Dynamic SLA Negotiation in Autonomic Federated Environments

Pawel Rubach[1,2] and Michael Sobolewski[1]

[1] Computer Science, Texas Tech University, SORCER Research Group,
Box 43104 Boston & 8th, Lubbock, TX 79409, USA
{pawel.rubach, sobol}@sorcersoft.org
[2] Business Informatics, Warsaw School of Economics,
Al. Niepodleglosci 162, 02-554 Warszawa, Poland

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 that includes Dynamic SLA Negotiation, 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 SLA negotiation process that includes on-demand provisioning and uses an object-oriented SLA model for large-scale service-oriented systems supported by SERVME. An initial reference implementation in the SORCER environment is also described.

Keywords: SLA Negotiation, QoS, SLA, Metacomputing, Service-Oriented Architecture, SORCER.

1 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 Dynamic SLA Negotiation process for the SERViceable Metacomputing Environment (SERVME)[2] which is based on the SORCER (Service-Oriented Computing EnviRonment) [3] environment extended by adding a QoS Management Framework. This paper presents the SLA negotiation process including the on-demand provisioning of services and briefly describes the architecture of the federated P2P 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 SLA negotiation process has been implemented and validated as part of the SERVME framework 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.

This paper is a follow-up to [2] – the first one to describe the SERVME framework. Here the focus is the SLA life-cycle and negotiation whereas [2] concentrated on the SERVME architecture and the SLA object model.

The rest of the paper is divided into the following sections: Section 2 describes the related work, Section 3 gives introduction to SORCER, Section 4 presents the overview of SERVME, Section 5 elaborates on the SLA negotiation, Section 6 presents the deployment of the framework, and Section 7 concludes the paper.

2 Related Work

SLA negotiation has been researched extensively at first in the area of networking. Its application to services was propagated with the emergence of *Grid Computing*. At first the *Globus Resource Allocation Manager* (GRAM) [4] lacked a general negotiation protocol that was added later (as described in [5]) in form of the *Service Negotiation and Acquisition Protocol* (SNAP) [6] that addresses complex, multi-level SLA management. SNAP defines three types of SLAs: Task SLAs, Resource SLAs and Binding SLAs and provides a generic framework, however as Quan et al. [7] underline the protocol needs further extensions for its implementation to address specific problems.

As grid technology started to move from traditional network batch queuing towards the application of *Web Services* (WS) the work of the grid community as well as others focused on incorporating SLA negotiation into the stack of WS technologies. The Web Service Level Agreement framework (WSLA) [8] and the WS-Agreement specification [9] have been proposed to standardize the SLA specification. WS-Agreement specifies also basic negotiation semantics, however, allows only a simple one-phase - offer-accept/reject negotiation. More complex two- and three-phase commit protocols applied in conjunction with WS-Agreement are described in [10]. A different approach to enable automatic SLA negotiation was taken by [11] and [12] who propose to use agents for the negotiation of SLAs in grids. In [13] authors propose to introduce a meta-negotiation protocol that will allow the parties to select via negotiation the protocol used for the actual SLA negotiation.

The above solutions concentrate on traditional grids or WS architectures, however, new challenges that reach beyond the multi-phase commit protocols arise when introducing P2P resource management. Significant work has also been pursued in this area, for example by [15], however, this research does not include SLA negotiation.

A novel approach to SLA management and negotiation for P2P distributed environments where federations of services are formed on-the-fly is presented. To fully address the problems with network/resource unreliability and contract SLAs for multi-level, multiple party scenarios this paper introduces a leasing mechanism that is used in conjunction with the 2-phase commit transactional semantics.

3 SORCER

SORCER [3] 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 [16] 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 [3]. SORCER uses Jini discovery/join protocols to implement its *exertion-oriented architecture* (EOA) [18], 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 collaboration. These messages are called service exertions and describe *collaboration data, operations* and collaboration's *control strategy.* An *exertion task* (or simply a *task*) is an elementary service request, a kind of 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, a kind of 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 an *exertion federation.* The federation provides the virtual processor (metaprocessor) 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 metaprocessor.* The metainstruction set of the metaprocessor consists of all operations offered by all providers in the service grid. Thus, an *exertion-oriented* (EO) program is composed of *metainstructions* with its own *control strategy* and a *service context* representing the metaprogram data. These operations can be specified in the requestor's exertion only, and the exertion is passed by itself on to the initializing service provider (found dynamically) 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.

