNS) partitioning. Thus, each *Intercloud Root* provides a subset of the global service repository and links to other *Intercloud Roots* for resolving a query. In particular, the service catalog is structured as depicted in figure 6.5. The hierarchy is built in the first line according to the categories and in the second line according to the geographical location. The hierarchical structure follows the ISO 3166-2 country and subdivision encoding.

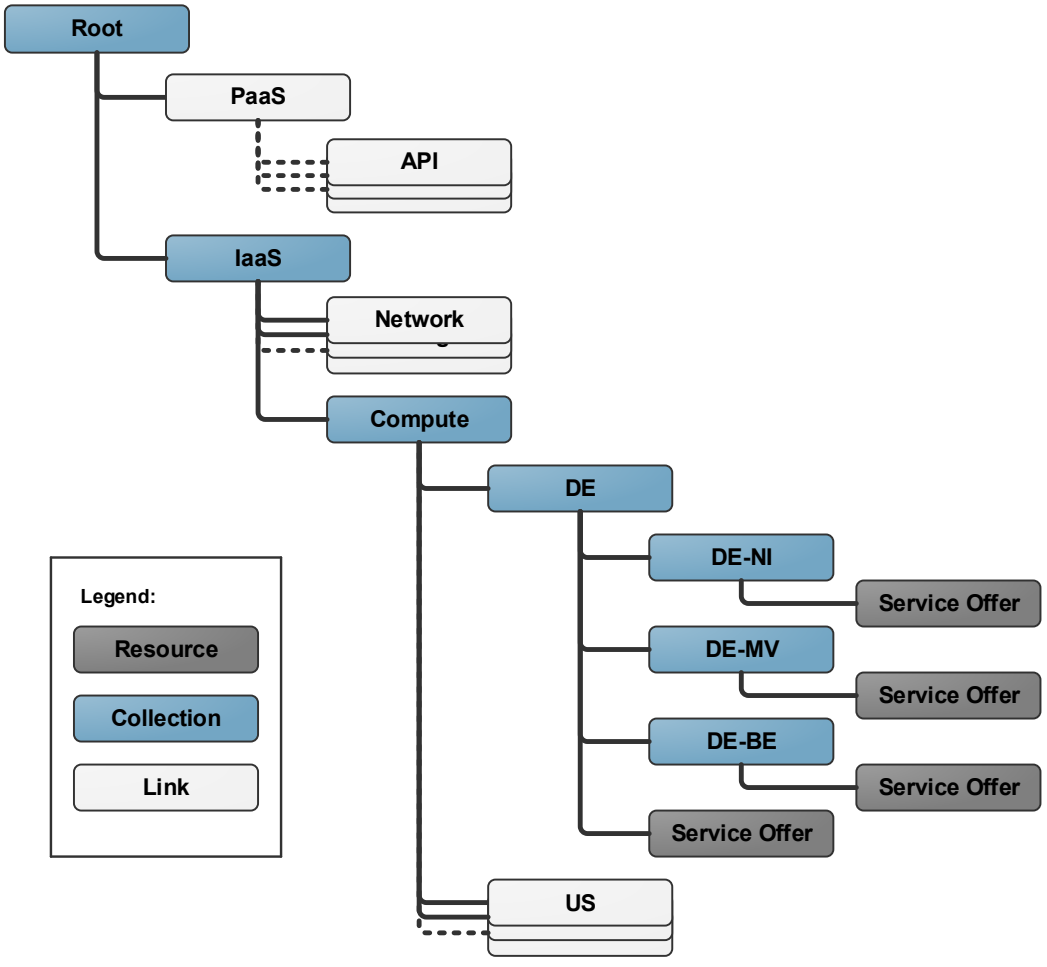

FIGURE 6.5.: Root's service catalog hierarchy

Search queries and look-ups are resolved similar to DNS as well: iterative, recursive, and authoritative. Here, the resolver is the *Intercloud Exchange* component of a particular provider's domain which the requester belongs to. This proves a scalable, fail safe, and flexible architecture, because caching mechanisms with time to live (TTL) tags can be applied. Of course, this may implicate data inconsistency but is negligible, because the service is at least provisioned based on an agreement that has to be created with the authoritative *Intercloud Exchange*.

Considering a unique service catalog structure as illustrated in figure 6.5, the publication of service offerings is straightforward. In fact, when a new service offer resource instance is created in the local service repository of an *Intercloud Exchange*, this component just has to synchronize its hierarchically structured data set with the *Intercloud Root* which this *Intercloud Exchange* belongs to. Since all the functionalities and behavior described in this section are approaches which are completely or partially provided by the prototypical implementation, but are still not defined in the IEEE Intercloud P2302 specification [146], the techniques and methods may change before the document will be standardized.

6.1.4. Complex Event Processing

The Complex Event Processor (CEP) used in this prototypical implementation is based on the Esper open source event stream analysis and event correlation engine [74]. It enables to detect situations in event series when event specific conditions occur. These conditions are formulated as statements that are built with expressions according to the Event Processing Language (EPL). The EPL allows to express filtering, aggregation, complex causality checks, and joins of multiple event series which are processed for pre-defined length-based and time-based batching and sliding windows. The event representation supported by the CEP of the prototypical implementation is either one of the implemented *LogEvent* types (i.e. Plain Old Java Objects) or XML documents according to the XEP-0337: Event Logging over XMPP protocol extension schema [158]. While the implemented *LogEvent* types enable to aggregate numerical data (e.g. Double, Integer, etc.) the XML-based events representation only allows to apply String-based filters.

The CEP of this concept can only be applied to the *Intercloud Exchange* or the *Intercloud Gateway*. It is used to process monitoring measurements, to aggregate the measurements in a *Collector* resource instance, and to filter the measurements at an *Intercloud Gateway*. On the other hand the CEP is used to aggregate, to filter, and to evaluate guarantee terms of an SLA at an *Intercloud Exchange*. An example of how event streams are processed, filtered, aggregated, and passed over the XMPP network is illustrated in figure 6.6.

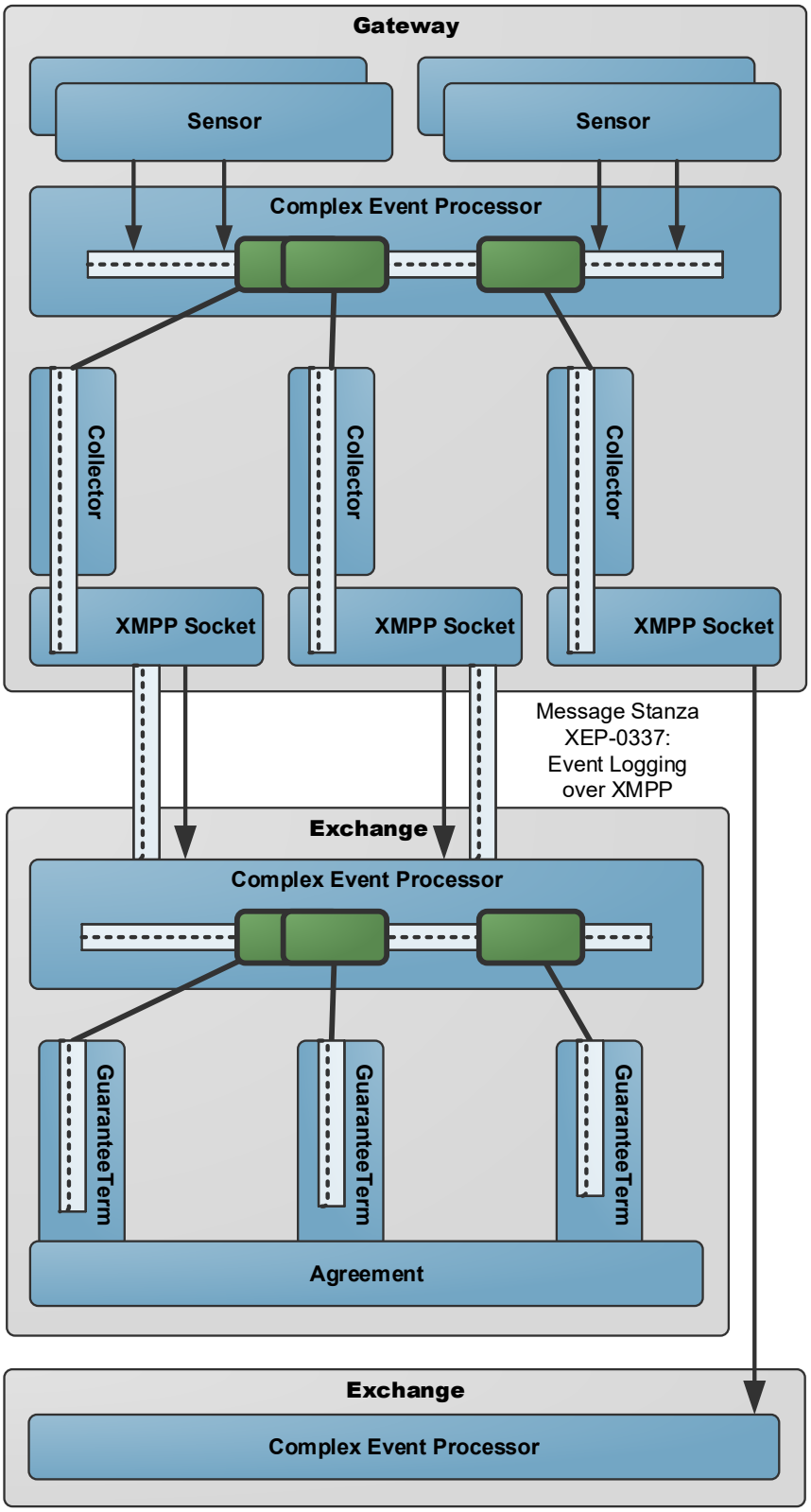

FIGURE 6.6.: Intercloud's complex event processing

Figure 6.6 illustrates an *Intercloud Gateway* where four *Sensor* resources are instantiated. All these sensors continuously add their measurements as events to the CEP. Three *Collector* link instances aggregate measurements based on a statement configured through the mixins as described in section 5.6.1 and 5.6.3. The targets of the *Collector* links are in this example two *Intercloud Exchanges*. The values are transferred from *Collector* instances to the remote CEP at *Intercloud Exchanges* as *EventLogs* in a message stanza. On the other side, one *Agreement* resource instance is available. This *Agreement* instance has three *GuaranteeTerm* links which evaluate their guarantee terms based on statements configured through the mixins as described in section 5.6. In contrast to the aggregation or filter statement of a *Collector* link, a *GuaranteeTerm* link has two statements: One for fulfilled guarantee term states and one for violated guarantee term states.

The number of event streams established in a CEP depends on the number of different event representation types. The reason for having a staged event processing is firstly the number of events that are transferred over the network and secondly the amount of events that have to be kept in memory for processing. Therefore, this staged approach aims at reaching a uniform distribution of computation and memory usage over the intercloud topology elements.

6.2. Evaluation

This section evaluates the presented intercloud approaches and their implementation. The goal of this evaluation is to present the scalability of this implementation and the functionality of the specified protocols. Furthermore, this evaluation also tests the behavior of the system depending on the traffic, the number of providers and customers, the benefits of the staged approach, and the maximum number of SLA evaluations for a specific hardware and test configuration. For this purpose two tests were performed: a Use Case Testing and a Load Testing. While the first test assesses hypothesis defined on experience, the second test is to identify the maximum operating capacity for a specific hardware and test configuration.

Both tests follow the same basic test procedure:

1. The system is idle for one minute after start-up.
2. After one minute idle, the test client creates a new sensor every ten seconds.
3. The sensors created in this use case implements an *ActiveSensor* mixin instance that samples the availability of a simulated virtual machine every second.
4. Each sensor created in this use case has a simulated *Collector* link instance that performs no aggregations of events, thus each second an event occurs the *Collector* link instance passes this event to the *Interclud Exchange* which it points to.

5. The sensors created in this use case do not start before a new *GuaranteeTerm* link instance is created. Therefore, the test client creates a new *GuaranteeTerm* link instance in the same interval as a *Sensor* is instantiated (i.e. every ten seconds).

6. The guarantee terms created in this use case are configured with the same SLO that defines an availability greater than or equal to 50%.

7. A *TimeWindowMetric* mixin is applied to each *GuaranteeTerm* link instance which defines a sliding time window of 15 minutes.

8. An *Aggregation* mixin is applied to each *GuaranteeTerm* link instance as well and defines to calculate the average of measurements by its aggregation operation.

