Autonomic SLA Management in Federated Computing Environments

Pawel Rubach
Computer Science, Texas Tech University
SORCER Research Group
Lubbock, USA
pawel.rubach@sorcersoft.org

Mike Sobolewski
Computer Science, Texas Tech University
SORCER Research Group
Lubbock, USA
sobol@sorcersoft.org

Abstract—**Federated computing environments offer requestors the ability to dynamically invoke services offered by collaborating providers in the virtual service network. Without an efficient resource management, however, the assignment of providers to customer's requests cannot be optimized and cannot offer high reliability without relevant SLA guarantees. We propose a new SLA-based SERViceable Metacomputing Environment (SERVME) capable of matching providers based on QoS requirements and performing autonomic provisioning and deprovisioning of services according to dynamic requestor needs. This paper presents the new autonomic SLA management and the object-oriented SLA model for large-scale service-oriented systems. An initial reference implementation in the SORCER environment is also described.**

Keywords-Service Oriented Computing; Metacomputing; Resource Management; Quality of Service; Service Level Agreements

I. Introduction

Many research activities worldwide are focused on developing smart, self-manageable systems that will allow applications to run smoothly and reliably in a distributed environment. IBM calls this *Autonomic Computing* [1]. The realization of this concept would enable the move towards *Utility Computing* – the long awaited vision where computing power would be available as a utility just like water or electricity is delivered to our homes today. One of the challenges in addressing this concept lies in the problem of guaranteeing a certain level of *Quality of Service* (QoS) to the customer for which he/she would be willing to pay.

In this paper we address related issues by proposing the SERViceable Metacomputing Environment (SERVME) which is based on the SORCER [2] environment extended by adding a QoS Management Framework. This paper presents an architecture overview of the federated P2P environment and focuses on the aspects of autonomic provisioning of required services within the environment.

SORCER provides a way of creating service-oriented programs and executing them in a metacomputing environment. The service-oriented paradigm is a distributed computing concept wherein objects across the network play their predefined roles as service providers. Service requestors can access these providers by passing messages called service exertions. An exertion defines how the service providers federate among themselves to supply the requestor with a required service collaboration. All these services form an instruction-set of a virtual metacomputer that looks to the end-user as a single computer.

The proposed QoS management framework has been deployed and validated in the SORCER environment. However, due to its generic nature we believe that both the *Service Level Agreements* (SLA) object model as well as the underlying communication model defined in terms of communication interfaces could be adopted for other service-oriented architectures.

The rest of the paper is divided into the following sections: Section II describes the related work, Section III gives introduction to SORCER, Section IV describes service messaging with exertions and QoS requirements, Section V presents the SLA object model, Section VI elaborates on the architecture of the SERVME framework and presents its autonomic provisioning features, Section VII presents the deployment of the framework and Section VIII concludes the paper.

II. Related Work

Much research has been done in the area of *Service Level Agreements* (SLA) management of services.

At first in *Grid Computing* the *Globus Architecture for Reservation and Allocation* (GARA) [3] addressed the SLA management issues of services. Further work challenged the problem of complex, multi-level SLA management and led to the development of a generic *Service Negotiation and Acquisition Protocol* (SNAP) [4].

As grid technology started to move from traditional network batch queuing towards the application of *Web Services* (WS) the research of the grid community, as well as others, concentrated on integrating SLA management into the standard stack of WS. The grid community developed the *Open Grid Service Architecture* (OGSA) [5] that stated the need to address SLA management.

As a means to allow dynamic SLA negotiation, efforts have been made to standardize the SLA specification. Some general architectural approaches can be observed that are usually taken to define an SLA specification. The most common approach uses mathematical formalization or creates a specific language to define QoS and SLAs.

One of the examples is [6] where the authors propose a specification language, similar to C, called Contract Definition Language (CDL), another one, presented in [7] is an object oriented language: *QoS Modelling Language* (QML). A slightly different approach is taken by C. Yang et al [8] who suggest to specify QoS requirements in the natural language and then convert them to UML or high level programming languages. A general overview and a comparison of many SLA specification languages is provided in [9] by J. Jin and K. Nahrstedt.

Another group of specifications that mostly concentrates on defining SLAs for WS create an XML schema and use XML as the representation of QoS parameters and SLAs. Most notable of them are the *Web Service Level Agreement* framework (WSLA) [10] and the *WS-Agreement* specification [11]. The latter, however, has a very limited ability to specify conditional expressions and alternatives. Therefore extensions have been proposed in the *CoreGRID* project [12] for example. Recently the developments within the NextGRID project introduce new QoS parameters such as i.a. Robustness and Resilience [20]. Apart from agreed standards such as the above there are also a number of custom solutions such as the *WS-QoS framework* [13] that in turn proposes to specify QoS parameters and provision prices in the WS protocol stack within the WSDL specification. The most recent tendency is to use ontologies to specify SLAs [14].

