Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Networks (PDCN 2011)
February 15 - 17, 2011 Innsbruck, Austria

SLA-BASED ASYNCHRONOUS COORDINATION IN METACOMPUTING ENVIRONMENTS

Pawel Rubach
Department of Business Informatics
Warsaw School of Economics
Al. Niepodległości 162
02-554 Warszawa, Poland
email: pawel.rubach@sgh.waw.pl

ABSTRACT
Federated metacomputing environments allow requesters to dynamically invoke services offered by collaborating providers in the virtual service network. They evolve into metaoperating systems that form an intermediary layer between metaprograms - programs written in terms of other programs - and the virtual metacomputer composed of collaborating services. These systems require resource management to efficiently handle the assignment of providers to customer's requests and to offer high reliability and SLA guarantees. This paper presents the SLA-based asynchronous coordination algorithm developed within the recently proposed SERViceable Metacomputing Environment (SERVME) capable of matching providers based on QoS requirements and performing on-demand provisioning of services according to dynamic requester needs. The performance analysis realized on a use case of protein structure prediction shows that the new algorithm allows us to optimize the usage of resources according to cost/time priorities and may also shorten the overall execution time in comparison to state-of-the-art methods.

KEY WORDS
Metacomputing, Tuple Spaces, SLA Negotiation, QoS, SLA, Service-Oriented Architecture

1 Introduction

In the late 1990s grid computing emerged as a solution to utilize distributed resources and recently the IT industry has proposed cloud computing as a way to achieve greater computational power that would lead towards the realization of the utility computing concept. Since every vendor has a different definition of the cloud, there is a lot of confusion and, apart from making use of virtualization, cloud computing does not introduce particularly many new solutions at the conceptual level of distributed computing. One of the reasons is that similarly to grids, also in the case of clouds, the way programmers write programs has not changed. They are still written to be executed on a single computer. As a result, it is clear that new concepts that reach beyond virtualized single computer platforms must be developed.

One of such concepts is envisioned in the idea of *federated metacomputing*. The goal is to create a new layer of abstraction on top of current grids, clouds and single computer platforms, one that, in contrast to today's solutions, introduces a new paradigm that changes the way programs are written. It is represented by the concept of *metaprograms*, that is, programs written in terms of other programs. These lower-level programs are in fact dynamic object-oriented services. Metaprograms are not executed by particular nodes of a cluster or a grid but instead by a network of service providers that federate dynamically to create a virtual metacomputer and dissolve after finalizing the execution.

Since the introduction of UNIX, operating systems (OS) evolved as an intermediary between the hardware and the user and his/her applications. It is the role of the OS to locate required libraries, allocate resources and execute a requested program while controlling the hardware and allowing others to concurrently use the computer. The same should apply to the virtual metacomputer and metaprograms. There is clearly a need to create this intermediary layer: a virtual *metaoperating system* (MOS).

The foundations for the concepts of *metaprogramming* and metaoperating system were laid in the FIPER project (Federated Intelligent Product EnviRonment) that was realized in between 2000 and 2003 under the sponsorship of NIST. This work evolved into the SORCER project (Service ORiented Computing EnviRonment) that was led by Michael Sobolewski at Texas Tech University and is continued today at the Air Force Research Laboratory, Dayton, Ohio and at the Polish-Japanese Institute of Information Technology in Warsaw, Poland.

SORCER introduced the concept of a metaprogram and named it *exertion* as well as defined the architecture and basic system services that allow exertions to be executed by the virtual metacomputer. Although several projects concentrated on, for example, creating a distributed file system or on issues related to security, the SORCER metacomputing environment still could not be regarded as a real metaoperating system because it lacked one of the crucial elements of an operating system: *resource management*.

This lacking element has been lately proposed in

DOI: 10.2316/P.2011.719-024

the form of the SERViceable Metacomputing Environment (SERVME). Previous papers [11], [12] introduced the architecture, the SLA object model and focused on autonomic SLA management and negotiation while this paper addresses the recently developed SLA-based coordination algorithm based on tuple spaces and shows how this algorithm performs in a real-world use case.