This test configuration implicates two guarantee term evaluations per sensor per second. Furthermore, the number of *LogEvents* to transfer and the total packet payload per second is increased every ten seconds. In order to create an appropriated test plan, an expected load calculation has been performed and is depicted in figure 6.7. While the number of events to transfer, the number of *LogEvents* to evaluate, and the number of guarantee term evaluations are constantly increased with the same amount, the total package payload varies over time. The reason for this behavior is the creation of *GuaranteeTerm* link instances which are taken into account and occur every ten seconds.

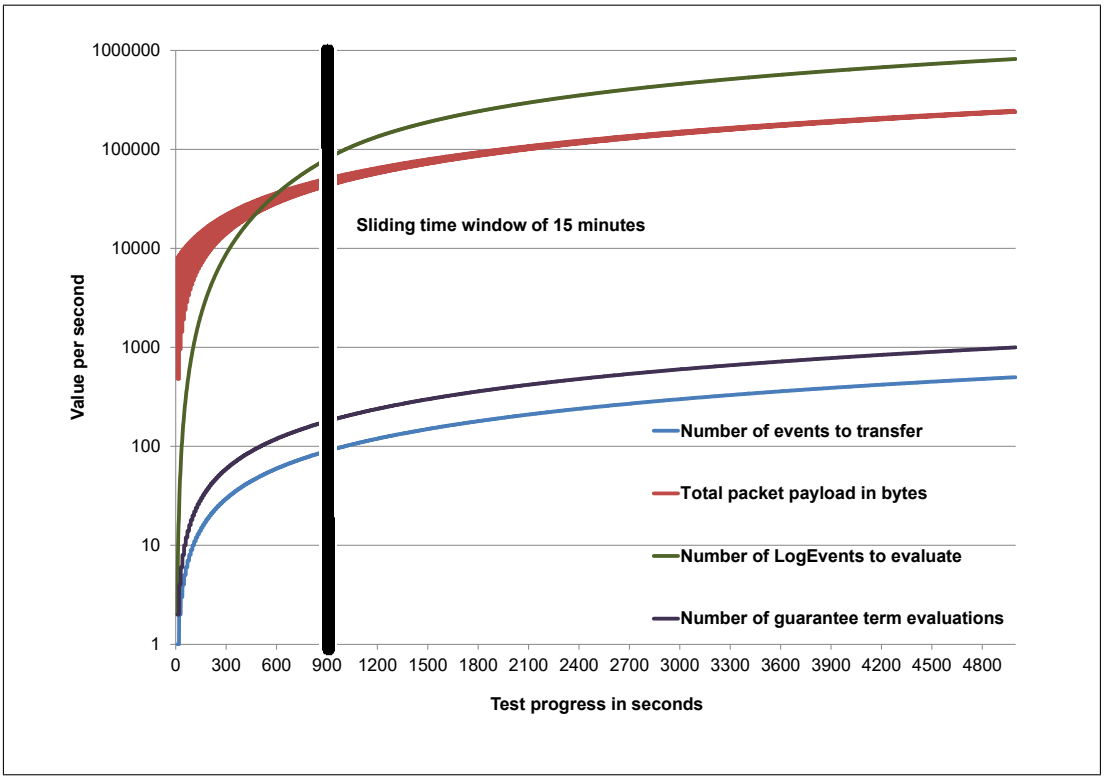

FIGURE 6.7.: Load calculation for test planning

The performance tests used one Mac Mini server and up to ten wally nodes as shown in figure 6.8. The Mac Mini server is equipped with one Intel Core 2 Duo 2.66 GHz CPU and 4 GB DDR3 SDRAM. The nodes are connected via regular Gigabit Ethernet link to the next switch and a TU Berlin internal 10 Gigabit optical fiber in between. The Mac Mini runs Mac OS X El Capitan version 10.11 (Darwin kernel version 15.0.0). The Mac Mini node run Java version 1.8.0_40 (Oracle Java SE Development Kit 8u40). The XMPP server used for the performance tests is an Openfire in version 3.10.0. Both, the Openfire and the *XMPP External Component* are installed on the same physical host. The wally nodes are equipped with a Quadcore Intel Xeon E3-1230 V2 3.30GHz CPU and 16 GB DDR3 SDRAM. The nodes run CentOS with kernel version 3.10.0-229.11.1.el7.x86_64 and Java version 1.8.0_51 (Oracle Java SE Development Kit 8u51).

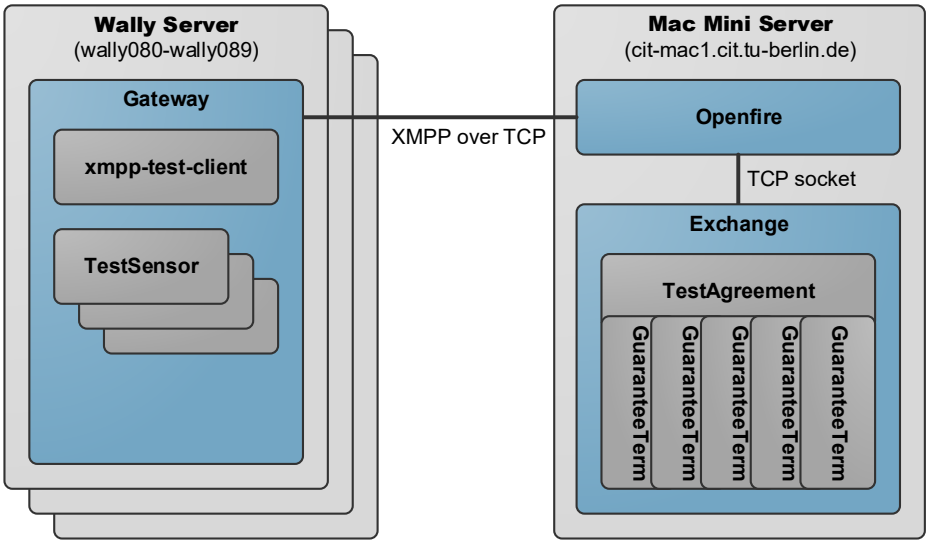

FIGURE 6.8.: Experimental setup for the evaluation

6.2.1. Use Case Testing

The purpose of this evaluation is to prove hypotheses defined by experience as follows:

Hypothesis 1. *If XMPP entities transfer stanzas to a particular intercloud topology element implemented as component, the scalability depends on the XMPP server in between that routes the packages to the particular component.*

Hypothesis 2. *If multiple XMPP entities located on different servers transfer stanzas to a particular intercloud topology element implemented as component, the traffic and throughput is the same as in the case when a single XMPP entity would transfer the same amount of stanzas.*

Hypothesis 3. *If multiple XMPP entities located on different servers with different JIDs invoke REST with XMPP operations on a particular intercloud topology element implemented as component, the performance is the same as in the case when a single XMPP entity would invoke the same amount of operations.*

In order to prove the hypotheses defined before, this evaluation compares a distributed execution of the test procedure with an end-to-end execution. While in the end-to-end evaluation the whole test procedure is executed on a single host (wally080), ten wally nodes (wally080 - wally089) executing a single step for a particular time frame in the distributed evaluation. In particular, wally080 instantiates its *Sensor* instance and the appropriated *GuaranteeTerm* link after one minute, wally081 instantiates its *Sensor* instance and the appropriated *GuaranteeTerm* link after two minutes, and so on. Though the Use Case Testing basically follows the test procedure described before, the interval in which the test clients create *Sensor* instances and appropriated *GuaranteeTerm* links is one minute. Thus, the overall test duration of a single evaluation is 11 minutes.

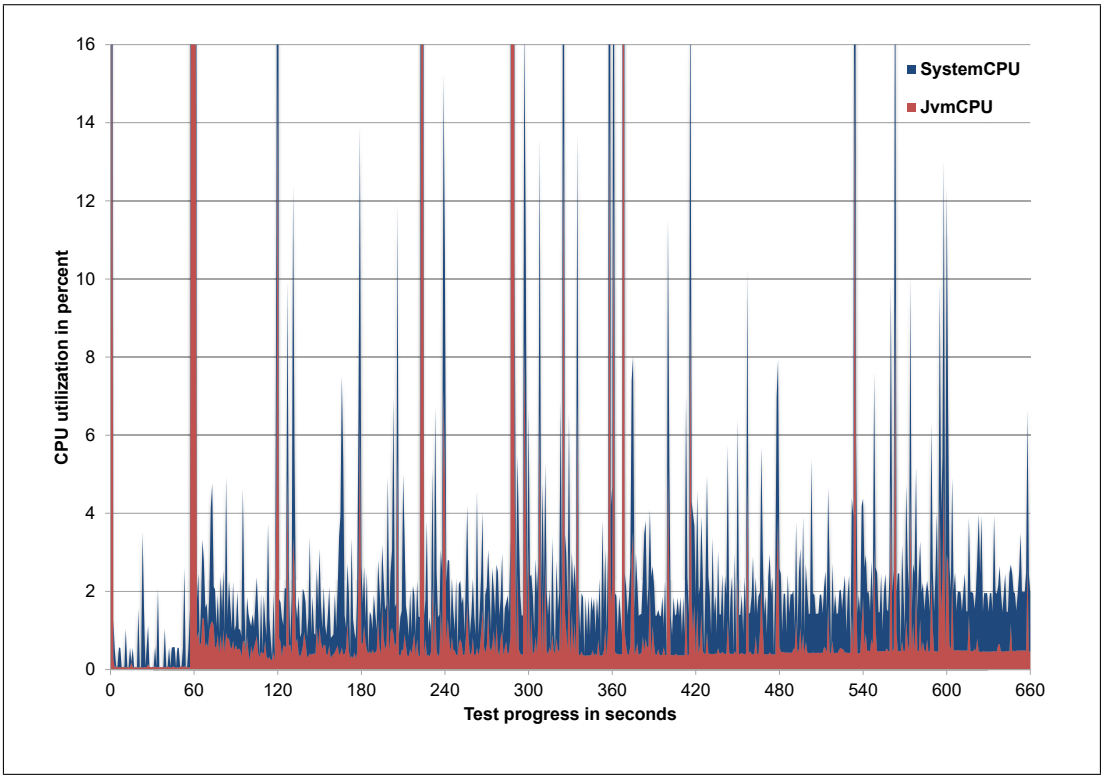

FIGURE 6.9.: CPU load of the distributed test with ten hosts

Figure 6.9 depicts the CPU utilization of the *Intercloud Exchange* to which ten distributed hosts transfer their sensor events. Figure 6.10 depicts the CPU utilization of the *Intercloud Exchange*

to which a single host transfers the same amount of sensor events. In both figures, the creation of *GuaraneteeTerm* links is well visible through the load peaks every minute.

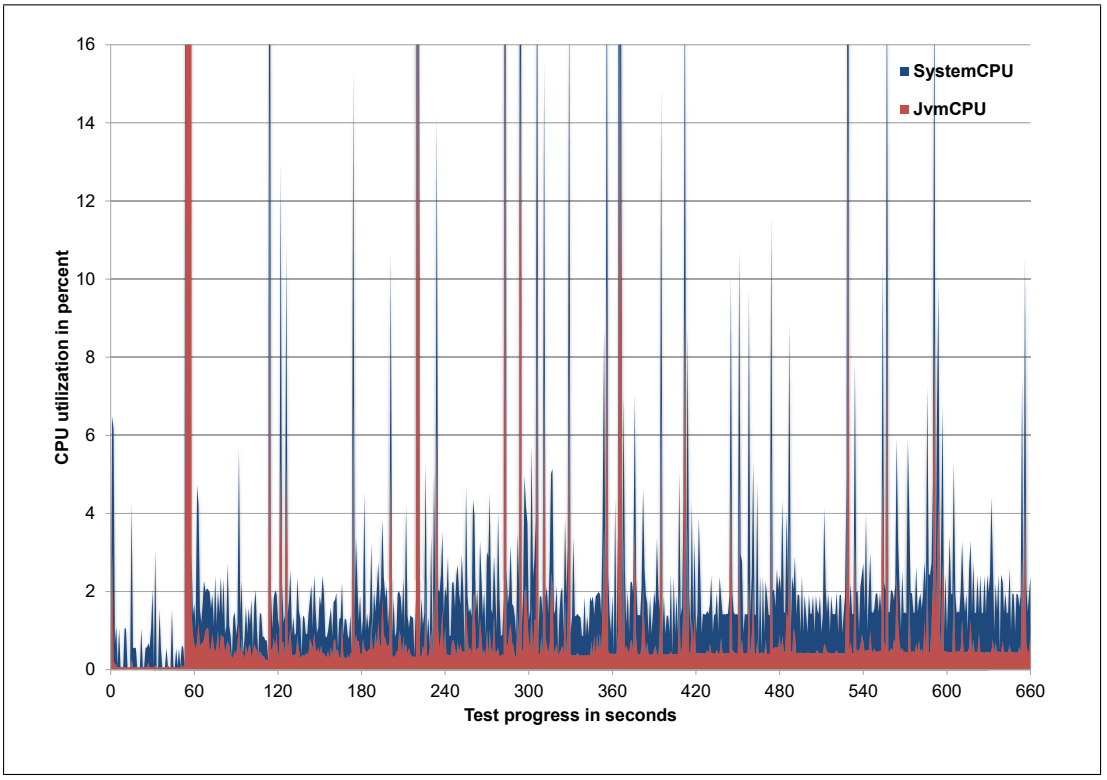

FIGURE 6.10.: CPU load of the end-to-end test with one host

The Openfire reports statistics in an averaged minute scale. Here, the number of packets per minute are the same for the distributed and the end-to-end evaluation. Furthermore, in both evaluations the number of waiting tasks in the queue of the component's `ThreadPool Executor` is on average zero and the average number of executing threads is equal as well. This is very well illustrated by the behavior of the curves in figure 6.11. This figure 6.11 depicts the average CPU utilization per minute and compares the CPU behavior of both evaluations. Besides a not notable difference of 1.1% after 5 minutes test execution, the curves have the same behavior and are correlated.