All the above mentioned solutions propose a kind of mathematical formalism or specific language semantics to describe QoS parameters and SLAs. The presented approach follows a different path and focuses on defining an object-oriented specification in terms of communication interfaces as abstract data types. Although the reference implementation is realized in Java the APIs are modeled generally to allow it to be utilized in any modern object-oriented language. The SLA specification provides an open framework that can be extended and implemented to meet the requirements of custom environments. We claim that this approach offers greater flexibility than i.e. using XML while preserving a richer and simpler degree of expressiveness and allows a more direct and more efficient implementation than i.e. using ontologies by eliminating the laborious intermediary conversion steps.

The SERVME framework concentrates on federated, distributed environments. Although, there have been several projects that claim to address SLA management in federated environments [6], [15], however, none of them refers to federation of services specified and created on-the-fly such as the one which exists in the *exertion-oriented architecture* utilized by SERVME. Most mentioned above projects use the term *federation* to underline the organizational challenges that arise due to the fact that a federated system is composed of usually static services that may belong to different administrative entities.

Most of the described SLA management research focuses on traditional grids or web/grid service architectures and little attention is drawn to federated metacomputing environments. The dynamic aspects as well as the P2P characteristics of those environments pose new challenges for resource management and this paper tries to address some of them.

In federated service-oriented environments services can be divided into two categories: 1) infrastructure services - those that provide basic functionality of the environment (Lookup Service, Transaction Manager, filesystem services, authentication and authorization services, QoS management services etc.) and 2) application services that are custom built for every application of the platform. (An example called `QosCaller` is mentioned in Section VII. This service was used during the validation of SERVME to invoke a legacy application used for Magnetic Resonance Image (MRI) processing.) Resource management for the first group can be handled using provisioning frameworks such as the Rio project [16] [17], however, in this research we propose a new dynamic, autonomic provisioning of the application services to allow the metacomputing environment to find existing services that satisfy requested QoS requirements or provision services with the requested QoS automatically on request and deprovision them when they are not used anymore.

The presented solution aims at delivering a complete, extensible framework, however, this paper provides only a general overview and focuses mainly on the SLA object model and autonomic management at this time.

III. SORCER

SORCER [2] (Service Oriented Computing EnviRonment) is a federated service-to-service (S2S) metacomputing environment that treats service providers as network objects with well-defined semantics of a federated service object-oriented architecture. It is based on Jini [17] semantics of services in the network and Jini programming model with explicit leases, distributed events, transactions, and discovery/join protocols. While Jini focuses on service management in a networked environment, SORCER focuses on exertion-oriented programming and the execution environment for exertions [2], SORCER uses Jini discovery/join protocols to implement its *exertion-oriented architecture* (EOA) using *federated method invocation* [18] [19], but hides all the low-level programming details of the Jini programming model.

In EOA, a service provider is an object that accepts remote messages from service requestors to execute a collaboration. These messages are called service exertions and describe *service (collaboration) data, operations* and collaboration's *control strategy.* An *exertion task* (or simply a *task*) is an elementary service request, a kind of an elementary instruction executed by a single service provider or a small-scale federation for the same service data. A composite exertion called an *exertion job* (or simply a *job*) is defined hierarchically in terms of tasks and other jobs, and thus isa kind of a federated procedure executed by a large-scale federation. The executing exertion is dynamically bound to all required and currently available service providers on the network. This collection of providers identified in runtime is called the *exertion federation*. The federation provides the implementation for the collaboration as specified by its exertion. When the federation is formed, each exertion's operation has its corresponding method (code) available on the network. Thus, the network *exerts* the collaboration with the help of the dynamically formed service federation. In other words, we send the request onto the network implicitly, not to a particular service provider explicitly.

The overlay network of service providers is called the *service grid* and an exertion federation is in fact a *virtual metacomputer*. The metainstruction set of the metacomputer consists of all operations offered by all service providers in the grid. Thus, an *exertion-oriented* (EO) program is composed of *metainstructions* with its own *control strategy* and a *service context* representing the metaprogram data. The service context describes the collaboration data that tasks and jobs work on. Each service provider offers services to other service peers on the object-oriented overlay network. These services are exposed *indirectly* by operations in well-known public remote interfaces and considered to be elementary (tasks) or compound (jobs) activities in EOA. Indirectly means here, that you cannot invoke any operation defined in provider's interface directly. These operations can be specified in the requestor's exertion only, and the exertion is passed by itself on to the relevant service provider via the top-level `Servicer` interface implemented by all service providers called *servicers*—service peers. Thus all service providers in EOA implement the
`service(Exertion, Transaction) : Exertion`
operation of the `Servicer` interface. When the servicer accepts its received exertion, then the exertion's operations can be invoked by the servicer itself, if the requestor is authorized to do so. Servicers do not have mutual associations prior to the execution of an exertion; they come together at runtime (federate) for a collaboration as defined by its exertion. In EOA requestors do not have to lookup for any network provider at all, they can submit an exertion, onto the network by calling
`Exertion.exert(Transaction : Exertion`
on the exertion. The `exert` operation will create a required federation that will run the collaboration as specified in the EO program and return the resulting exertion back to the exerting requestor. Since an exertion encapsulates everything needed (data, operations, and control strategy) for the collaboration, all results of the execution can be found in the returned exertion's service contexts.