The next sections of this paper are organized as follows: Section 2 introduces the SORCER metaoperating system. Section 3 focuses on related research. Section 4 concentrates on the proposed coordination algorithm. Section 5 presents briefly the deployment. Section 6 discusses the performance and analyzes the experiments that involve the use of the proposed algorithm. Finally, Section 7 concludes the paper.

2 SORCER Metaoperating System

SORCER 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 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 [14] and the execution environment for exertions. SORCER uses Jini discovery/join protocols to implement its exertion-oriented architecture (EOA) using federated method invocation [13], but hides all the low-level programming details of the Jini model.

In EOA, a service provider is an object that accepts remote messages from service requesters to execute a collaboration. These messages are called *exertions* and describe service (collaboration) data, operations and the collaboration's control strategy. An *exertion task* (or simply a task) is an elementary service request, a kind of elementary federated 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, it is a kind of a federated procedure executed by a large-scale federation. The execution of an exertion job requires coordination algorithms, in particular, the algorithm proposed in this paper is essential when QoS guarantees are required. 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. 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 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 network. 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. Indirectly means here, that you cannot invoke any operation defined in the provider's interface directly. These operations can be specified in the requester'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. Servicers do not have mutual associations prior to the execution of an exertion; they come together dynamically (federate) for a collaboration as defined by its exertion. In EOA requesters 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 requester. Since an exertion encapsulates everything needed for the collaboration, all results of the execution can be found in the returned exertion's service contexts.

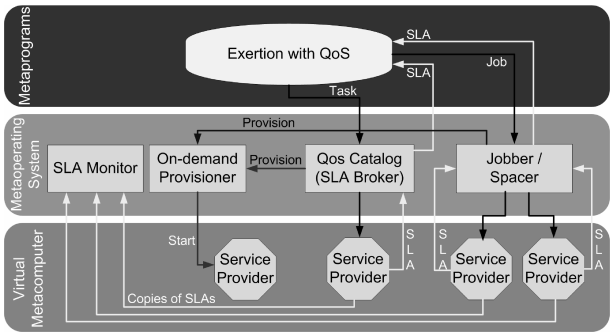

Figure 1. SERVME Conceptual Architecture

Once the exertion execution is complete, the federation dissolves and the providers disperse to seek other collaborations to join. SORCER supports also a traditional approach to grid computing where the business logic comes from the service requester's executable codes that seek compute resources on the network.

Just like a regular OS, such as UNIX, SORCER has a number of system services that enable other providers to function in the environment. Rendezvous peers (*Jobbers* and *Spacers*) coordinate the execution of exertion jobs. The spacer was recently updated to accomodate the new coordination algorithm and other infrastructure services were defined within SERVME to handle resource management. As a result SERVME forms, in fact, the resource manage-

Table 1. Grid computing vs. federated metacomputing

	Grid Computing	Federated Metacomputing
Architecture	Centralized/Client Server	Peer-2-Peer (Service-2-Service)
Task scheduling	Schedulers and queues	No schedulers or queues - resources are assigned dynamically at runtime from all available providers in the network
Managed by	Central resource manager /scheduler	Decentralized coordinating service invoked per request (many instances may run simultaneously)
Configuration	Static using DNS or IP addresses	Dynamic using discovery/join protocols
Remote invocation	RPC over web services/SOAP	Federated Method Invocation – executing metaprograms

ment module of the MOS. It is responsible for: SLA negotation, prioritization, on-demand provisioning and monitoring of the execution of exertions and it consists of the following services: *QosCatalog*, *SlaMonitor*, *SlaPrioritizer* and *OnDemandProvisioner*. The conceptual architecture of SERVME is presented in Figure 1. Due to space limitations a complete architecture and details regarding the aforementioned components as well as the SLA object model that defines QoS parameters is omitted in this paper, however, can be found in [11].