The payload recording of this evaluation is illustrated in appendix A.6. Figure 6.12 depicts the memory usage of the JVM during test execution. Here, the memory usage of both evaluations are compared. The distributed evaluation allocates after two minutes of test execution more memory as compared to the end-to-end evaluation test run. However, both curves are correlating after eight minutes test execution. Thus, the CPU load and the memory usage behaviors can be considered as correlated. Consequently, hypothesis 2 and 3 are proved.

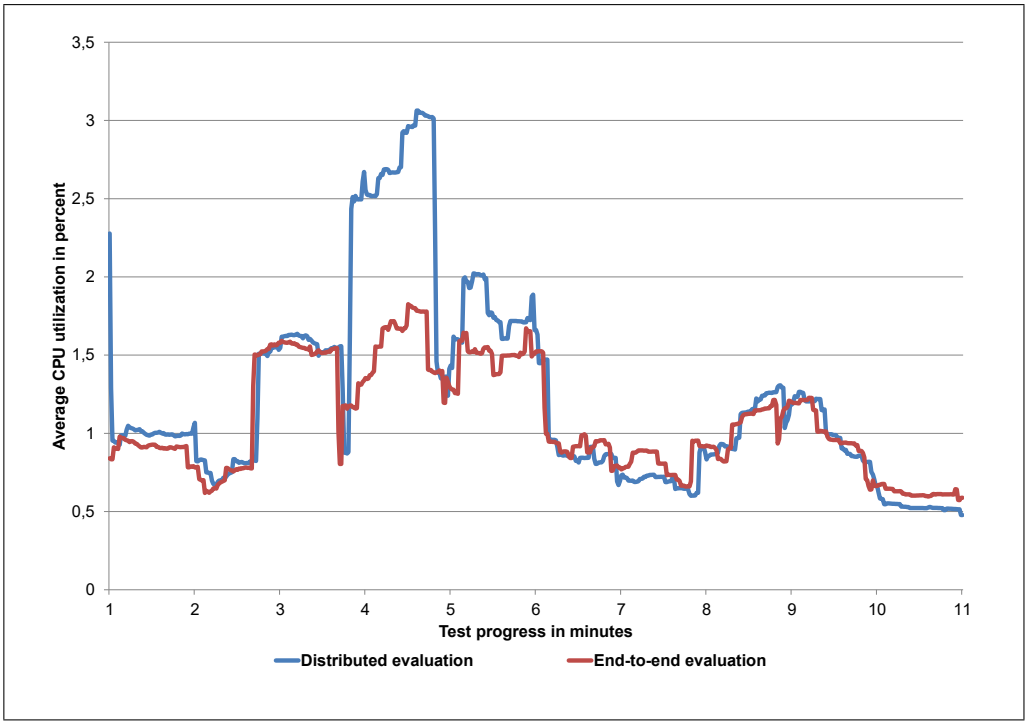

FIGURE 6.11.: Averaged CPU load comparison of both tests

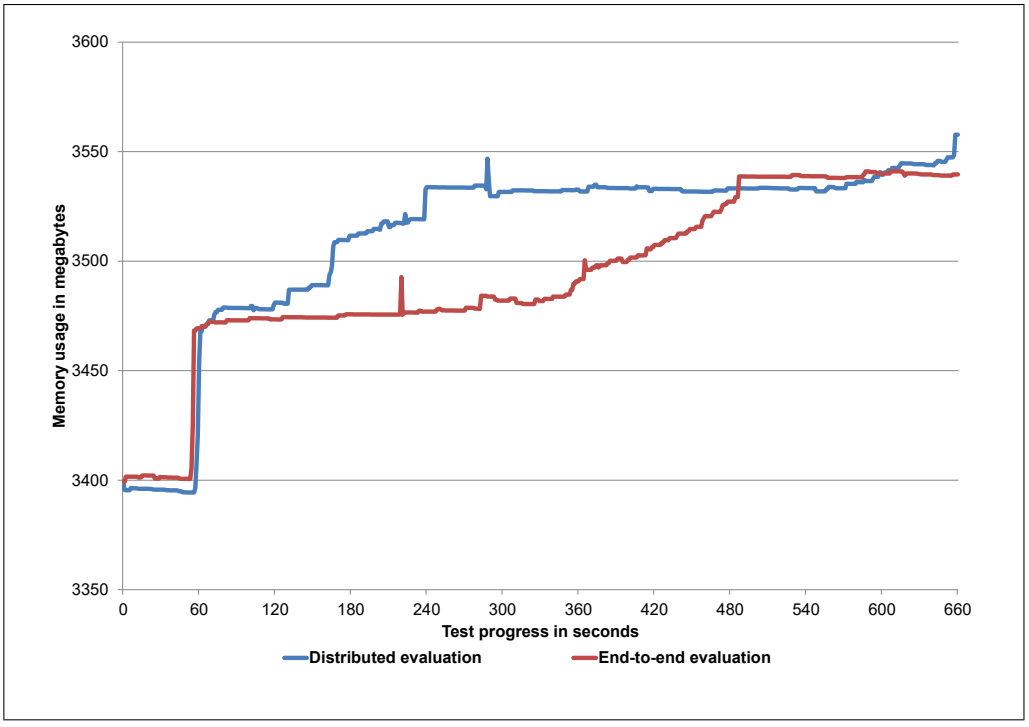

FIGURE 6.12.: Physical memory usage comparison of both tests

While hypothesis 2 and 3 are proved by this evaluation, hypothesis 1 is tried to be proven by the following load testing.

6.2.2. Load Testing

The purpose of this load testing is to generate a growing load in order to identify the maximum operation capacity for a particular hardware and test configuration. The wally nodes obviously provide much more memory and more computing power as the Mac Mini on which the XMPP server and *Intercloud Exchange* component is executed. However, the goal of this evaluation is not to show how many SLA evaluations can be performed on a fat node or on a cluster of high-performance nodes, but to identify the limits that are resulting from the configuration and the dimensioning of evaluation metrics applied to *GuaranteeTerm* link instances at an *Intercloud Exchange*.

The Load Testing follow the basic test procedure described before in the test plan. The results of the previous Use Case Testing show that the performance of a particular intercloud topology element implemented as component is independent from the number and from their distribution. The performance depends on the number of stanzas to process, thus a single host (wally080) executes the test client for this evaluation.

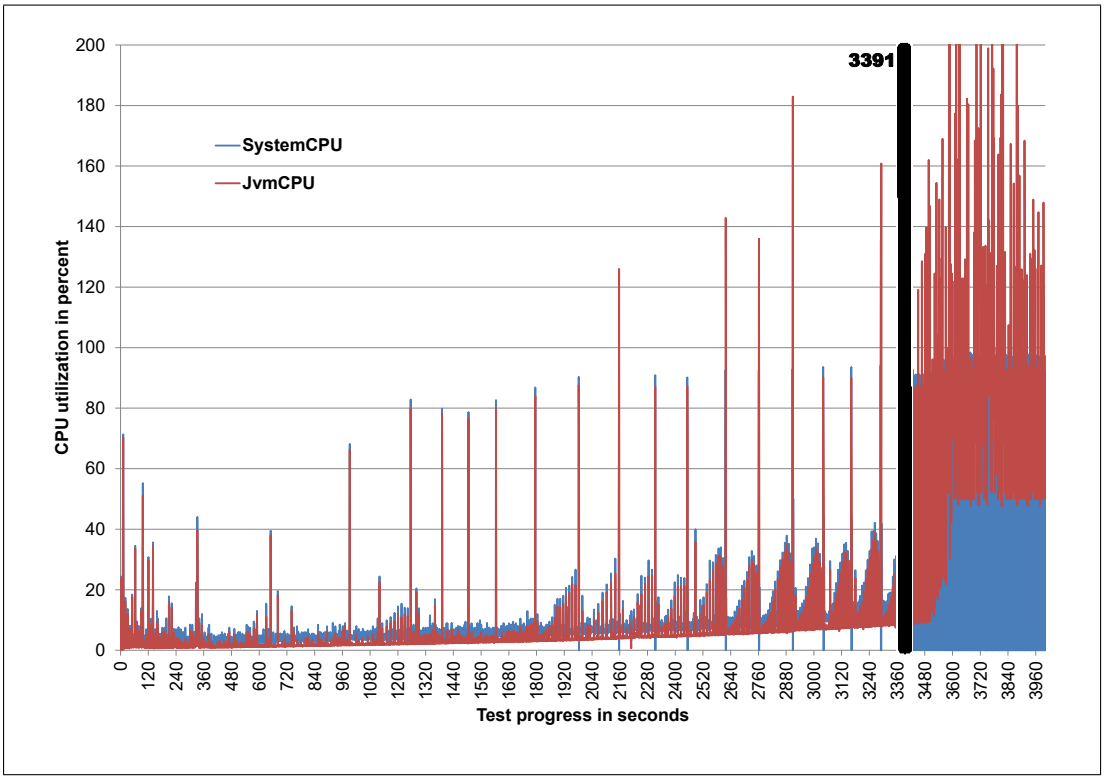

FIGURE 6.13.: CPU load of the load test

Figure 6.13 depicts the CPU utilization for this load test. Here, the maximum CPU load is 200%, because the Mac Mini is equipped with two CPU cores. While the CPU load is relatively constant during the first 30 minutes, the CPU load after this point of time shows a periodic sawtooth wave behavior. In general, the minimum CPU load increases linear over the test

progress until second 3391. At this point of time, the system is no longer working stable and properly.

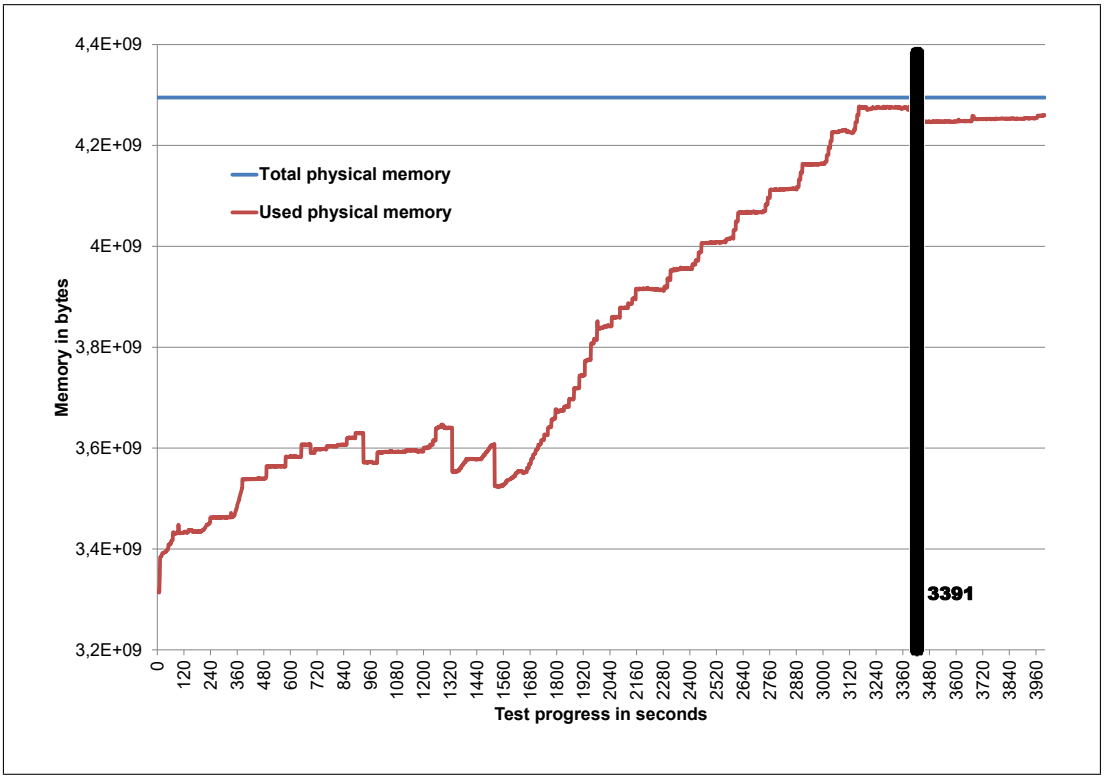

FIGURE 6.14.: Physical memory usage of the load test

Figure 6.14 depicts the memory utilization for this load test. Here, the blue curve illustrates the total physical memory of this Mac Mini. The red curve illustrates the physical memory that is in use. At second 3155 the maximum memory utilization is reached. Here, 99.21% of the total physical memory is in use. At this point of time, the threads at the *Intercloud Exchange* component cannot allocate memory fast enough in order to complete their tasks. From this point in time on the number of waiting tasks in the queue of the component's `ThreadPoolExecutor` increases rapidly. At second 3391, all tasks of the thread pool are in use and the full capacity of the task queue is reached. From this point in time on, no `PacketProcessor` tasks are created and an exception is thrown for each XMPP packet that is received.