Domain specific servicers within the federation, or task peers (*taskers*), execute task exertions. *Rendezvous* peers (jobbers and spacers) coordinate the execution of job exertions. Providers of the `Tasker`, `Jobber`, and `Spacer` type are three of SORCER main infrastructure servicers.

IV. Service Messaging and Exertions

In object-oriented terminology, a message is the single means of passing control to an object. If the object responds to the message, it has an operation and its implementation (method) for that message. Because object data is encapsulated and not directly accessible, a message is the only way to send data from one object to another. Each message specifies the name (identifier) of the receiving object, the name of operation to be invoked, and its parameters. In the unreliable network of objects, the receiving object might not be present or can go away at any time. Thus, we should postpone receiving object identification as late as possible. Grouping related messages per one request for the same data set makes a lot of sense due to network invocation latency and common errors in handling. These observations lead us to service-oriented messages called exertions.

To further clarify what an exertion is, an exertion consists mainly of three parts: a set of service signatures, which is a description of operations in a collaboration, the associated service context upon which to execute the exertion, and the control strategy (default provided) that defines how signatures are applied in the collaboration. A service signature specifies at least the provider's interface that the service requestor would like to use and a selected operation to run within that interface. There are four types of signatures that can be used for an exertion: `PREPROCESS`, `PROCESS`, `POSTPROCESS`, and `APPEND`. An exertion must have one and only one `PROCESS` signature that specifies what the exertion should do and who works on it. An exertion can optionally have multiple `PREPROCESS`, `POSTPROCESS`, and `APPEND` signatures that are primarily used for formatting the data within the associated service context. A service context consists of several data nodes used for either input, output, or both. A task may work with only a single service context, while a job may work with multiple service contexts since it can contain multiple tasks. The programmer can define a control strategy as needed for the underlying exertion by choosing relevant exertion types and configuring attributes of service signatures and service contexts accordingly [19].

In SERVME a signature includes a QoS Context (defined in Section V) that encapsulates all QoS/SLA data. Different types of control exertions (`IfExertion`, `ForExertion`, and `WhileExertion`) can be used to define flow of control that can also be configured additionally with adequate signature attributes [18].