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

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 collaboration, the associated service context upon which to execute the exertion, and control strategy (default provided) that defines how signatures are applied in the collaboration. A service signature specifies at least the provider's service type (interface) that the service requestor would like to use and a selected operation to run within that interface. A service context consists of several data nodes used for either: input, output, or both. A task works with only a single service context, while a job may work with multiple service contexts since it can contain multiple tasks [18].

In SERVME a signature includes a QoS Context (described in Section 4.2) that encapsulates all QoS/SLA data.

4 SERVME Overview

To perform SLA negotiation one has to define: 1) a SLA negotiation protocol and interactions between components, 2) a QoS/SLA specification and 3) a negotiation strategy or a decision-making model. SERVME defines the negotiation protocol in the form of a generic communication model, its components tailored to the requirements of federated environments as well as the SLA specification in form of an object model and its data structures. A default negotiation strategy and a decision-making model is presented below, however, SERVME is designed to allow an easy customization of the negotiation business logic for each provider and requestor since in a real-world scenario of a free-market service economy these rules may decide which provider receives more requests and thus may become part of its competitive advantage and be considered confidential.

4.1 SERVME Components

SERVME builds on the SORCER environment by extending its interfaces and adding new service providers. The details of the architecture have been described in [2]. The components used in the SLA negotiation process are shortly presented below.

- `ServiceProvider` provides the requested service and has a built-in component called `SlaDispatcher` that retrieves the QoS parameters from the operating system and is responsible for the SLA management on the provider side.
- `QosCatalog` is an independent service that acts as an extended Lookup Service (QoS LUS) as well as the SLA negotiation broker between the provider and the requestor.
- `SlaPrioritizer` is a component that allows controlling the prioritization of the execution of exertions according to organizational requirements (see section 4.2)
- `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 and cancel active SLAs.
- `OnDemandProvisioner` is a SERVME provider that enables on-demand provisioning of services in cooperation with the Rio Provisioner [14] [16]. The `QosCatalog` uses it when no matching service provider can be found that meets requestor QoS requirements.

4.2 SLA Object Model

The key feature of the framework is the proposed SLA object model designed to meet the requirements of federated metacomputing environments. For a detailed description including a UML class diagram please refer to [2].

The two main artifacts are: `QosContext` and `SlaContext`. The first one groups the requirements submitted by the requestor. It contains: 1) *Functional Requirements*—a service type (interface) identifying a requested provider, operation to be executed, and related provider's attributes, 2) *System Requirements*—fixed properties that describe the requested provider’s hardware and software environment (i.e. CPU architecture, OS name and version etc.), 3) *Organizational Requirements*—properties of the submitting entity (department, team, project, requested timeframe for the execution, priority etc.), 4) *Metrics*—dynamic, user defined, compound parameters which are calculated on the basis of System- or Organizational Requirements, 5) *Service Cost*—requirements (i.e. Maximum cost of the execution) and 6) *SLA Parameter Requests*—the demanded ranges of values or fixed values of QoS parameters.

The second critical interface—`SlaContext` defines the actual SLA. It contains the related requirements in form of the `QosContext` as well as: 1) SLA Parameters offered or guaranteed by the provider 2) the offered price 3) data used to identify the provider (its ID, proxy etc.) and 4) the state of the negotiation that can have one of the enumerated values: `SLA_REQUESTED, SLA_UPDATED, SLA_OFFERED, SLA_ACCEPTED, SLA_GRANTED, SLA_ARCHIVED.`

5 SLA Negotiation

This section describes the SLA negotiation process. Fig. 1 shows how the negotiation process is integrated into the life-cycle of executing exertions. The diagram refers also to other two activity diagrams presented below: SLA Negotiation and SLA Monitoring.