6.2.3. Conclusion

The evaluations present that the specified protocols work and are well applied. Considering the performance of the prototypical implementation at second 3155, where the maximum memory utilization was reached, the *Intercloud Exchange* adds 315 new *LogEvents* per second to the CEP, processes 486630 *LogEvents* per second in 315 event streams, and evaluates 630 guar-

antee terms per second. This shows an excellent performance of the prototypical *Intercloud Exchange* implementation that is executed on a simple Mac Mini.

The limitation of the maximum number of SLA evaluations is connected to the number of events in the streams. This number of events to process is reflected in the memory usage, because all events in the streams are stored in memory. Thus, the memory of a system on which an *Intercloud Exchange* is executed should be dimensioned appropriately. In the Load Testing evaluation, the CEP used around one Gigabyte (exactly 943,665,152 Bytes) of memory for a time window of 15 minutes. Therefore, the staged CEP approach is introduced in section 6.1.4. This approach allows to transmit already aggregated measurements of a *Collector* link instance to an *Intercloud Exchange* and is thus relieving the CEP for SLA evaluations. If this is the case, *Agreement* instances could not only have plenty of terms, but also allow to define very large window sizes for *GuaranteeTerm* link instances.

7. Concluion and Future Work

This thesis analysed existing solutions for *Agreement-Mediators* on the cloud market and developed an actual autonomous web service based *Agreement-Mediator* for the intercloud project that sources out SLA management tasks into a neutral zone without taking the consumer's or the provider's side. Furthermore, an architecture is introduced that facilitates service discovery mechanisms and allows for comparing services based on their advertised QoS. The solutions in terms of agreement mediation, negotiation, establishment, and evaluation achieved in this thesis can be summarized as follows: individual service offerings can be advertised with individual service levels and guarantees that a provider is willing and able to deliver; consumers are able to discover and to easily compare service offerings of different providers; providers are able to publish service offerings of mutual intentions for business relationships in the form of SLA offers; monitoring services are supported in order to feed in monitoring measurements for automatic compliance verification of SLA terms; participating parties are notified about agreement-related events like creation, violation or termination; the overhead for managing SLAs between consumers and providers is minimized by Intercloud developed ready-to-use solutions; and higher trust is established by transferring mediation affiliations into an external and neutral entity.

Besides the achievements in terms of business related goals, this thesis introduces a novel protocol that allows to use the REST architectural style in XMPP-based solutions. Based on this protocol, the application of the OCCI Core Model to intercloud architectures is presented and appropriated protocol extensions that provide the SLA related capabilities are specified. These achievements close the lack of a missing service framework for the intercloud initiatives. Furthermore, a prototypical implementation with the functionalities and protocols presented in this work has been developed and is an ideal candidate for the initial setup of the IEEE Intercloud Testbed Project.

Finally, the protocols, the extensions, and the overall concept is evaluated. Here, the performance of this implementation and the functionality of the specified protocols is proved. Furthermore, the behavior of the system under extreme load, the benefits of a staged aggregation with distributed CEPs, and the maximum number of SLA evaluations for a specific hardware and test configuration is evaluated.

This thesis presents a foundation on which further Intercloud methodologies, protocol extensions, and related developments can be established. In particular, the following functionalities are considerable to extend the achievements presented in this thesis:

- A protocol extension to transfer huge files, e.g. a virtual machine image or snapshot.
- A protocol extension to transfer rapidly changing data sets, e.g. the memory pages of a running virtual machine that is going to live migrate.
- Business related protocol extensions for accounting, billing, and so on.

Another functional enhancement is related to the identity management which is out of scope of this thesis. In fact, the prototypical implementation of the *Marshaller* does not restricts the access to resources. An appropriated identity management would ensure that only privileged users are permitted to access resources which they own.

Further functional enhancements may cover a facilitated service life-cycle management, which allows non-technical employees to design product offerings with appropriated provisioning and SLA configurations. Furthermore, a service life-cycle management with corresponding business processes is considerable. Here, the nature of XMPP could be fully exploited, thus employees from different business divisions can be involved in such a business process and do an assigned work in this process that has to be accomplished by people.

Bibliography

[1] P. Mell and T. Grance, "The nist definition of cloud computing (draft)," National Institute of Standards and Technology, Tech. Rep., 2011.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, pp. 50–58, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672

[3] A. Stanik, M. Hovestadt, and O. Kao, "Hardware as a service (haas): The completion of the cloud stack," in *Proceedings of the 3rd Intl. Conference on Next Generation Information Technology (ICNIT)*, ser. ICNIT 2012, vol. 2. IEEE publishers, April 2012, pp. 830–836.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html

[5] A. Stanik, M. Hovestadt, and O. Kao, "Hardware as a service (haas): Physical and virtual hardware on demand," in *Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, ser. CLOUDCOM '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 149–154. [Online]. Available: http://dx.doi.org/10.1109/CloudCom.2012.6427579

[6] G. Wang, "The impact of virtualization on network performance of amazon ec2 data center," in *INFOCOM, 2010 Proceedings IEEE*, march 2010, pp. 1–9.

[7] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, and M. Morrow, "Blueprint for the intercloud - protocols and formats for cloud computing interoperability," in *Internet and Web Applications and Services, 2009. ICIW '09. Fourth International Conference on*, May 2009, pp. 328–336.

[8] F. Galán, A. Sampaio, L. Rodero-Merino, I. Loy, V. Gil, and L. M. Vaquero, "Service specification in cloud environments based on extensions to open standards," in *Proceedings of the Fourth International ICST Conference on COMmunication System*

softWAre and middlewaRE, ser. COMSWARE '09. New York, NY, USA: ACM, 2009, pp. 19:1–19:12. [Online]. Available: http://doi.acm.org/10.1145/1621890.1621915

[9] L. Lamers, et. all., "Open virtualization format specification," Distributed Management Task Force Inc. (DMTF), December 2013, latest version: 2.1.0, document number: DSP0243. [Online]. Available: http://www.dmtf.org/standards/vman

[10] Open Grid Forum (OGF), Open Cloud Computing Interface (OCCI) working group, "Open cloud computing interface - about," 2015, (accessed January 2015). [Online]. Available: http://occi-wg.org/about/

[11] P. Lipton, S. Moser, D. Palma, and T. Spatzier, "Topology and orchestration specification for cloud applications version 1.0," Organization for the Advancement of Structured Information Standards (OASIS), November 2013, oASIS Standard, TOSCA-v1.0. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html

[12] H. Rasheed, A. Rumpl, O. Wäldrich, and W. Ziegler, "A standards-based approach for negotiating service qos with cloud infrastructure providers," *eChallenges e-2012 Conference Proceedings*, 2012.

[13] W. K. Hon, C. Millard, and I. Walden, "Negotiating cloud contracts: Looking at clouds from both sides now." *Stanford Technology Law Review*, vol. 16, no. 1, pp. 79–129, 2013, fall 2012. [Online]. Available: http://stlr.stanford.edu/pdf/cloudcontracts.pdf

[14] "Amazon ec2 service level agreement," Amazon Web Services, Inc. or its affiliates., June 1 2013. [Online]. Available: http://aws.amazon.com/ec2/sla/

[15] "Amazon s3 sla," Amazon Web Services, Inc. or its affiliates., June 1 2013. [Online]. Available: http://aws.amazon.com/s3/sla/

[16] "Amazon rds service level agreement," Amazon Web Services, Inc. or its affiliates., June 1 2013. [Online]. Available: http://aws.amazon.com/rds/sla/

[17] "Google apps service level agreement," Google Inc., accessed February 2014. [Online]. Available: http://www.google.com/apps/intl/en/terms/sla.html

[18] "Google cloud storage, google prediction api, and google bigquery sla," Google Inc., accessed February 2014. [Online]. Available: https://developers.google.com/storage/sla

[19] "Windows azure support - technical and billing support: Service level agreements," Microsoft, January 2014. [Online]. Available: http://www.windowsazure.com/en-us/support/legal/sla/

[20] "Legal information: Cloud big data platform sla," Rackspace US, Inc., January 2014. [Online]. Available: http://www.rackspace.com/information/legal/cloud/sla

[21] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: Towards a cloud definition," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 50–55, Dec. 2008. [Online]. Available: http://doi.acm.org/10.1145/1496091.1496100

[22] "Best management practice portfolio: common glossary of terms and definitions," AXELOS Limited, October 2012, version 1. [Online]. Available: http://www.axelos.com/gempdf/Axelos_Common_Glossary_2013.pdf

[23] D. D. Lamanna, J. Skene, and W. Emmerich, "Slang: a language for defining service level agreements," in *Distributed Computing Systems, 2003. FTDCS 2003. Proceedings. The Ninth IEEE Workshop on Future Trends of*, May 2003, pp. 100–106.

[24] J. Skene, D. D. Lamanna, and W. Emmerich, "Precise service level agreements," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 179–188. [Online]. Available: http://dl.acm.org/citation.cfm?id=998675.999422

[25] A. Keller and H. Ludwig, "The wsla framework: Specifying and monitoring service level agreements for web services," *Journal of Network and Systems Management*, vol. 11, no. 1, pp. 57–81, 2003. [Online]. Available: http://dx.doi.org/10.1023/A%3A1022445108617

[26] H. Ludwig, A. Keller, A. Dan, R. P. King, and R. Franck, "Web service level agreement (wsla) language specification," *IBM Corporation*, pp. 815–824, 2003.

[27] V. Tosic, K. Patel, and B. Pagurek, "Wsol - web service offerings language," in *Web Services, E-Business, and the Semantic Web*, ser. Lecture Notes in Computer Science, C. Bussler, R. Hull, S. McIlraith, M. Orlowska, B. Pernici, and J. Yang, Eds. Springer Berlin Heidelberg, 2002, vol. 2512, pp. 57–67. [Online]. Available: http://dx.doi.org/10.1007/3-540-36189-8_5

[28] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, "Web services agreement specification (ws-agreement)," Open Grid Forum (OGF), Grid Resource Allocation Agreement Protocol (GRAAP) WG, 03 2007, updated version 2011. [Online]. Available: http://www.ogf.org/documents/GFD.192.pdf

[29] O. Wäldrich, D. Battre, F. Brazier, K. Clark, M. Oey, A. Papaspyrou, P. Wieder, and W. Ziegler, "Ws-agreement negotiation version 1.0," Open Grid Forum (OGF), Grid Resource Allocation Agreement Protocol (GRAAP) WG, 03 2011. [Online]. Available: http://www.ogf.org/documents/GFD.193.pdf

[30] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (xml)," *World Wide Web Journal*, vol. 2, no. 4, pp. 27–66, 1997. [Online]. Available: http://www.w3pdf.com/W3cSpec/XML/2/REC-xml11-20060816.pdf

[31] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, "Simple object access protocol (soap) 1.2," 2002. [Online]. Available: http://www.w3.org/TR/soap/

[32] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, *et al.*, "Web services description language (wsdl) 1.1," 2001. [Online]. Available: http://www.w3.org/TR/wsdl

[33] "Web services resource 1.2 (ws-resource)," OASIS, 04 2006. [Online]. Available: http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf

[34] D. Box, E. Christensen, F. Curbera, D. Ferguson, J. Frey, M. Hadley, C. Kaler, D. Langworthy, F. Leymann, B. Lovering, *et al.*, "Web services addressing (ws-addressing)," 2004.