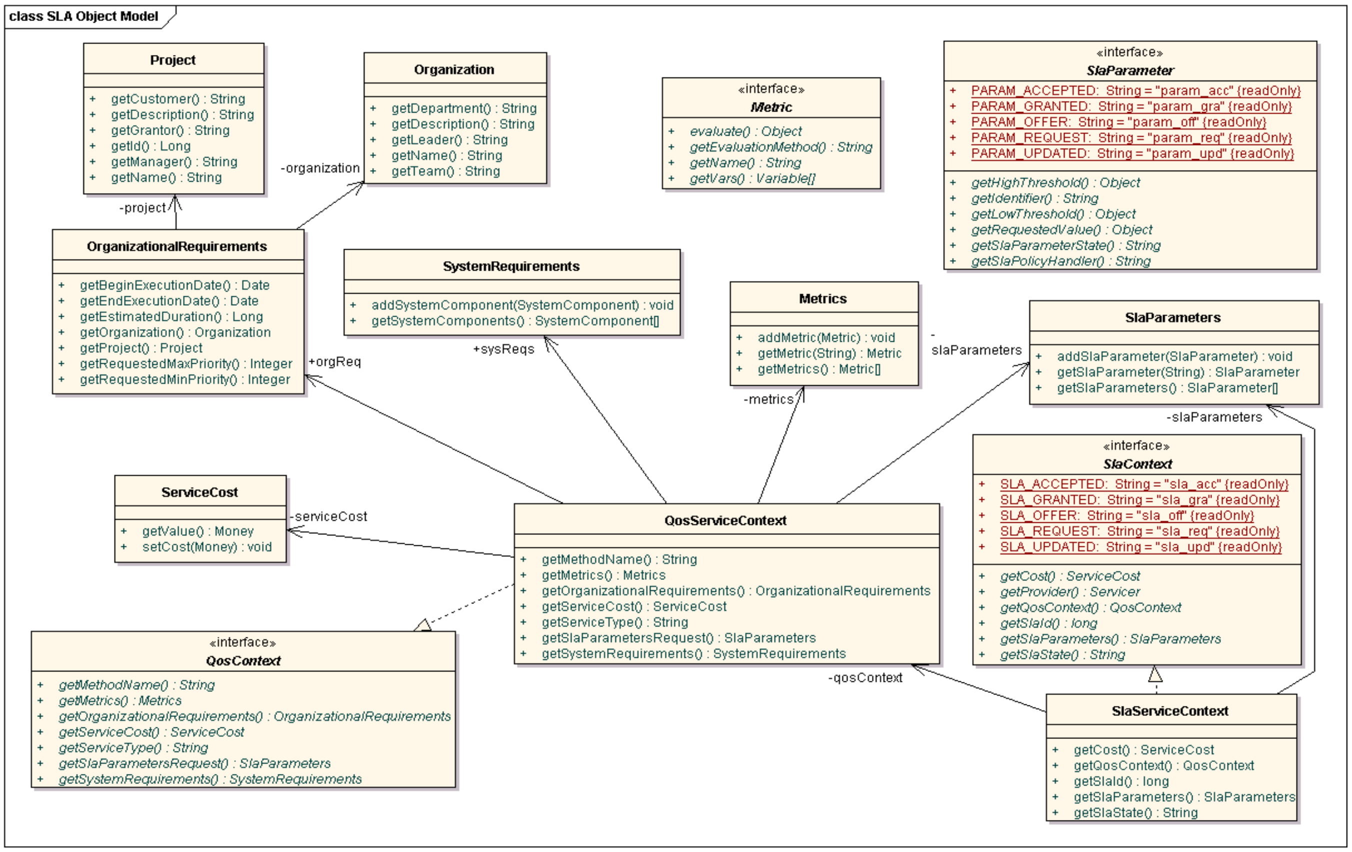
class SLA Object Model
Project
+ getCustomer() : String
+ getDescription() : String
+ getGrantor() : String
+ getId() : Long
+ getManager() : String
+ getName() : String
Organization
+ getDepartment() : String
+ getDescription() : String
+ getLeader() : String
+ getName() : String
+ getTeam() : String
«interface» Metric
+ evaluate() : Object
+ getEvaluationMethod() : String
+ getName() : String
+ getVars() : Variable[]
«interface» SlaParameter
+ PARAM_ACCEPTED: String = "param_acc" {readOnly}
+ PARAM_GRANTED: String = "param_gra" {readOnly}
+ PARAM_OFFER: String = "param_off" {readOnly}
+ PARAM_REQUEST: String = "param_req" {readOnly}
+ PARAM_UPDATED: String = "param_upd" {readOnly}
+ getHighThreshold() : Object
+ getIdentifier() : String
+ getLowThreshold() : Object
+ getRequestedValue() : Object
+ getSlaParameterState() : String
+ getSlaPolicyHandler() : String
-organization
-project
OrganizationalRequirements
+ getBeginExecutionDate() : Date
+ getEndExecutionDate() : Date
+ getEstimatedDuration() : Long
+ getOrganization() : Organization
+ getProject() : Project
+ getRequestedMaxPriority() : Integer
+ getRequestedMinPriority() : Integer
SystemRequirements
+ addSystemComponent(SystemComponent) : void
+ getSystemComponents() : SystemComponent[]
Metrics
+ addMetric(Metric) : void
+ getMetric(String) : Metric
+ getMetrics() : Metric[]
SlaParameters
+ addSlaParameter(SlaParameter) : void
+ getSlaParameter(String) : SlaParameter
+ getSlaParameters() : SlaParameter[]
+orgReq
+sysReqs
-metrics
-slaParameters
-slaParameters
«interface» SlaContext
+ SLA_ACCEPTED: String = "sla_acc" {readOnly}
+ SLA_GRANTED: String = "sla_gra" {readOnly}
+ SLA_OFFER: String = "sla_off" {readOnly}
+ SLA_REQUEST: String = "sla_req" {readOnly}
+ SLA_UPDATED: String = "sla_upd" {readOnly}
+ getCost() : ServiceCost
+ getProvider() : Servicer
+ getQosContext() : QosContext
+ getSlaId() : long
+ getSlaParameters() : SlaParameters
+ getSlaState() : String
ServiceCost
+ getValue() : Money
+ setCost(Money) : void
-serviceCost
QosServiceContext
+ getMethodName() : String
+ getMetrics() : Metrics
+ getOrganizationalRequirements() : OrganizationalRequirements
+ getServiceCost() : ServiceCost
+ getServiceType() : String
+ getSlaParametersRequest() : SlaParameters
+ getSystemRequirements() : SystemRequirements
«interface» QosContext
+ getMethodName() : String
+ getMetrics() : Metrics
+ getOrganizationalRequirements() : OrganizationalRequirements
+ getServiceCost() : ServiceCost
+ getServiceType() : String
+ getSlaParametersRequest() : SlaParameters
+ getSystemRequirements() : SystemRequirements
-qosContext
SlaServiceContext
+ getCost() : ServiceCost
+ getQosContext() : QosContext
+ getSlaId() : long
+ getSlaParameters() : SlaParameters
+ getSlaState() : String

Figure 1. SLA Object Model

An exertion can be invoked by calling exertion's `exert` operation:
`Exertion.exert(Transaction) : Exertion`,
where a parameter of the `Transaction` type is required when the transactional semantics is needed for all participating nested exertions within the parent one, otherwise can be `null`. Thus, EO programming allows us to submit an exertion onto the network and to perform executions of exertion's signatures on various service providers indirectly, but where does the service-to-service communication come into play? How do these services communicate with one another if they are all different? Top-level communication between services, or the sending of service requests (exertions), is done through the use of the generic `Servicer` interface and the operation `service` that all SORCER services are required to provide—`Servicer.service(Exertion, Transaction)`. This top-level service operation takes an exertion as an argument and gives back an exertion as the return value.

V. SLA Object Model

SERVME builds on the SORCER environment by extending its interfaces and adding new service providers. It is a generic resource management framework based on the *Commonality-Variability Analysis* model in terms of common data structures and extensible communication interfaces hiding all implementation details.