5.1 Recursive Acquisition of SLAs

The negotiation sequence for a single exertion of `Task` type is presented below in detail, however for completeness in this subsection more complex exertions that require recursive SLA acquisition are shortly described.

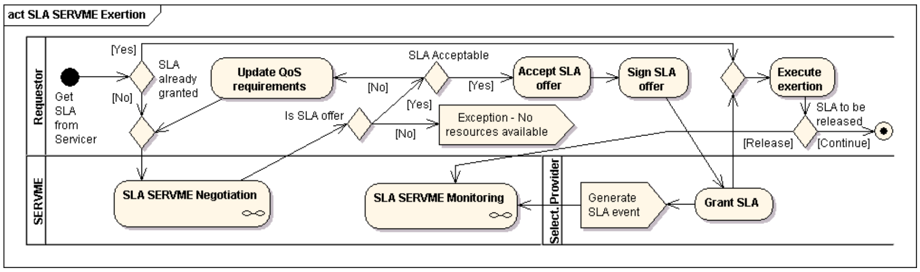

Fig. 1. Activity Diagram showing how SERVME SLA Negotiation is integrated into EOP

The execution of an exertion begins when the requestor calls `Exertion.exert()`. In case the exertion is of `Task` type the request is passed on to the `QosCalatog` that acts as a broker in the negotiation process described below. However, if the exertion is of `Job` type, then `QosCalatog` finds in runtime a matching rendezvous provider (`Jobber` or `Spacer`) with a guaranteed SLA.

Before the guaranteed SLA is returned, the rendezvous provider recursively acquires SLAs for all component exertions as described below depending on the type (`Task` or `Job`) of component exertion. To ensure transactional semantics of the SLA acquisition the rendezvous peer uses a leasing mechanism (described below) that is similar to the two-phase commit protocol defined by the Jini Transaction model.

Exertions of `Task` type may also contain multiple signatures (as explained in Section 3), so the same recursive mechanism is used to acquire the final SLA. However, in this case the requestor only receives the final SLA for the dynamically binding – signature of the `PROCESS` type.

For intelligibility in the following subsections the assumption is that the outcome of the negotiation should be a single SLA contract for a Task with only one signature.

5.2 Preliminary Selection of Providers

As depicted in Fig. 2 at first `QosCatalog` analyzes the QoS requirements passed in the `QosContext` and extracts the functional requirements (provider's interface, method, and other attributes) as well as system requirements. Based on the functional

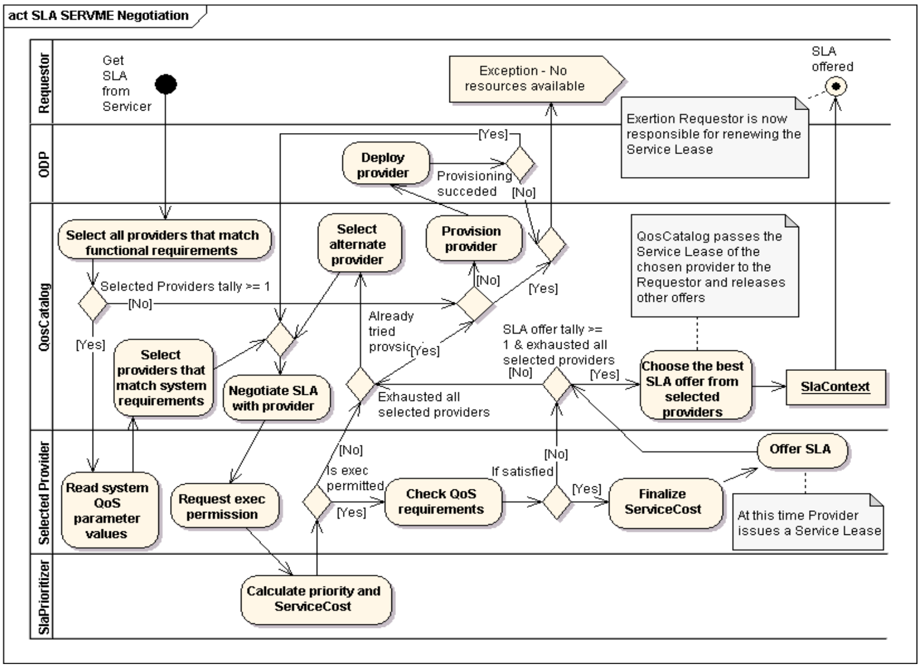
act SLA SERVME Negotiation
Requestor
ODP
QosCatalog
Selected Provider
SlaPrioritizer
Get SLA from Servicer
Exception - No resources available
SLA offered
Exertion Requestor is now responsible for renewing the Service Lease
[Yes]
Deploy provider
Provisioning succeded
[No]
Select all providers that match functional requirements
Select alternate provider
Provision provider
QosCatalog passes the Service Lease of the chosen provider to the Requestor and releases other offers
Selected Providers tally >= 1
[No]
[Yes]
[No]
[Yes]
Already tried provsio
[Yes]
SLA offer tally >= 1 & exhausted all selected providers
[No]
[Yes]
Select providers that match system requirements
Negotiate SLA with provider
Exhausted all selected providers
Choose the best SLA offer from selected providers
SlaContext
Offer SLA
[No]
Is exec permitted
[Yes]
Check QoS requirements
[No]
If satisfied
[Yes]
Finalize ServiceCost
At this time Provider issues a Service Lease
Read system QoS parameter values
Request exec permission
Calculate priority and ServiceCost