[35] B. D., F. Brazier, K. Clark, M. Oey, A. Papaspyrou, O. Waäldrich, P. Wieder, and W. Ziegler, "A proposal for ws-agreement negotiation," in *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, Oct 2010, pp. 233–241.

[36] L. Srinivasan and T. Banks, "Web services resource lifetime 1.2 (ws-resourcelifetime) oasis standard," April 2006. [Online]. Available: http://docs.oasis-open.org/wsrf/wsrf-ws_resource_lifetime-1.2-spec-os.pdf

[37] Apache Software Foundation. (2013, January) Apache ws muse. Version 2.2.0, Project Website, (accessed January 2015). [Online]. Available: http://attic.apache.org/projects/muse.html

[38] Globus Alliance. (2009) Gt 5.0.0 release notes. Version 5 (GT5), Project Website, (accessed January 2015). [Online]. Available: http://toolkit.globus.org/toolkit/docs/5.0/5.0.0/rn/

[39] T. Berners-Lee, R. Fielding, and L. Masinter, "Rfc 3986: Uniform resource identifier (uri): Generic syntax," *The Internet Society*, 2005.

[40] O. Wäldrich *et al.* Wsag4j: Web service agreement for java. Version 2.0, Project Website. [Online]. Available: http://wsag4j.sourceforge.net

[41] D. Battre, P. Wieder, and W. Ziegler, "Ws-agreement specification version 1.0 experience document," Open Grid Forum, March 2010. [Online]. Available: https://www.ogf.org/documents/GFD.167.pdf

[42] D. Guinard, I. Ion, and S. Mayer, "In search of an internet of things service architecture: Rest or ws-*? a developers' perspective," in *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, A. Puiatti and T. Gu, Eds. Springer Berlin Heidelberg, 2012, vol. 104, pp. 326–337. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30973-1_32

[43] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, 2000, AAI9980887.

[44] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. "big"' web services: Making the right architectural decision," in *Proceedings of the 17th International Conference on World Wide Web*, ser. WWW '08. New York, NY, USA: ACM, 2008, pp. 805–814. [Online]. Available: http://doi.acm.org/10.1145/1367497.1367606

[45] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the web services web: an introduction to soap, wsdl, and uddi," *Internet Computing, IEEE*, vol. 6, no. 2, pp. 86–93, March 2002.

[46] M. J. Hadley, "Web application description language (wadl)," Sun Microsystems, Inc., Mountain View, CA, USA, Tech. Rep., 2006.

[47] ——, "Web application description language (wadl) specification," 2009. [Online]. Available: http://www.w3.org/Submission/wadl/

[48] R. Kübert, G. Katsaros, and T. Wang, "A restful implementation of the ws-agreement specification," in *Proceedings of the Second International Workshop on RESTful Design*, ser. WS-REST '11. New York, NY, USA: ACM, 2011, pp. 67–72. [Online]. Available: http://doi.acm.org/10.1145/1967428.1967444

[49] F. Blumel, T. Metsch, and A. Papaspyrou, "A restful approach to service level agreements for cloud environments," in *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, Dec 2011, pp. 650–657.

[50] O. Wäldrich, "Orchestration of resources in distributed, heterogeneous grid environments using dynamic service level agreements," Ph.D. dissertation, Technische Universität Dortmund, Sankt Augustin, 12 2011.

[51] W3C, "Team comment on the "web application description language" submission," 2009. [Online]. Available: http://www.w3.org/Submission/2009/03/Comment

[52] D. Clinton, J. Tesler, M. Fagan, J. Gregorio, A. Sauve, and J. Snell, "Opensearch 1.1 specification (draft 5)," 2014. [Online]. Available: http://www.opensearch.org/Specifications/OpenSearch/1.1

[53] J. Gregorio and B. M. Consulting, "The atom publishing protocol," Network Working Group, Tech. Rep., 2007. [Online]. Available: http://tools.ietf.org/search/rfc5023

[54] M. Kelly, "Json hypertext application language," 2013. [Online]. Available: http://tools.ietf.org/html/draft-kelly-json-hal-06

[55] M. zur Muehlen, J. V. Nickerson, and K. D. Swenson, "Developing web services choreography standards - the case of {REST} vs. {SOAP}," *Decision Support Systems*, vol. 40, no. 1, pp. 9–29, 2005, web services and process management. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167923604000612

[56] B. Upadhyaya, Y. Zou, H. Xiao, J. Ng, and A. Lau, "Migration of soap-based services to restful services," in *Web Systems Evolution (WSE), 2011 13th IEEE International Symposium on*, 2011, pp. 105–114.

[57] G. Mulligan and D. Gracanin, "A comparison of soap and rest implementations of a service based interaction independence middleware framework," in *Simulation Conference (WSC), Proceedings of the 2009 Winter*, 2009, pp. 1423–1432.

[58] M. Comuzzi and G. Spanoudakis, "Dynamic set-up of monitoring infrastructures for service based systems," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 2414–2421. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774591

[59] "The grinder, a java load testing framework," 2013. [Online]. Available: http://grinder.sourceforge.net

[60] A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo, "Web services security: Soap message security 1.0," 2004. [Online]. Available: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf

[61] D. Eastlake and J. Reagle, "Xml signature," IETF and W3C, 2000. [Online]. Available: http://www.w3.org/Signature/

[62] J. Durand, A. Otto, G. Pilz, and T. Rutt, "Cloud application management for platforms version 1.1," Organization for the Advancement of Structured Information Standards (OASIS), November 2014, oASIS Standard, CAMP-v1.1. [Online]. Available: http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.html

[63] D. Doug and G. Pilz, "Cloud infrastructure management interface (cimi) model and restful http-based protocol)," Distributed Management Task Force Inc. (DMTF), September 2012, latest version: 1.0.1, document number: DSP0263. [Online]. Available: http://dmtf.org/standards/cmwg

[64] R. Nyren, A. Edmond, A. Papaspyrou, and T. Metsch, "Open cloud computing interface - core," Open Grid Forum (OGF), Open Cloud Computing Interface (OCCI) working group, April 2011, updated version: GFD-P-R.183. [Online]. Available: https://www.ogf.org/documents/GFD.183.pdf

[65] D. A. Moon, "Object-oriented programming with flavors," in *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, ser. OOPLSA '86. New York, NY, USA: ACM, 1986, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/28697.28698

[66] T. Metsch and A. Edmond, "Open cloud computing interface - restful http rendering," Open Grid Forum (OGF), Open Cloud Computing Interface (OCCI) working group, June 2011, version: GFD-P-R.185. [Online]. Available: https://www.ogf.org/documents/GFD.185.pdf

[67] ——, "Open cloud computing interface - infrastructure," Open Grid Forum (OGF), Open Cloud Computing Interface (OCCI) working group, April 2011, updated version: GFD-P-R.184. [Online]. Available: https://www.ogf.org/documents/GFD.184.pdf

[68] M. Papazoglou and W.-J. van den Heuvel, "Service oriented architectures: approaches, technologies and research issues," *The VLDB Journal*, vol. 16, no. 3, pp. 389–415, 2007. [Online]. Available: http://dx.doi.org/10.1007/s00778-007-0044-3

[69] L. Clement, A. Hately, C. von Riegen, and T. Rogers, "Universal description, discovery and integration v3.0.2 (uddi)," Organization for the Advancement of Structured Information Standards (OASIS), February 2005, uDDI Spec Technical Committee Draft, Dated 20041019. [Online]. Available: http://www.uddi.org/pubs/uddi_v3.htm

[70] H. Ludwig, "Ws-agreement concepts and use - agreement-based service-oriented architectures," IBM Research Report, Tech. Rep., 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.121.476&rep=rep1&type=pdf

[71] A. Cruz, P. Ramalheira, and E. Troup, "Service delivery broker," SAPO part of Portugal Telecom, Tech. Rep., January 2012, whitepaper v1.1. [Online]. Available: http://sdb.sapo.pt/en/index.html

[72] D. Tektonidis, A. Bokma, G. Oatley, and M. Salampasis, "Onar: An ontologies-based service oriented application integration framework," in *Interoperability of Enterprise Software and Applications*, D. Konstantas, J.-P. Bourrires, M. Leonard, and N. Boudjlida, Eds. Springer London, 2006, pp. 65–74. [Online]. Available: http://dx.doi.org/10.1007/1-84628-152-0_7

[73] MO-BIZZ Marketplace, 2015, (accessed May 2015). [Online]. Available: https://market.mobizz-project.eu/en/

[74] EsperTech, "Event series intelligence - continuous event processing for the right time enterprise," EsperTech Inc., Tech. Rep., 2015, product overview technical Data Sheet. [Online]. Available: http://www.espertech.com/download/public/EsperTech%20technical%20datasheet.pdf

[75] TM Forum, "Software enabled services management interface (smi) specification package-tip smi specification v1.6.0 standard," 2012, (accessed May 2015). [Online]. Available: https://www.tmforum.org/resources/standard/smi-specification-packagetip_smi_specification__v1-6-0/

[76] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?" *In USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[77] ——, "Flowvisor: A network virtualization layer," *Technical Report Openflow-tr-2009-1, Stanford University*, July 2009.

[78] E. Salvadori, R. Corin, A. Broglio, and M. Gerola, "Generalizing virtual network topologies in openflow-based networks," in *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, Dec., pp. 1–6.

[79] R. Corin, M. Gerola, R. Riggio, F. De Pellegrini, and E. Salvadori, "Vertigo: Network virtualization and beyond," in *Software Defined Networking (EWSDN), 2012 European Workshop on*, oct. 2012, pp. 24 –29.

[80] M. Koerner and H. Almus, "Hla - a hierarchical layer application for openflow management abstraction," in *Proceedings of the Fourth International Conference on Network of the Future (NoF'13)*, Pohang, Korea, oct 2013, pp. 1–4.

[81] M. Koerner, A. Stanik, and O. Kao, "Applying qos in software defined networks by using ws-agreement," in *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, Dec 2014, pp. 893–898.

[82] The OpenFlow Consortium, "Openflow switch specification - version 1.0.0," Decmber 2009. [Online]. Available: http://www.openflow.org/wp/documents/

[83] Open Networking Foundation, "Openflow switch specification - version 1.4.0," October 2013. [Online]. Available: https://www.opennetworking.org/

[84] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, April 2008.

[85] A. Fei, G. Pei, R. Liu, and L. Zhang, "Measurements on delay and hop-count of the internet," in *IEEE GLOBECOM'98-Internet Mini-Conference*. Citeseer, 1998.

[86] F. Begtaševiü and P. Van Mieghem, "Measurements of the hopcount in internet," in *Proceedings of Passive and Active Measurement (PAM*, 2001, pp. 23–24.

[87] S. García-Gómez, M. Jimenez-Ganan, Y. Taher, C. Momm, F. Junker, J. Biro, A. Menychtas, V. Andrikopoulos, and S. Strauch, "Challenges for the comprehensive management of cloud services in a paas framework," *Scalable Computing: Practice and Experience*, vol. 13, no. 3, pp. 201–2013, 2012.

[88] A. Menychtas, S. Gomez, A. Giessmann, A. Gatzioura, K. Stanoevska, J. Vogel, and V. Moulos, "A marketplace framework for trading cloud-based services," in *Economics of Grids, Clouds, Systems, and Services*, ser. Lecture Notes in Computer Science, K. Vanmechelen, J. Altmann, and O. Rana, Eds. Springer Berlin Heidelberg, 2012, vol. 7150, pp. 76–89. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28675-9_6

[89] D. Zeginis, F. D'Andria, S. Bocconi, J. Gorronogoitia Cruz, O. Collell Martin, P. Gouvas, G. Ledakis, and K. A. Tarabanis, "A user-centric multi-paas application management solution for hybrid multi-cloud scenarios," *Scalable Computing: Practice and Experience*, vol. 14, no. 1, pp. 17–32, 2013.