One of the key features of the framework is the proposed SLA specification that has been specifically developed to meet the requirements of metacomputing environments. It is defined in object-oriented terms and thus forms an object model. The class diagram in Fig. 1 shows the elements of the SLA object model. For better readability all setter methods and most attributes are omitted and thus only getter methods are presented.

The `QosContext` interface defines the data structure that incorporates all requirements submitted by the requestor in the exertion's signature. It includes:

- *Functional Requirements*—a service type identifying a requested provider (the classname of the required interface: `sorcer.provider.QosCaller`, for example), operation to be executed (`getMethodName()`), and related provider's attributes,
- *System Requirements*—fixed properties that describe the requested provider's hardware and software environment (i.e. CPU architecture, OS name and version, required libraries etc.). Each requirement is defined in terms of the `SystemComponent` class based on the class by the same name in the Rio project [16] that defines the basic properties and may be customized for every component.

- *Organizational Requirements*—properties of the submitting entity (`getProject()` and `getOrganization()`), requested priority range, requested execution time frame and estimated duration of the execution. Those parameters are used by the `SlaPrioritizer` service (described in section VI) to manage and assign resources to projects or organizational entities in large computing environments.
- *Metrics*—dynamic, user defined, compound parameters which are calculated on the basis of System- or Organizational Requirements. Each metric must implement the class `Metric` that defines its name, set of variables and the method `evaluate()` that performs on-the-fly evaluation based on input variables. As variables, names of `SystemComponent`s and their attributes may be passed and then they will be substituted during evaluation with their current values. Current reference implementation includes the Groovy Metric, for example, that allows to specify custom expressions in the Groovy language.
- *Service Cost*—(i.e. Maximum cost of the execution). SERVME defines the `ServiceCost` class that includes the `getValue()` method. This way customized implementations of complex cost specification algorithms are supported.
- *SLA Parameter Requests*—the demanded ranges of values or fixed values of QoS parameters, Metrics or Organizational and System Requirements. Each of them is defined using an implementation of the `SlaParameter` interface.

The `QosContext` interface is implemented by the `QosServiceContext` class.

The `SlaParameter` interface is used both to specify requests as well as offers, each regarding one specific metric-, organizational- or system requirement. `SlaParameter` defines the requested value (in case it should be fixed) or low and high thresholds. It also specifies the `SlaPolicyHandler` class that may be added to define actions (notifications, penalties etc.) invoked during execution when the contracted parameter values are breached.

`SlaParameter` specifies also the `SlaParameterState` that can have one of the enumerated values: `PARAM_REQUEST, PARAM_UPDATED, PARAM_OFFER, PARAM_ACCEPTED, PARAM_GRANTED.` This attribute is used primarily during the negotiation phase.

Another critical interface is the `SlaContext` that is implemented by the `SlaServiceContext` class and used by the service provider to offer or guarantee the required `QosContext. SlaContext` consists of:

- `SlaParameters`—set of objects that implement the `SlaParameter` interface and define the SLA Parameter ranges or values offered or guaranteed by the provider.
- `QosContext`—the related requestor's QoS requirements satisfied by the `SlaParameters` and specified as `QosContext` type.
- Offered price of the proposed SLA (`getCost()`)
- `SlaState`—property that defines the state of the negotiation process (`SLA_REQUEST,SLA_UPDATED, SLA_OFFER,SLA_ACCEPTED, SLA_GRANTED`).
- `Servicer`—(`getProvider()`) - the proxy of the service provider that guarantees the SLA.

VI. Architecture of SERVME

A. SERVME Components

Along with the above SLA object model SERVME defines basic components and communication interfaces as depicted in the UML component diagram illustrated in Fig. 2. We distinguish two forms of autonomic provisioning: monitored and on-demand. In monitored provisioning the provisioner (Rio Provisioner [16]) deploys a requested collection of providers, then monitors them for presence and in the case of any failure in the collection, the provisioner makes sure that the required number of providers is always on the network as defined by a provisioner's deployment descriptor. On-demand provisioning refers to a type of provisioning (On-demand Provisioner) where the actual provider is presented to the requestor, once a subscription to the requested service is successfully processed. In both cases, if services become unavailable, or fail to meet processing requirements, the recovery of those service providers to available compute resources is enabled by Rio provisioning mechanisms. The basic components are defined as follows:

- `QosProviderAccessor` is a component used by the service requestor (customer) that is responsible for processing the exertion request containing `QosContext` in its signature. If the exertion type is `Task` then `QosCatalog` is used, otherwise a relevant rendezvous peer: `Jobber`, `Spacer` is used.
- `QosCatalog` is an independent service that acts as an extended Lookup Service (QoS LUS). The `QosCatalog` uses the functional requirements as well as related non-functional QoS requirements to find a service provider from currently available in the network. If a matching provider does not exist, the `QosCatalog` may provision the needed one by calling the `On-demandProvisioner` (described below).
- `SlaDispatcher` is a component built into each service provider. It performs two roles. On one hand, it is responsible for retrieving the actual QoS parameter values from the operating system in which it is running, and on the other hand, it exposes the interface used by