Fig. 2. Activity Diagram showing the actual SLA Negotiation

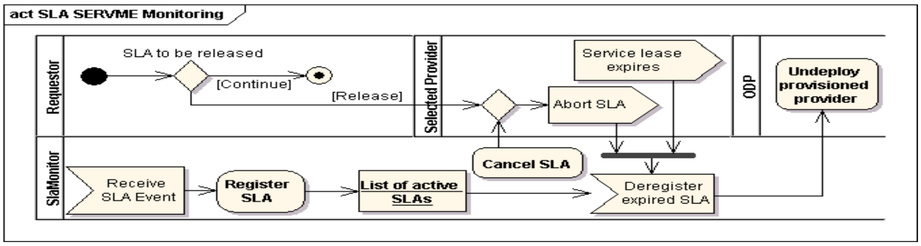

Fig. 3. Activity Diagram showing the SLA Monitoring

requirements `QosCatalog` performs a dynamic lookup and retrieves a list of all providers offering the requested interface and method. If none are found `QosCatalog` tries to provision them using the `OnDemandProvisioner` (ODP) (see subsection 5.5). Next, `QosCatalog` queries the `ServiceProvider` to retrieve the basic QoS parameters that it can offer. The supplied data allows it to select providers that match the system requirements. Those are then called via their `SlaManagement` interface to start the SLA negotiation process.

5.3 Negotiation

The negotiation is initiated by the `QosCatalog` that invokes the `negotiateSla` operation of the `SlaManagement` interface of the provider. In the first step the provider extracts the organizational requirements from the `QosContext` and passes them to the `SlaPrioritizer` where the exertion's organizational properties are evaluated against strategic rules defined by the management in the `SlaPrioritizer` service. The provider then receives a permission or denial to execute the exertion and optionally a cost parameter that it may use to calculate the final service cost of the offer. In case no permission is given the provider returns a no-go exception and the `QosCatalog` has to select an alternate provider or autonomically provision one if no others are available. After locating another provider the negotiation sequence is repeated for that provider.

In case the permission is given the provider checks the QoS requirements against its current resource utilization and allocations for other concurrently guaranteed SLAs. If a parameter can be guaranteed the provider copies the corresponding `SlaParameter` object including the requested threshold values from the `QosContext`'s SLA parameter requests to `SlaContext`'s SLA parameters and sets its state to `PARAM_OFFERED`. However, if the requirement cannot be fulfilled the corresponding SLA parameter request is also copied to `SlaContext` but its state is set to `PARAM_UPDATE` and its threshold range is updated to the maximum/minimum offered value. After processing individual parameters the provider sets the state of the whole `SlaContext` to `SLA_OFFERED` if all SLA parameters can be guaranteed or `SLA_UPDATED` otherwise.