[90] M. Comuzzi, C. Kotsokalis, C. Rathfelder, W. Theilmann, U. Winkler, and G. Zacco, "A framework for multi-level sla management," in *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*, ser. Lecture Notes in Computer Science, A. Dan, F. Gittler, and F. Toumani, Eds. Springer Berlin Heidelberg, 2010, vol. 6275, pp. 187–196. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16132-2_18

[91] J. Happe, W. Theilmann, A. Edmonds, and K. T. Kearney, "A reference architecture for multi-level sla management," in *Service Level Agreements for Cloud Computing*, P. Wieder, J. M. Butler, W. Theilmann, and R. Yahyapour, Eds. Springer New York, 2011, pp. 13–26. [Online]. Available: http://dx.doi.org/10.1007/978-1-4614-1614-2_2

[92] A. Ciuffoletti, A. Congiusta, G. Jankowski, M. Jankowski, O. Krajicek, and N. Meyer, "Grid infrastructure architecture: A modular approach from coregrid," in *Web Information Systems and Technologies*, ser. Lecture Notes in Business Information Processing, J. Filipe and J. Cordeiro, Eds. Springer Berlin Heidelberg, 2008, vol. 8, pp. 72–84. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68262-2_6

[93] C. Cacciari, D. Mallmann, C. Zsigri, F. D'Andria, B. Hagemeier, A. Rumpl, W. Ziegler, and J. Martrat, "Sla-based management of software licenses as web service resources in distributed computing infrastructures," *Future Generation Computer Systems*, vol. 28, no. 8, pp. 1340–1349, 2012, including Special sections SS: Trusting Software Behavior and SS: Economics of Computing Services. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X11002287

[94] A. Lawrence, K. Djemame, O. Wäldrich, W. Ziegler, and C. Zsigri, "Using service level agreements for optimising cloud infrastructure services," in *Towards a Service-Based Internet. ServiceWave 2010 Workshops*, ser. Lecture Notes in Computer Science, M. Cezon and Y. Wolfsthal, Eds. Springer Berlin Heidelberg, 2011, vol. 6569, pp. 38–49. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22760-8_4

[95] R. Kuebert, A. Tenschert, O. Wäldrich, W. Ziegler, and D. Battre, "A service level agreement layer for the d-grid infrastructure."

[96] M. Couceiro, P. Ruivo, P. Romano, and L. Rodrigues, "Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, June 2013, pp. 1–12.

[97] M. Couceiro, P. Romano, and L. Rodrigues, "Polycert: Polymorphic self-optimizing replication for in-memory transactional grids," in *Proceedings of the 12th International Middleware Conference*, ser. Middleware '11. Laxenburg, Austria, Austria: International Federation for Information Processing, 2011, pp. 300–319. [Online]. Available: http://dl.acm.org/citation.cfm?id=2414338.2414360

[98] R. Cascella, L. Blasi, Y. Jegou, M. Coppola, and C. Morin, "Contrail: Distributed application deployment under sla in federated heterogeneous clouds," in *The Future Internet*, ser. Lecture Notes in Computer Science, A. Galis and A. Gavras, Eds. Springer Berlin Heidelberg, 2013, vol. 7858, pp. 91–103. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38082-2_8

[99] E. Carlini, M. Coppola, P. Dazzi, L. Ricci, and G. Righetti, "Cloud federations in contrail," in *Euro-Par 2011: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science, M. Alexander, P. D'Ambra, A. Belloum, G. Bosilca, M. Cannataro, M. Danelutto, B. Di Martino, M. Gerndt, E. Jeannot, R. Namyst, J. Roman, S. Scott, J. Traff, G. Valle, and J. Weidendorfer, Eds. Springer Berlin Heidelberg, 2012, vol. 7155, pp. 159–168. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-29737-3_19

[100] K. Voss, K. Djemame, I. Gourlay, and J. Padgett, "Assessgrid, economic issues underlying risk awareness in grids," in *Grid Economics and Business Models*, ser. Lecture Notes in Computer Science, D. Veit and J. Altmann, Eds. Springer Berlin Heidelberg, 2007, vol. 4685, pp. 170–175. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74430-6_14

[101] D. Battre, K. Djemame, I. Gourlay, M. Hovestadt, O. Kao, J. Padgett, K. Voss, and D. Warneke, "Assessgrid strategies for provider ranking mechanisms in risk-aware grid systems," in *Grid Economics and Business Models*, ser. Lecture

Notes in Computer Science, J. Altmann, D. Neumann, and T. Fahringer, Eds. Springer Berlin Heidelberg, 2008, vol. 5206, pp. 226–233. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85485-2_19

[102] H. Pouyllau and R. Douville, "End-to-end qos negotiation in network federations," in *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*, April 2010, pp. 173–176.

[103] N. Le Sauze, A. Chiosi, R. Douville, H. Pouyllau, H. Lonsethagen, P. Fantini, C. Palasciano, A. Cimmino, M. C. Rodriguez, O. Dugeon, *et al.*, "Etics: Qos-enabled interconnection for future internet services," *Future network and mobile summit*, 2010. [Online]. Available: https://www.ict-etics.eu/fileadmin/documents/publications/scientific_papers/ETICS_mobile_summit2010.pdf

[104] H. Pouyllau and G. Carofiglio, "Inter-carrier sla negotiation using q-learning," *Telecommunication Systems*, vol. 52, no. 2, pp. 611–622, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11235-011-9505-5

[105] G. Brataas, E. Stav, S. Lehrig, S. Becker, G. Kopčak, and D. Huljenic, "Cloudscale: Scalability management for cloud systems," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '13. New York, NY, USA: ACM, 2013, pp. 335–338. [Online]. Available: http://doi.acm.org/10.1145/2479871.2479920

[106] R. Jimenez-Peris, M. Patiño-Martinez, K. Magoutis, A. Bilas, and I. Brondino, "Cumulonimbo: A highly-scalable transaction processing platform as a service," *ERCIM News*, vol. 89, no. null, pp. 34–35, 2012.

[107] I. Anghel, M. Bertoncini, T. Cioara, M. Cupelli, V. Georgiadou, P. Jahangiri, A. Monti, S. Murphy, A. Schoofs, and T. Velivassaki, "Geyser: Enabling green data centres in smart cities," in *Energy Efficient Data Centers*, ser. Lecture Notes in Computer Science, S. Klingert, M. Chinnici, and M. Rey Porto, Eds. Springer International Publishing, 2015, vol. 8945, pp. 71–86. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-15786-3_5

[108] B. Jones and F. Casu, "Helix nebula-the science cloud: a public-private partnership to build a multidisciplinary cloud platform for data intensive science," in *EGU General Assembly Conference Abstracts*, vol. 15, 2013, p. 1510.

[109] A.-F. Antonescu, A. Gomes, P. Robinson, and T. Braun, "Sla-driven predictive orchestration for distributed cloud-based mobile services," in *Communications Workshops (ICC), 2013 IEEE International Conference on*, June 2013, pp. 738–743.

[110] D. Ardagna, E. di Nitto, P. Mohagheghi, S. Mosser, C. Ballagny, F. D'Andria, G. Casale, P. Matthews, C.-S. Nechifor, D. Petcu, A. Gericke, and C. Sheridan, "Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds," in *Modeling in Software Engineering (MISE), 2012 ICSE Workshop on*, June 2012, pp. 50–56.

[111] E. Tego, F. Matera, V. Attanasio, and D. Del Buono, "Quality of service management based on software defined networking approach in wide gbe networks," in *Euro Med Telco Conference (EMTC), 2014*, Nov 2014, pp. 1–5.

[112] M. Addis, M. Jacyno, M. Hall-May, M. McArdle, and S. Phillips, "Planning and managing the cost of compromise for av retention and access," *Motion Imaging Journal, SMPTE*, vol. 121, no. 1, pp. 32–38, Feb 2012.

[113] M. Surridge, A. Chakravarthy, M. Hall-May, X. Chen, B. Nasser, and R. Nossal, "Serscis: Semantic modelling of dynamic, multi-stakeholder systems," November 2012. [Online]. Available: http://eprints.soton.ac.uk/349295/

[114] F. Longo, D. Bruneo, M. Villari, A. Puliafito, E. Salant, and Y. Wolfsthal, "From vision cloud to cloudwave: Towards the future internet and a new generation of services," in *Intelligent Networking and Collaborative Systems (INCoS), 2014 International Conference on*, Sept 2014, pp. 641–646.

[115] A. Barker, B. Varghese, and L. Thai, "Cloud services brokerage: A survey and research roadmap," in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, June 2015, pp. 1029–1032.

[116] A. Amato, B. Di Martino, and S. Venticinque, "A distributed cloud brokering service," *INFORMATICA*, vol. 26, no. 1, pp. 1–15, 2015.

[117] A. Amato, G. Cretella, B. Di Martino, and S. Venticinque, "Semantic and agent technologies for cloud vendor agnostic resource brokering," in *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*, March 2013, pp. 1253–1258.

[118] G. Anastasi, E. Carlini, M. Coppola, and P. Dazzi, "Qbrokage: A genetic approach for qos cloud brokering," in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, June 2014, pp. 304–311.

[119] S. Sundareswaran, A. Squicciarini, and D. Lin, "A brokerage-based approach for cloud service selection," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, June 2012, pp. 558–565.

[120] I. U. Haq, A. A. Huqqani, and E. Schikuta, "A conceptual model for aggregation and validation of slas in business value networks," in *The 3rd International Conference on Adaptive Business Information Systems (ABIS 2009)*, March 2009. [Online]. Available: http://eprints.cs.univie.ac.at/175/

[121] P. Pawluk, B. Simmons, M. Smit, M. Litoiu, and S. Mankovski, "Introducing stratos: A cloud broker service," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, June 2012, pp. 891–898.

[122] I. ul Haq, A. Huqqani, and E. Schikuta, "Aggregating hierarchical service level agreements in business value networks," in *Business Process Management*, ser. Lecture Notes in Computer Science, U. Dayal, J. Eder, J. Koehler, and H. Reijers, Eds. Springer Berlin Heidelberg, 2009, vol. 5701, pp. 176–192. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03848-8_13

[123] S. Wieser and L. Boszormenyi, "Decentralized topology aggregation for qos estimation in large overlay networks," in *Network Computing and Applications (NCA), 2011 10th IEEE International Symposium on*, Aug 2011, pp. 298–301.

[124] "IEEE Cloud Computing Initiative," IEEE Foundation, 2015, web Side, accessed June 2015. [Online]. Available: http://cloudcomputing.ieee.org/

[125] "IEEE Standards Association," IEEE Foundation, 2015, web Side, accessed June 2015. [Online]. Available: http://standards.ieee.org/index.html

[126] "IEEE Project P2301 - Guide for Cloud Portability and Interoperability Profiles (CPIP)," IEEE Computer Society, CPWG/2301 WG - Cloud Profiles WG (CPWG) Working Group, 2015, web Side, accessed June 2015. [Online]. Available: https://standards.ieee.org/develop/project/2301.html

[127] "IEEE Project P2302 - Standard for Intercloud Interoperability and Federation (SIIF)," IEEE Computer Society, ICWG/2302 WG - Intercloud WG (ICWG) Working Group, 2015, web Side, accessed June 2015. [Online]. Available: https://standards.ieee.org/develop/project/2302.html

[128] "IEEE Cloud Computing Intercloud Testbed," IEEE Intercloud Testbed, IEEE Cloud Computing Initiative, 2015, web Side, accessed June 2015. [Online]. Available: http://www.intercloudtestbed.org/

[129] "The XMPP Standards Foundation," XMPP Standards Foundation, 2015, web Side, accessed June 2015. [Online]. Available: http://xmpp.org/

[130] P. Saint-Andre, "Extensible messaging and presence protocol (xmpp): Core," *RFC 6120*, March 2011. [Online]. Available: http://xmpp.org/rfcs/rfc6120.html

[131] P. Waher, "Xep-0332: Http over xmpp transport," *XMPP Standards Foundation (XSF)*, 2014, version: 0.3. [Online]. Available: http://xmpp.org/extensions/xep-0332.html

[132] J. Hildebrand, P. Millard, R. Eatmon, and P. Saint-Andre, "Xep-0030: Service discovery," *XMPP Standards Foundation (XSF)*, 2008, version: 2.4. [Online]. Available: http://xmpp.org/extensions/xep-0030.html

[133] N. Grozev and R. Buyya, "Inter-cloud architectures and application brokering: taxonomy and survey," *Software: Practice and Experience*, vol. 44, no. 3, pp. 369–390, 2014. [Online]. Available: http://dx.doi.org/10.1002/spe.2168

[134] D. Bernstein, D. Vij, and S. Diamond, "An intercloud cloud computing economy - technology, governance, and market blueprints," in *SRII Global Conference (SRII), 2011 Annual*, March 2011, pp. 293–299.

[135] D. Bernstein and D. Vij, "Intercloud directory and exchange protocol detail using xmpp and rdf," in *Services (SERVICES-1), 2010 6th World Congress on*, July 2010, pp. 431–438.

[136] L. Yu, *A Developer's Guide to the Semantic Web*. Springer-Verlag Berlin Heidelberg, 2014, vol. 2.