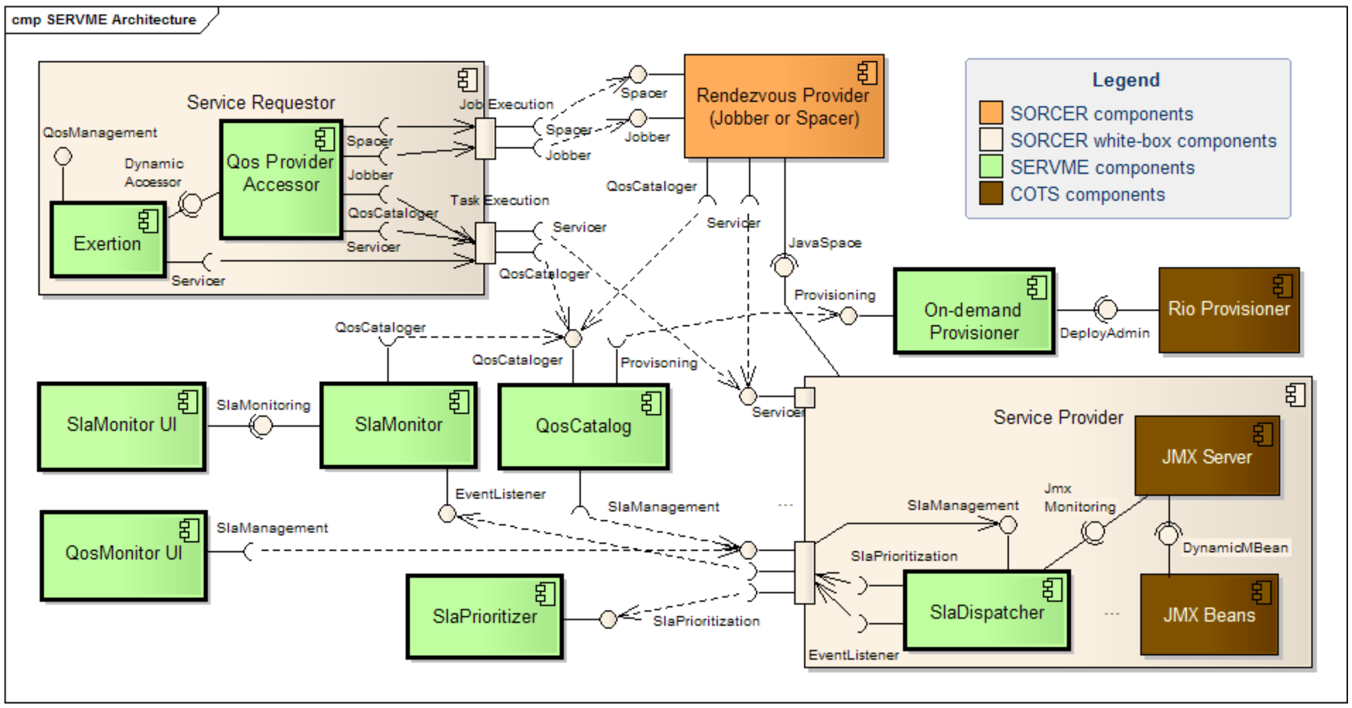

Figure 2. SERVME Components Diagram

`QosCatalog` to negotiate, sign and manage the SLA with its provider.

- `SlaPrioritizer` is a component that allows controlling the prioritization of the execution of exertions according to the organizational requirements of `SlaContext`. It allows to specify a resource allocation strategy either by simply allowing/disallowing certain projects or organizational entities to execute on certain resources or by using a “managed” free market economy approach and manipulating execution price parameters depending on organizational requirements.
- `QosMonitor` (UI) provides an embedded GUI that allows the monitoring of provider’s QoS parameters at runtime.
- `SlaMonitor` is an independent service that acts as a registry for negotiated SLA contracts and exposes the user interface (UI) for administrators to allow them to monitor, update or cancel active SLAs.
- `On-demandProvisioner` is a SERVME provider that enables on-demand provisioning of services in cooperation with the Rio Provisioner [16][17]. The `QosCatalog` uses it when no matching service provider can be found that meets requestor QoS requirements.

The SERVME framework is integrated directly into the federated metacomputing environment. As described in Section IV, the service requestor submits the exertion with QoS requirements (`QosContext`) into the network by invoking `Exertion.exert` operation. If the exertion is of `Task` type, then `QosProviderAccessor` via `QosCalatog` finds in runtime a matching service provider with a corresponding SLA. If the SLA can be directly provided then the contracting provider approached by the `QosCalatog` returns it in the form of `SlaContext`, otherwise a negotiation can take place for the agreeable `SlaContext` between the requestor and provider. The provider's `SlaDispatcher` drives this negotiation in cooperation with `SlaPrioritizer` and the requestor. If the task contains multiple signatures then the provider is responsible for contracting SLAs for all other signatures of the task before the SLA for its `PROCESS` signature is guaranteed.

However, if the submitted exertion is of `Job` type, then `QosProviderAccessor` via `QosCalatog` finds in runtime a matching rendezvous provider with a guaranteed SLA.

Before the guaranteed SLA is returned, the rendezvous provider recursively acquires SLAs for all component exertions as described above depending on the type (`Task` or `Job`) of component exertion.