3 Related Work

Much research has been done in the area of resource allocation in distributed environments, in particular, on the Service Level Agreements (SLA) management of services. However, most approaches focus either on low-level resource allocation in clustered environments or on grids where computing tasks are assigned to particular nodes by a centralized scheduler. This work introduces a new approach to resource management, where all resources available in the network are treated together and thus form the virtual metacomputer that is capable of executing metaprograms. Since this approach uses a decentralized distributed zero-configuration architecture where federations of service providers are created on-the-fly during the execution time, this technique poses new challenges and requires new algorithms and a new architecture to manage its resources in an efficient and scalable way.

A detailed comparison of the new approach with grid technologies is presented in Table 1.

Grid computing introduced a significant number of architectures, standards and protocols. The initial problems of complexity were to a large extent addressed by the introduction of grid/web services (WS) that allowed to solve some integration issues by standardizing the communication protocol in the form of XML and SOAP. However, as Sobolewski points out [15], the “one size fits all” approach is not always the best choice, especially in the case of scientific applications where often vast amounts of data must be exchanged between services and, parsing the messages, adds a lot of overhead. Another problem with WSs is their lack of object orientation (interface inheritence, polimorphism, reflection etc.) and as a consequence lack of behavioral transfer. Most current grids focus on using BPEL or similar WS assembly techniques to invoke remote services. BPEL can be used to pass data and control flow between services, however, as it is built on top of WSs it suffers from the same problems. Therefore, in contrast to the proposed metaprogramming concept it cannot be treated as a full-fledged programming language for distributed applications. Besides, BPEL is not really decentralized - the process has to run on a concrete engine and, as many practitioners admit, these engines often become bottlenecks. In contrast, in the proposed architecture additional coordinating peers may be started on-demand and metaprograms will automatically migrate using load-balancing techniques such as the one presented in this paper, for example.

Although grid technologies foster the use of networks of compute nodes instead of single supercomputers, the programming methods have not changed. As a result, most applications are still written for use on a single computer and grid schedulers are used to move executable code to currently available hosts. As Sobolewski points out in [15] this approach “...is reminiscent of batch processing in the era when operating systems were not yet developed. A series of programs ("jobs") is executed on a computer without human interaction or the possibility to view any results before the execution is complete.”

Apart from the SORCER project other Jini-based metacomputing approaches were proposed by: Sunderam et al. [16] and Johasz et al. [7], however the literature shows no evidence of any of them creating a complete metaoperating system and, in particular, of defining the concept of metaprogramming.

One of the key challenges of a distributed system is the coordination method. A number of different techniques were researched by the grid community. For example, Chao et al. [2] propose a centralized method using BPEL whereas Eymann et al.[3] and Cao et al. [1] concentrate on a decentralized agent-based approach. Another notable example is the work of Favarim et al. [4] who propose to use the tuple-space model for coordination. The last approach seems most promising since its space and time independence allow for late-binding and disconnected operation - qualities that fit well to the decentralized service-2-service nature of the SORCER environment. However, Favarim et al. concentrate on traditional grids and do not include the support for a full QoS/SLA negotiation in their approach.

The coordination algorithm proposed in this paper follows the same idea known as space-based computing or tuple spaces and introduced by David Gelerntner in 1985 [6]. Since then this technique was researched extensively and implemented in java in form of the JavaSpaces. Several extensions were proposed. For example, Khushraj et al. [8] introduced semantic tuples and Kuhn et al. propose

the Extensible Tuple Model [9].

The proposed coordination technique was developed specifically to address the coordination of complex metaprograms that require QoS guarantees. To the best of my knowledge this is the first attempt to combine a space-based approach with SLA contracts for the execution of metaprograms within a distributed system.

4 Coordination Algorithms

SORCER uses two types of job coordination techniques. The explicit *Push* type is one, where tasks within the exertion job are executed (pushed towards providers) by the Jobber that calls their `exert` method sequentially or in parallel depending on the control strategy. The coordination algorithm that extends this technique by introducing QoS guarantees and SLA negotiation for *Push* jobs was presented in a previous paper by Rubach and Sobolewski [12].