[137] G. Klyne, J. J. Carroll, and B. McBride, "Rdf 1.1 concepts and abstract syntax," Februray 2014. [Online]. Available: http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/

[138] B. Di Martino, G. Cretella, A. Esposito, A. Willner, A. Alloush, D. Bernstein, D. Vij, and J. Weinman, "Towards an ontology-based intercloud resource catalogue – the ieee p2302 intercloud approach for a semantic resource exchange," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, March 2015, pp. 458–464.

[139] A. Willner, R. Loughnane, and T. Magedanz, "Fiddle: Federated infrastructure discovery and description language," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, March 2015, pp. 465–471.

[140] D. Bernstein and D. Vij, "Intercloud exchanges and roots topology and trust blueprint," in *Proc. of 11th International Conference on Internet Computing*, 2011, pp. 135–141. [Online]. Available: http://weblidi.info.unlp.edu.ar/worldcomp2011-mirror/ICM3485.pdf

[141] ——, "Intercloud federation using via semantic resource federation api and dynamic sdn provisioning," in *Network of the Future (NOF), 2014 International Conference and Workshop on the*, vol. Workshop, Dec 2014, pp. 1–8.

[142] ——, "Intercloud security considerations," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, Nov 2010, pp. 537–544.

[143] ——, "Using semantic web ontology for intercloud directories and exchanges," in *International Conference on Internet Computing*, 2010, pp. 18–24. [Online]. Available: http://www.cloudstrategypartners.com/resources/Using+Semantic+Web+Ontology+for+Intercloud+Directories+and+Exchanges+-+Draft.pdf

[144] ——, "Using xmpp as a transport in intercloud protocols," in *2010 the 2nd International Conference on Cloud Computing, CloudComp*, 2010. [Online]. Available: http://www.cloudstrategypartners.com/resources/Using+XMPP+as+a+transport+in+Intercloud+Protocols+-+Draft+Copy.pdf

[145] D. Bernstein and Y. Demchenko, "The ieee intercloud testbed – creating the global cloud of clouds," in *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, vol. 2, Dec 2013, pp. 45–50.

[146] "IEEE P2302/D0.9 - Draft Standard for Intercloud Interoperability and Federation (SIIF)," IEEE Computer Society, ICWG/2302 WG - Intercloud WG (ICWG) Working Group, January 2015, unpublished, Restricted Member Access. [Online]. Available: https://mentor.ieee.org/p2302/dcn/15/p2302-15-0004-00-DRFT-intercloud-p2302-draft-0-9.doc

[147] J. Wagener, E. Willighagen, A. Heusler, T. Markmann, and O. Spjuth, "Xep-0244: Io data," *XMPP Standards Foundation (XSF)*, 2008, version: 0.1. [Online]. Available: http://xmpp.org/extensions/xep-0244.html

[148] ——. Xws4j: Xmpp web services for java (xws4j). Version 09.05.19, Project Website, accessed November 2015. [Online]. Available: http://xws4j.sourceforge.net/index.html

[149] J. Wagener, O. Spjuth, E. Willighagen, and J. Wikberg, "Xmpp for cloud computing in bioinformatics supporting discovery and invocation of asynchronous web services," *BMC Bioinformatics*, vol. 10, p. 279, 2009. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2755485/

[150] Internet Society (ISOC), 2015, web Side, accessed November 2015. [Online]. Available: http://www.internetsociety.org/

[151] Internet Corporation for Assigned Names and Numbers (ICANN), 2015, web Side, accessed November 2015. [Online]. Available: https://www.icann.org/

[152] F. Forno and P. Saint-Andre, "Xep-0072: Soap over xmpp," *XMPP Standards Foundation (XSF)*, 2005, version: 1.0. [Online]. Available: http://xmpp.org/extensions/xep-0072.html

[153] D. Winer, "XML-RPC Specification," Scripting News, Inc., 2003, web Side, accessed June 2015, Version: updated 6/30/03 DW. [Online]. Available: http://xmlrpc.scripting.com/spec.html

[154] D. Adams, "Xep-0009: Jabber-rpc," *XMPP Standards Foundation (XSF)*, 2011, version: 2.2. [Online]. Available: http://xmpp.org/extensions/xep-0009.html

[155] H. Nomoto, "State oriented programming," in *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*, March 2004, pp. 304–305.

[156] T. Berners-Lee, R. T. Fielding, and L. Masinter, "Uniform resource identifiers (uri): Generic syntax," *The Internet Society*, 1998, rFC 2396. [Online]. Available: https://www.ietf.org/rfc/rfc2396.txt

[157] P. Saint-Andre, "Xep-0114: Jabber component protocol," *XMPP Standards Foundation (XSF)*, 2012, version: 1.6. [Online]. Available: http://xmpp.org/extensions/xep-0114.html

[158] P. Waher, "Xep-0337: Event logging over xmpp," *XMPP Standards Foundation (XSF)*, 2008, version: 0.1. [Online]. Available: http://xmpp.org/extensions/xep-0337.html

[159] Igniterealtime Open Source community, "Smack api," Jive Software, 2015, latest version: 4.1.5. [Online]. Available: http://www.igniterealtime.org/projects/smack/

[160] ——, "Whack api," Jive Software, 2015, latest version: 2.0.0. [Online]. Available: http://www.igniterealtime.org/projects/whack/

[161] Apache Software Foundation, "Apache wicket," Apache Software Foundation, 2015, latest version: v7.0. [Online]. Available: http://wicket.apache.org/

[162] Bootstrap Community, "Bootstrap is the most popular html, css, and js framework for developing responsive, mobile first projects on the web," Twitter, 2015, latest version: v3.3.6. [Online]. Available: http://getbootstrap.com/

A. Appendix

Contents

A.1. XWADL Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="urn:xmpp:rest-xwadl" xmlns="urn:xmpp:rest-xwadl"
  elementFormDefault="qualified">

  <xs:element name="resource_type">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="documentation" type="documentationType"
          minOccurs="0" maxOccurs="1" />
        <xs:element ref="grammars" minOccurs="0" maxOccurs="1" />
        <xs:element ref="method" minOccurs="0" maxOccurs="unbounded" />
        <xs:element ref="action" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="path" type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:complexType name="documentationType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="title" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
```

```
  </xs:complexType>

  <xs:element name="grammars">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="documentation" type="documentationType"
          minOccurs="0" maxOccurs="unbounded" />
        <xs:any namespace="##other" processContents="lax" minOccurs="0"
          maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="method">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="documentation" type="documentationType"
          minOccurs="0" maxOccurs="1" />
        <xs:element ref="request" minOccurs="0" maxOccurs="1" />
        <xs:element ref="response" minOccurs="0" maxOccurs="1" />
      </xs:sequence>
      <xs:attribute name="type" type="methodType" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:simpleType name="methodType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="GET" />
      <xs:enumeration value="POST" />
      <xs:enumeration value="PUT" />
      <xs:enumeration value="DELETE" />
    </xs:restriction>
  </xs:simpleType>

  <xs:element name="request">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="documentation" type="documentationType"
          minOccurs="0" maxOccurs="1" />
        <xs:element name="template" type="xs:string" minOccurs="0"
          maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="mediaType" type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>
```

```
<xs:element name="response">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="documentation" type="documentationType"
        minOccurs="0" maxOccurs="1" />
    </xs:sequence>
    <xs:attribute name="mediaType" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>

<xs:element name="action">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="documentation" type="documentationType"
        minOccurs="0" maxOccurs="1" />
      <xs:element ref="parameter" minOccurs="0" maxOccurs="unbounded"
        />
      <xs:element ref="result" minOccurs="0" maxOccurs="1" />
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>

<xs:element name="parameter">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="documentation" type="documentationType"
        minOccurs="0" maxOccurs="1" />
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required" />
    <xs:attribute name="default" type="xs:string" />
    <xs:attribute name="type" type="parameterType" use="required" />
  </xs:complexType>
</xs:element>

<xs:element name="result">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="documentation" type="documentationType"
        minOccurs="0" maxOccurs="1" />
    </xs:sequence>
    <xs:attribute name="type" type="parameterType" use="required" />
  </xs:complexType>
</xs:element>
```

```
  <xs:simpleType name="parameterType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="STRING" />
      <xs:enumeration value="INTEGER" />
      <xs:enumeration value="DOUBLE" />
      <xs:enumeration value="BOOLEAN" />
      <xs:enumeration value="LINK" />
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

LISTING A.1: XWADL Schema

A.2. REST-XML Schema

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="urn:xmpp:xml-rest" xmlns="urn:xmpp:xml-rest"
  elementFormDefault="qualified">

  <xs:element name="resource">
    <xs:complexType>
      <xs:choice>
        <xs:element ref="method" minOccurs="1" maxOccurs="1" />
        <xs:element ref="action" minOccurs="1" maxOccurs="1" />
      </xs:choice>
      <xs:attribute name="path" type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:element name="method">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="request" minOccurs="0" maxOccurs="1" />
        <xs:element ref="response" minOccurs="0" maxOccurs="1" />
      </xs:sequence>
      <xs:attribute name="type" type="methodType" use="required" />
    </xs:complexType>
  </xs:element>
```

```
<xs:simpleType name="methodType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="GET" />
    <xs:enumeration value="POST" />
    <xs:enumeration value="PUT" />
    <xs:enumeration value="DELETE" />
  </xs:restriction>
</xs:simpleType>

<xs:element name="request">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="representation" type="xs:string"
        minOccurs="0" maxOccurs="1" />
      <xs:any namespace="##other" processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="mediaType" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>

<xs:element name="response">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="representation" type="xs:string"
        minOccurs="0" maxOccurs="1" />
      <xs:any namespace="##other" processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="mediaType" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>

<xs:element name="action">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="parameter" minOccurs="0" maxOccurs="unbounded"
        />
      <xs:element ref="result" minOccurs="0" maxOccurs="1" />
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>
```

```
  <xs:element name="parameter">
    <xs:complexType>
      <xs:choice>
        <xs:element name="STRING" type="xs:string" minOccurs="1"
          maxOccurs="1" />
        <xs:element name="INTEGER" type="xs:integer" minOccurs="1"
          maxOccurs="1" />
        <xs:element name="DOUBLE" type="xs:double" minOccurs="1"
          maxOccurs="1" />
        <xs:element name="BOOLEAN" type="xs:boolean" minOccurs="1"
          maxOccurs="1" />
        <xs:element name="LINK" type="xs:anyURI" minOccurs="1"
          maxOccurs="1" />
      </xs:choice>
      <xs:attribute name="name" type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:element name="result">
    <xs:complexType>
      <xs:choice>
        <xs:element name="STRING" type="xs:string" minOccurs="1"
          maxOccurs="1" />
        <xs:element name="INTEGER" type="xs:integer" minOccurs="1"
          maxOccurs="1" />
        <xs:element name="DOUBLE" type="xs:double" minOccurs="1"
          maxOccurs="1" />
        <xs:element name="BOOLEAN" type="xs:boolean" minOccurs="1"
          maxOccurs="1" />
        <xs:element name="LINK" type="xs:anyURI" minOccurs="1"
          maxOccurs="1" />
      </xs:choice>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

LISTING A.2: REST-XML Schema

A.3. Classification Schema

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="urn:xmpp:occi-classification" xmlns="urn:xmpp:occi-classification"
  elementFormDefault="qualified">

  <xs:element name="Classification">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="KindType" type="CategoryClassification"
          minOccurs="0" maxOccurs="1" />
        <xs:element name="MixinType" type="MixinClassification"
          minOccurs="0" maxOccurs="unbounded" />
        <xs:element name="LinkType" type="LinkClassification"
          minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="MixinClassification">
    <xs:complexContent>
      <xs:extension base="CategoryClassification">
        <xs:sequence>
          <xs:element name="applies" type="xs:string"
            minOccurs="1" maxOccurs="1"
            default="http://schema.ogf.org/occi/core#category" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="LinkClassification">
    <xs:complexContent>
      <xs:extension base="CategoryClassification">
        <xs:sequence>
          <xs:element name="relation" type="xs:string"
            minOccurs="1" maxOccurs="1"
            default="http://schema.ogf.org/occi/core#category" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

```
<xs:complexType name="CategoryClassification">
  <xs:sequence>
    <xs:element name="term" type="xs:string" minOccurs="1"
      maxOccurs="1" />
    <xs:element name="schema" type="xs:string" minOccurs="1"
      maxOccurs="1" />
    <xs:element name="title" type="xs:string" minOccurs="0"
      maxOccurs="1" />
    <xs:element ref="attributeClassification" minOccurs="0"
      maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

<xs:element name="attributeClassification">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string" minOccurs="1"
        maxOccurs="1" />
      <xs:element name="type" type="attributeType" minOccurs="1"
        maxOccurs="1" />
      <xs:element name="mutable" type="xs:boolean" minOccurs="1"
        maxOccurs="1" />
      <xs:element name="required" type="xs:boolean" minOccurs="1"
        maxOccurs="1" />
      <xs:element name="default" type="xs:string" minOccurs="0"
        maxOccurs="1" />
      <xs:element name="description" type="xs:string"
        minOccurs="0" maxOccurs="1" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
  <xs:simpleType name="attributeType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="STRING" />
      <xs:enumeration value="ENUM" />
      <xs:enumeration value="INTEGER" />
      <xs:enumeration value="FLOAT" />
      <xs:enumeration value="DOUBLE" />
      <xs:enumeration value="BOOLEAN" />
      <xs:enumeration value="URI" />
      <xs:enumeration value="SIGNATURE" />
      <xs:enumeration value="KEY" />
      <xs:enumeration value="DATETIME" />
      <xs:enumeration value="DURATION" />
      <xs:enumeration value="LIST" />
      <xs:enumeration value="MAP" />
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

LISTING A.3: Classification Schema

A.4. XML-Rendering Schema

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="urn:xmpp:occi-representation"
     xmlns="urn:xmpp:occi-representation"
  elementFormDefault="qualified">

  <xs:element name="Category_List">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Category" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