B. *SLA Lifecycle*

The negotiation process is shortly described in this subsection. The details along with an activity diagram will be presented in future papers.

1) *Preparing for the negotiation*

As described above upon the execution of `Exertion.exert()`, `QosCatalog` is called either by the `QosProviderAccessor` or by one of the rendezvous providers. The QosCatalog acts as a QoS broker between the requestor and service providers. At first QosCatalog analyzes the QoS requirements passed in the QosContext and extracts the system requirements as well as functional requirements of the requested provider type. Based on the functional requirements QosCatalog performs a standard lookup and retrieves a list of all providers offering the requested interface and method. Next, QosCatalog queries via the SlaManagement interface the SlaDispatcher of each of those providers to retrieve the current QoS that it can offer. The supplied data allows it to select providers that match the system requirements. Those providers are then called via the same SlaManagement interface to start the SLA negotiation process.

2) Negotiation

The exact details of the negotiation rules and the algorithms used by both the requestor and the provider are not of the main focus of this research. SERVME defines the common negotiation protocol and the negotiation business logic is customized by each provider according to specific needs. SERVME however, specifies the common data structures defined in Section V and communication interfaces presented in Fig. 2.

When the QosCatalog invokes the negotiateSla operation of the SlaManagement interface then the provider creates the SlaContext object and includes the QosContext that contains the original QoS requirements passed as a parameter. The provider sets the SlaState to SLA_OFFER or to SLA_UPDATED depending whether it can guarantee the requested QoS requirements. At this time the provider allocates the requested resources.

SERVME introduces a SLA leasing mechanism to address the problem of unnecessary resource reservations that may occur if the requestor discontinues the negotiation process without notifying the provider – in case of a system failure, for example.

3) On-demand Provisioning

If any of the providers queried by QosCatalog responds with an SLA_OFFER, the process continues on to the signing of the contract, otherwise if only SLA_UPDATEs are returned, the QosCatalog tries to deploy a new provider with the required QoS parameters by calling the OnDemandProvisioner.

OnDemandProvisioner constructs on-the-fly an OperationalString required by Rio and calls the ProvisionMonitor component of Rio [16] to deploy the required providers. Then QosCatalog invokes the same negotiation sequence to sign the SLA with one of them.

If provisioning still does not supply a relevant service provider a failure exception containing the SlaContext is thrown by the QosCatalog and returned to the requestor. The requestor may implement negotiation handlers that process such an exception and continues the negotiation with lowered requirement's threshold values.

4) SLA Signing

Digital signing of an offered SLA is performed in two steps. First the QosCatalog chooses the best offer (i.e. based on price or other criteria) and then passes the offered SLA, its lease and the chosen provider's proxy to the requestor. The leases from other providers are aborted. The requestor is now responsible for renewing the lease and finalizing the acquiring of the SLA by calling the signSla operation on the provider. To guarantee the non-repudiation of contracts or offers both parties use the SORCER security framework based on PKI infrastructure.

5) SLA Monitoring and Management

At this point the signed SLA is also passed on to the SlaMonitor via notify in its EventListener interface. The received SLA is then registered and persisted. The SlaMonitor allows the administrator to manage and monitor SLAs that have been negotiated and signed.

6) Deprovisioning services

Thanks to the leasing mechanism the provider knows when its resources are not needed anymore. When the lease expires the provider notifies the AutonomicProvisioner and this service undeploys the unused provider by calling the Rio ProvisionMonitor. The provider cannot just simply destroy itself since in that case Rio's failover mechanism would immediately deploy another instance of the provider.

VII. Deployment

SERVME has been deployed and successfully tested within the SORCER environment. The reference implementation was written in Java 1.6 and requires Jini 2.1 and Rio 4.0-M1. The Rio runtime has been used for provisioning as well as a source of QoS parameters and also as intermediary between the framework's components and the underlying service provider's JMX Server. The Rio's Service UI has been integrated into the SERVME service provider's UI and so, it allows the user to view and monitor QoS parameters at runtime.

The framework was validated in a real-world example taken from neuroscience. SERVME was used to invoke and control multiple parallel and sequential computations that dealt with the processing of MRIs of human brains. Six heterogeneous hosts (different hardware and OS) where used to perform several simultaneous computations. Each provider, called QosCaller collected historical execution times for similar computations and used this data to calculate the estimated time and cost of execution. Cost was calculated as inversely proportional to time of execution extended with some parameters that altogether caused that running the computation on faster hardware was much more expensive than on lower end hosts.

The simulations where run several times and have shown that with SERVME it is possible to optimize the execution of complex computations for lowest price or best performance. The overhead time resulting from the communication needed to select the appropriate provider, perform SLA negotiation, and sign the SLA contract has been measured in this environment at around 1-1.5 seconds and as such is negligible in comparison to the computations run, that took minimally 3-4 minutes each.

Detailed validation results along with a complete statistical analysis will be published in a forthcoming paper on performance analysis.