In case the QoS requirements can be met the provider calculates the estimated service cost, allocates the offered resources and creates a `Lease` that is attached to the SLA offer. This `Lease` has a short expiration time and thus guarantees that the resources are not blocked unnecessarily. Before the exertion is finally executed the

`Lease` must be renewed by the requestor to extend the life of the SLA. The estimated cost in the validation case, for example, is calculated on the basis of historical executions with similar input data on the same host. Cost is inversely proportional to time of execution extended with some parameters that altogether causes that running computations on faster hardware is much more expensive than on lower-end hosts.

To guarantee the non-repudiation of contracts or offers the provider uses the SORCER security framework based on PKI infrastructure to sign the SLA offer before passing it on to the `QosCatalog`.

The described negotiation sequence is repeated by the `QosCatalog` for all providers that initially matched the system requirements. Out of all offers the `QosCatalog` chooses the best one depending on the specified parameters and passes it to the requestor for acceptance and signing (see Fig. 1). Currently time only or cost only optimizations are supported but the inclusion of non-linear optimization methods that will allow to select a set of offers matching both parameters (i.e., fastest execution but costing no more than X) are a work in progress.

5.4 SLA Acceptance and Signing

The requestor may now decide to accept or deny the received offer. However, in case it is denied the SLA negotiation process has to be reinitiated from the very beginning. In case of acceptance the requestor updates the SLA's state to `SLA_ACCEPTED` and performs digital signing using the PKI infrastructure.

From now on the requester is responsible for renewing the `Lease` of the SLA.

The requester calls the `signSla` method of the provider and passes the `SlaContext`. If the `Lease` has not expired the provider grants the SLA by setting its state to `SLA_GRANTED`. The `SlaContext` is then returned to the requestor and the execution of the exertion may finally begin.

At the same time the provider sends a copy of the `SlaContext` asynchronously to the `SlaMonitor` where it is registered and persisted.

5.5 On-Demand Provisioning

SERVME reduces the overall resource utilization by allowing service providers to be provisioned on-demand and deprovisioned when they are not used anymore.

In the above negotiation process there are three scenarios that may lead to on-demand provisioning: 1) when no providers are available that meet functional requirements 2) when none of the available providers receive a permission to execute the exertion from the `SlaPrioritizer` and 3) when none of the SLA offers returned by providers to the `QosCatalog` fully fulfills the requirements (all have a state of negotiation set to `SLA_UPDATED`).

In any of these cases the `QosCatalog` tries to deploy a new provider with the required QoS parameters by calling the `OnDemandProvisioner` object. `OnDemandProvisioner` constructs on-the-fly an `OperationalString` required by Rio and calls the `ProvisionMonitor` component of Rio [14] to deploy the required providers. If the provisioning succeeds `QosCatalog` invokes the same negotiation sequence on the newly provisioned provider. Otherwise `QosCatalog` returns to the requestor the full list of SLAs that it negotiated, none of which however, fully fulfills the requestors requirements. The requestor may now choose to accept one of these offers or try to start another round of negotiation with lowered QoS requirements.

5.6 SLA Monitoring and Management

As depicted in Fig. 3 the `SlaMonitor` can be used to monitor the execution and delete an active SLA. It communicates with providers and asynchronously receives messages with updated states of the SLA's lifecycle.

5.7 Deprovisioning Services

The leasing mechanism described in subsection 5.3 ensures that the provider is aware when any of the granted SLAs expires or the exertion simply finishes execution. This information is passed on to the `SlaMonitor` that also receives events regarding the provisioning actions taken by the `OnDemandProvisioner`. `SlaMonitor` is thus able to detect situations when the provisioned provider is not used anymore. In that case it notifies the `OnDemandProvisioner` and this service undeploys the unused provider by calling the Rio's `ProvisionMonitor`. The provider cannot just simply destroy itself upon finishing the execution of the exertion since in that case Rio's failover mechanism would immediately deploy another instance of that provider.

6 Deployment

SERVME has been deployed in the SORCER environment. 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 where used to perform several simultaneous computations. The simulations were 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, performing SLA negotiation, and signing 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.