```
<xs:element name="Category">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Kind" type="CategoryType" minOccurs="0"
        maxOccurs="1" />
      <xs:element name="Mixin" type="CategoryType" minOccurs="0"
        maxOccurs="unbounded" />
      <xs:element name="Link" type="LinkType" minOccurs="0"
        maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="LinkType">
  <xs:complexContent>
    <xs:extension base="CategoryType">
      <xs:sequence>
        <xs:element name="target" type="xs:anyURI" minOccurs="0"
          maxOccurs="1" />
        <xs:element name="Mixin" type="CategoryType" minOccurs="0"
          maxOccurs="unbounded" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="CategoryType">
  <xs:sequence>
    <xs:element name="term" type="xs:string" minOccurs="1"
      maxOccurs="1" />
    <xs:element name="schema" type="xs:string" minOccurs="1"
      maxOccurs="1" />
    <xs:element name="title" type="xs:string" minOccurs="0"
      maxOccurs="1" />
    <xs:element name="attribute" type="AttributeType"
      minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="AttributeType">
  <xs:choice>
    <xs:element name="STRING" type="xs:string" minOccurs="1"
      maxOccurs="1" />
    <xs:element name="ENUM" type="xs:string" minOccurs="1"
      maxOccurs="1" />
    <xs:element name="INTEGER" type="xs:int" minOccurs="1"
      maxOccurs="1" />
    <xs:element name="FLOAT" type="xs:float" minOccurs="1"
      maxOccurs="1" />
    <xs:element name="DOUBLE" type="xs:double" minOccurs="1"
      maxOccurs="1" />
    <xs:element name="BOOLEAN" type="xs:boolean" minOccurs="1"
      maxOccurs="1" />
    <xs:element name="URI" type="xs:anyURI" minOccurs="1"
      maxOccurs="1" />
    <xs:element name="SIGNATURE" type="xs:base64Binary"
      minOccurs="1" maxOccurs="1" />
    <xs:element name="KEY" type="xs:base64Binary" minOccurs="1"
      maxOccurs="1" />
    <xs:element name="DATETIME" type="xs:dateTime" minOccurs="1"
      maxOccurs="1" />
    <xs:element name="DURATION" type="xs:duration" minOccurs="1"
      maxOccurs="1" />
    <xs:element name="LIST" type="ListType" minOccurs="1"
      maxOccurs="1" />
    <xs:element name="MAP" type="MapType" minOccurs="1"
      maxOccurs="1" />
  </xs:choice>
  <xs:attribute name="name" type="xs:string" use="required" />
</xs:complexType>

<xs:complexType name="ListType">
  <xs:sequence>
    <xs:element name="item" type="xs:string"
      minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="MapType">
  <xs:sequence>
    <xs:element name="item" type="MapItem"
      minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
```

```
  <xs:complexType name="MapItem">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="key" type="xs:string" use="required" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

</xs:schema>
```

LISTING A.4: XML-Rendering Schema

A.5. EventLog Schema

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

<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
  targetNamespace='urn:xmpp:eventlog' xmlns='urn:xmpp:eventlog'
  elementFormDefault='qualified'>

  <xs:element name='log'>
    <xs:complexType>
      <xs:sequence>
        <xs:element name='message' type='xs:string' minOccurs='1'
          maxOccurs='1' />
        <xs:element name='tag' minOccurs='0' maxOccurs='unbounded'>
          <xs:complexType>
            <xs:attribute name='name' type='xs:string' use='required' />
            <xs:attribute name='value' type='xs:string' use='required' />
            <xs:attribute name='type' type='xs:QName' use='optional'
              default='xs:string' />
          </xs:complexType>
        </xs:element>
        <xs:element name='stackTrace' type='xs:string' minOccurs='0'
          maxOccurs='1' />
      </xs:sequence>
      <xs:attribute name='timestamp' type='xs:dateTime' use='required' />
      <xs:attribute name='id' type='xs:string' use='optional' />
      <xs:attribute name='type' type='EventType' use='optional'
        default='Informational' />
      <xs:attribute name='level' type='EventLevel' use='optional'
        default='Minor' />
```

```
      <xs:attribute name='object' type='xs:string' use='optional' />
      <xs:attribute name='subject' type='xs:string' use='optional' />
      <xs:attribute name='facility' type='xs:string' use='optional' />
      <xs:attribute name='module' type='xs:string' use='optional' />
    </xs:complexType>
  </xs:element>

  <xs:simpleType name='EventType'>
    <xs:restriction base='xs:string'>
      <xs:enumeration value='Debug' />
      <xs:enumeration value='Informational' />
      <xs:enumeration value='Notice' />
      <xs:enumeration value='Warning' />
      <xs:enumeration value='Error' />
      <xs:enumeration value='Critical' />
      <xs:enumeration value='Alert' />
      <xs:enumeration value='Emergency' />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name='EventLevel'>
    <xs:restriction base='xs:string'>
      <xs:enumeration value='Major' />
      <xs:enumeration value='Medium' />
      <xs:enumeration value='Minor' />
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

LISTING A.5: EventLog Schema

A.6. Evaluation Recording

```
<iq to="exchange.intercloud.cit.tu-berlin.de" id="3BHdb-14" type="get"
    from="alex0@intercloud.cit.tu-berlin.de/Smack">
  <resource_type xmlns="urn:xmpp:rest-xwadl" path="/agreement/testSLA"/>
</iq>
```

LISTING A.6: IQ payload of an XWADL Get

```
<iq type="result" id="3BHdb-14" from="exchange.intercloud.cit.tu-berlin.de"
    to="alex0@intercloud.cit.tu-berlin.de/Smack">
  <resource_type xmlns="urn:xmpp:rest-xwadl"
      xmlns:urn="urn:xmpp:occi-classification" path="/agreement/testSLA">
    <documentation title="Summary">This resource allows for managing
       service level agreements.</documentation> <grammars>
      <urn:Classification>
        <urn:KindType>
          <urn:term>agreement</urn:term>
          ...
        </urn:KindType>
        ...
      </urn:Classification>
    </grammars>
    <method type="PUT">
      <documentation title="addGuaranteeTerm">This method adds a set of
         guarantee terms as link resources.</documentation>
      <request mediaType="xml/occi"/>
      <response mediaType="text/uri-list"/>
    </method>
    ...
    <method type="GET">
      <documentation title="getRepresentation">This method returns the
         representation of this resource.</documentation>
      <response mediaType="xml/occi"/>
    </method>
    <method type="GET">
      <documentation title="getSubResources">This method returns a list
          of all sub resources.</documentation>
      <response mediaType="text/uri-list"/>
    </method>
  </resource_type>
</iq>
```

LISTING A.7: IQ payload of an XWADL Result

```
<iq to=”exchange.intercloud.cit.tu−berlin.de” id=”3BHdb−17” type=”set”
    from=”alex0@intercloud.cit.tu−berlin.de/Smack”>
  <resource xmlns=”urn:xmpp:xml−rest” path=”/agreement/testSLA”>
    <method type=”PUT”>
      <request mediaType=”xml/occi”>
        <representation><![CDATA[<Category
            xmlns="urn:xmpp:occi-representation">
  <Link>
    <term>guaranteeterm</term>
    <schema>http://schema.cit.tu-berlin.de/occi/sla#</schema>
    <title>GuaranteeTerm Link</title>
    <attribute name="occi.guaranteeterm.sensor">
      <STRING>xmpp://wally080.cit.tu-berlin.de#/sensor/senX1</STRING>
    </attribute>
    <attribute name="occi.guaranteeterm.state">
      <ENUM>undefined</ENUM>
    </attribute>
    <attribute name="occi.guaranteeterm.relationaloperator">
      <ENUM>GREATER_THAN_OR_EQUAL_TO</ENUM>
    </attribute>
    <target>xmpp://wally080.cit.tu-berlin.de#/compute/vmX1</target>
    <Mixin>
      <term>timewindowmetric</term>
      <schema>http://schema.cit.tu-berlin.de/occi/cep
      #</schema>
      <title>Time Window Metric Mixin</title>
      <attribute name="occi.metric.windowtype">
        <ENUM>SlidingWindow</ENUM>
      </attribute>
      <attribute name="occi.metric.windowduration">
        <INTEGER>15</INTEGER>
      </attribute>
      <attribute name="occi.metric.durationunit">
        <ENUM>minutes</ENUM>
      </attribute>
    </Mixin>
    <Mixin>
      <term>eventlog</term>
      <schema>http://schema.cit.tu-berlin.de/occi/cep
      #</schema>
      <title>Event Log Mixin</title>
      <attribute name="occi.eventlog.eventid">
        <STRING>AvailabilityEvent</STRING>
      </attribute>
```

```
    </Mixin>
    <Mixin>
      <term>aggregation</term>
      <schema>http://schema.cit.tu-berlin.de/occi/cep
      #</schema>
      <title>Aggregation Mixin</title>
      <attribute name="occi.aggregation.operation">
        <ENUM>avg</ENUM>
      </attribute>
    </Mixin>
    <Mixin>
      <term>availability</term>
      <schema>http://schema.cit.tu-berlin.de/occi/sla/guaranteeterm
      #</schema>
      <title>Availability Mixin</title>
      <attribute name="occi.guaranteeterm.availability.slo">
        <DOUBLE>50.0</DOUBLE>
      </attribute>
    </Mixin>
  </Link>
</Category>]]></representation>
      </request>
      <response mediaType="text/uri-list"/>
    </method>
  </resource>
</iq>
```

LISTING A.8: IQ payload of a REST-XML Set

```
<iq type="result" id="3BHdb-17" from="exchange.intercloud.cit.tu-berlin.de"
    to="alex0@intercloud.cit.tu-berlin.de/Smack">
  <resource xmlns="urn:xmpp:xml-rest" path="/agreement/testSLA">
    <method type="PUT">
      <response mediaType="text/uri-list">
        <representation>xmpp://exchange.intercloud.cit.tu-berlin.de#
        /agreement/testSLA/413382d9-f0db-4425-a8eb-41b69cd807d5;
        </representation>
      </response>
    </method>
  </resource>
</iq>
```

LISTING A.9: IQ payload of a REST-XML Result

```
select avg(availability) from AvailabilityEvent.win:time(15
   minutes) where object = 'xmpp://wally080.cit.tu-berlin.de#/sensor/senX1' and
   subject = 'xmpp://wally080.cit.tu-berlin.de#/compute/vmX1' having
   avg(availability)>= 50.0
```

LISTING A.10: Generated expression for fulfilled guarantee terms

```
select avg(availability) from AvailabilityEvent.win:time(15
   minutes) where object = 'xmpp://wally080.cit.tu-berlin.de#/sensor/senX1' and
   subject = 'xmpp://wally080.cit.tu-berlin.de#/compute/vmX1' having
   avg(availability)< 50.0
```

LISTING A.11: Generated expression for violated guarantee terms

```
<message to="exchange.intercloud.cit.tu-berlin.de" id="3BHdb-19" type="normal"
   from="alex0@intercloud.cit.tu-berlin.de/Smack">
  <log xmlns="urn:xmpp:eventlog"
     object="xmpp://wally080.cit.tu-berlin.de#/sensor/senX1"
     subject="xmpp://wally080.cit.tu-berlin.de#/compute/vmX1"
     timestamp="2015-12-01T01:24:59.222+01:00" id="AvailabilityEvent">
    <tag xmlns:xs="http://www.w3.org/2001/XMLSchema" name="availability"
       type="xs:double" value="45.37660022765001"></tag>
  </log>
</message>
```

LISTING A.12: Message payload of an EventLog