VIII. Conclusions

The new Autonomic SLA Management architecture for federated, metacomputing environments is presented in this paper. SERVME introduces the new QoS/SLA object model defined by the two generic interfaces: `QosContext` and related `SlaContext` along with supporting service providers: `QosCatalog`, `SlaDispatcher`, `SlaMonitor`, `SlaPrioritizer`, and `On-demandProvisioner`. To the best of our knowledge this is the first attempt to define a framework capable of autonomic service provisioning for *exertion-oriented programming.*

The presented framework addresses the challenges of spontaneous federations in SORCER and allows for better resource allocation (best performance or lowest cost). Also, SERVME provides for better hardware utilization due to Rio monitored provisioning and SORCER on-demand provisioning. The presented architecture scales very well with on-demand provisioning that reduces the number of compute resources to those presently required for collaborations defined by corresponding exertions. When diverse and specialized hardware is used, SERVME provides means to manage the prioritization of tasks according to the organization's strategy that defines "who is computing what and where".

Two zero-install and friendly user graphical interfaces attached to SLA Monitor and SORCER Servicer are available for administration purposes.

The SERVME providers are SORCER Servicers so additional providers can be dynamically provisioned if needed autonomically. Finally, the framework enables the accounting of resource utilization based on dynamic cost metrics and thus it contributes towards the realization of the *utility computing* concept.

Acknowledgment

This work was partially supported by Air Force Research Lab, Air Vehicles Directorate, Multidisciplinary Technology Center, the contract number F33615-03-D-3307, Service-Oriented Optimization Toolkit for Distributed High Fidelity Engineering Design Optimization.

We would like to thank Dennis Reedy - the architect of project Rio for his invaluable assistance that helped us to understand the details of the Rio framework.

References

[1] J.O. Kephart & D.M. Chess, "The vision of autonomic computing," Computer, vol. 36, 2003, pp. 41-50.

[2] M. Sobolewski, "Metacomputing with Federated Method Invocation", Engineering Computer Science and IT, In-Tech, ISBN978-953-7619-32-9, 2009

[3] I. Foster, A. Roy, & V. Sander, "A quality of service architecture that combines resource reservation and application adaptation," Quality of Service, 2000. IWQOS. 2000 Eighth International Workshop on, 2000, pp. 181-188.

[4] K. Czajkowski, I. Foster, C. Kesselman, V. Sander, & S. Tuecke, "SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems," Job Scheduling Strategies for Parallel Processing, 2002, pp. 153-183.

[5] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, & R. Subramaniam, The open grid services architecture, version 1.0, 2005.

[6] P. Bhoj, S. Singhal, & S. Chutani, "SLA management in federated environments," Computer Networks, vol. 35, 2001, pp. 5-24.

[7] S. Frolund & J. Koistinen, "Quality-of-service specification in distributed object systems," Distributed Systems Engineering, vol. 5, 1998, pp. 179-202.

[8] C. Yang, B.R. Bryant, C.C. Burt, R.R. Raje, A.M. Olson, & M. Auguston, "Formal Methods for Quality of Service Analysis in Component-Based Distributed Computing," Journal of Integrated Design & Process Science, vol. 8, 2004, pp. 137-149.

[9] L. Jin, V. Machiraju, & A. Sahai, Analysis on Service Level Agreement of Web Services, Technical Report HPL-2002-180, HP Laboratories, 2002.

[10] H. Ludwig, A. Keller, A. Dan, R.P. King, & R. Franck, "Web service level agreement (WSLA) language specification," IBM Corporation, 2003.

[11] A. Andrieux, K. Czajkowski, A.D. Ibm, K. Keahey, H.L. Ibm, T.N. Nec, J.P. Hp, J.R. Ibm, S. Tuecke, & M. Xu, Web Services Agreement Specification (WS-Agreement), 2007.

[12] W. Ziegler, P. Wieder, & D. Battre, Extending WS-Agreement for dynamic negotiation of Service Level Agreements, CoreGRID Technical Report TR-0172, Institute on Resource Management and Scheduling, 2008.

[13] M. Tian, A. Gramm, H. Ritter, & J. Schiller, "Efficient Selection and Monitoring of QoS-Aware Web Services with the WS-QoS Framework," Proceedings of the 2004 IEEE/WIC/ACM International Conference on Web Intelligence, IEEE Computer Society, 2004.

[14] G. Dobson, R. Lock, & I. Sommerville, "Quality of service requirements specification using an ontology," SOCCER Workshop, Requirements Engineering, 2005.

[15] V. Machiraju, A. Sahai, & A. van Moorsel, "Web Services Management Network: an overlay network for federated service management," Integrated Network Management, 2003. IFIP/IEEE Eighth International Symposium on, 2003, pp. 351-364.

[16] "Project Rio," http://rio.dev.java.net, accessed on March 13, 2009.

[17] "Jini architecture specification, Version 2.1," http://www.jini.org/wiki/Jini_Architecture_Specification, accessed on March 12, 2009.

[18] M. Sobolewski, "SORCER: Computing and Metacomputing Intergrid," Proc. 10th International Conference on Enterprise Information Systems, Barcelona, Spain, 2008, pp. 74-85.

[19] M. Sobolewski, "Exertion Oriented Programming," 2008, IADIS, vol. 3 no. 1, pp. 86-109, ISBN: 1646-3692.

[20] "NextGRID Project," http://www.nextgrid.org, accessed on July 2, 2009.