7 Conclusions

The new SLA Negotiation process for Autonomic Federated Metacomputing Environments is presented in this paper. The described process includes the on-demand provisioning of services and refers to components defined in the SERVME framework: `QosCatalog, SlaDispatcher, SlaMonitor, SlaPrioritizer`, and `OnDemandProvisioner`. The negotiation uses the SLA object model introduced in SERVME and defined by the two generic interfaces: `QosContext` and related `SlaContext`. To the best of our knowledge this is the first attempt to describe the SLA negotiation process for *exertion-oriented programming*.

The presented framework addresses the challenges of spontaneous federations in SORCER and allows for better resource allocation. 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 graphical user interfaces attached to SLA Monitor and SORCER Servicer are available for administration purposes.

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

Acknowledgments. 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 also to thank Dennis Reedy the architect of project Rio for his invaluable assistance that helped us to integrate the Rio provisioning framework with the SERVME framework.

References

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

[2] Rubach, P., Sobolewski, M.: Autonomic SLA Management in Federated Computing Environments. In: Proceedings of the 2009 International Conference on Parallel Processing Workshops (ICPPW 2009). IEEE Computer Society, Los Alamitos (in press, 2009)

[3] Sobolewski, M.: SORCER: Computing and Metacomputing Intergrid. In: 10th International Conference on Enterprise Information Systems, Barcelona, Spain (2008)

[4] Czajkowski, K., Foster, I., Karonis, N., Kesselman, C., Martin, S., Smith, W., Tuecke, S.: A resource management architecture for metacomputing systems. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1998, SPDP-WS 1998, and JSSPP 1998. LNCS, vol. 1459, pp. 62–82. Springer, Heidelberg (1998)

[5] Czajkowski, K., Foster, I., Kesselman, C., Tuecke, S.: Grid service level agreements: Grid resource management with intermediaries. In: Grid resource management: state of the art and future trends, pp. 119–134. Kluwer Academic Publishers, Dordrecht (2004)

[6] Czajkowski, K., Foster, I., Kesselman, C., Sander, V., Tuecke, S.: SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 153–183. Springer, Heidelberg (2002)

[7] Quan, D.M., Kao, O.: SLA Negotiation Protocol for Grid-Based Workflows. In: High Performance Computing and Communcations, pp. 505–510 (2005)

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

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

[10] Pichot, A., Wieder, P., Waeldrich, O., Ziegler, W.: Dynamic SLA-negotiation based on WS-Agreement, CoreGRID Technical Report TR-0082, Institute on Resource Management and Scheduling (2007)

[11] Shen, W., Li, Y.: Adaptive negotiation for agent-based grid computing. In: Proceedings of the Agentcities/Aamas 2002, vol. 5, pp. 32–36 (2002)

[12] Ouelhadj, D., Garibaldi, J., MacLaren, J., Sakellariou, R., Krishnakumar, K.: A Multi-agent Infrastructure and a Service Level Agreement Negotiation Protocol for Robust Scheduling in Grid Computing. In: Sloot, P.M.A., Hoekstra, A.G., Priol, T., Reinefeld, A., Bubak, M. (eds.) EGC 2005. LNCS, vol. 3470, pp. 651–660. Springer, Heidelberg (2005)

[13] Brandic, I., Venugopal, S., Mattess, M., Buyya, R.: Towards a Meta-Negotiation Architecture for SLA-Aware Grid Services, Technical Report, GRIDS-TR-2008-9, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia (2008)

[14] Project Rio, `http://rio.dev.java.net/` (accessed on March 13, 2009)

[15] Cao, J., Kwong, O.M.K., Wang, X., Cai, W.: A peer-to-peer approach to task scheduling in computation grid. Int. J. Grid Util. Comput. 1, 13–21 (2005)

[16] Jini architecture specification, Version 2.1 (accessed on March 2009)

[17] Sobolewski, M.: Federated Method Invocation with Exertions. In: Proceedings of the 2007 IMCSIT Conference, pp. 765–778. PTI Press (2007)

[18] Sobolewski, M.: Exertion Oriented Programming. In: IADIS, vol. 3, pp. 86–109 (2008)