A more advanced coordination technique mostly suitable for larger exertions uses the JavaSpaces. This approach is implemented by the Spacer rendezvous peer and is referred to as the *Pull* type since the actual tasks are pulled from the shared space by relevant service providers.

The basic handling of *Pull* jobs (without QoS) in SORCER consists of the following steps. The spacer service analyzes the job and for every inner exertion (both tasks and jobs) it creates an envelop object that describes the exertion to be executed. The envelop's fields used for matching contain the required provider's service type (interface) and a flag that defines the current state of the execution. This envelop is dropped to the space, where it is matched against templates prepared previously by service providers. Those templates contain their service types. This allows service providers to receive from the space exertions that they are able to execute. Whenever the envelop matches the template, it is taken from the space by the provider that owned the matched template to avoid a situation where a different provider executes the same exertion simultaneously. When the execution is finished the envelop is written back to the space by the provider but its state is updated to allow the spacer coordinating peer to match the executed envelops. Spacer collects envelops of executed exertions and returns the results to the requester.

The algorithms used to coordinate *Pull* jobs are in reality more complicated due to the requirements of robustness and thus the introduction of transactional semantics and security. These issues, however, were addressed previously.

The introduction of SLA management to the execution of *Pull* jobs substantially changes the assumptions and the control flow of the execution of *Pull* jobs. In this case, the space is also used to match exertions to service providers. However, the space is only used during the first stage when SLA offers are acquired and the actual execution (the second stage) is invoked directly by the spacer service. Consequently, the execution does not pass through the space. This change has many implications on the coordination algorithms.

4.1 SLA Negotiation in Space Computing

An overview of the SLA negotiation process for *Pull* jobs is presented in Figure 2. The control flow illustrated in this diagram starts when the top-level exertion (*Pull* job) is executed by the requester and is passed to the Spacer service for coordination. Spacer selects independent inner exertions (1) and for every one of them creates an SLA envelop (described in details in Section 4.5) and writes it to the space (2). SLA envelops are matched against templates created earlier by service providers. Those providers that fulfill functional requirements (offer the requested service type) read the corresponding envelops (3) and start the process of issuing an SLA offer. At first they call the *SlaPrioritizer* service and request a permission to execute and the assigned priorities (4). If a permission is given (5) the provider matches its current QoS parameter values against QoS requirements retrieved from the envelop (6) and based on the outcome creates an SLA offer (if all QoS requirements are met) or proposes an updated SLA contract (otherwise). Next the provider calculates the final estimated cost of the execution (7) and writes its offer to the space (8) by appending it to the distributed array (see Section 4.6) designated for this exertion.

The QoS parameters matched before issuing an SLA offer are devided into 3 groups: System Requirements (i.e. number and model of CPUs, OS type and version etc.), SLA Parameter Requests (CPU usage, available memory/diskspace) and Metrics (user-defined based on other QoS parameters). The details concerning the SLA object model were described in: [11].

In the meantime the Spacer monitors the SLA offers written by providers to the space and uses the algorithm presented below to decide how to process them. This algorithm has three parameters specified individually for each exertion: MIN_OFFERS, REPEAT_TIMES and TIMEOUT.

```
I = 0;
While I < REPEAT_TIMES Do:
 Begin
   START_TIME = currentTime();
   While currentTime() - START_TIME < TIMEOUT And
        NUM_OFFERS() < MIN_OFFERS Do:
        readSLAOffersFromSpace();
   End;
   If NUM_OFFERS >= MIN_OFFERS Then Do:
        selectBestOffer();
   Else
        tryToProvisionProvider();
   If PROVISION_SUCCEED And I = REPEAT_TIMES - 1 Then
        I = I -1;
   I++;
  End;
Done;
```

As shown above, the Spacer service monitors the number of available SLA offers for a given exertion and waits until this number is greater or equal to the MIN_OFFERS parameter specified in the configuration or until the elapsed time reaches the TIMEOUT parameter. If

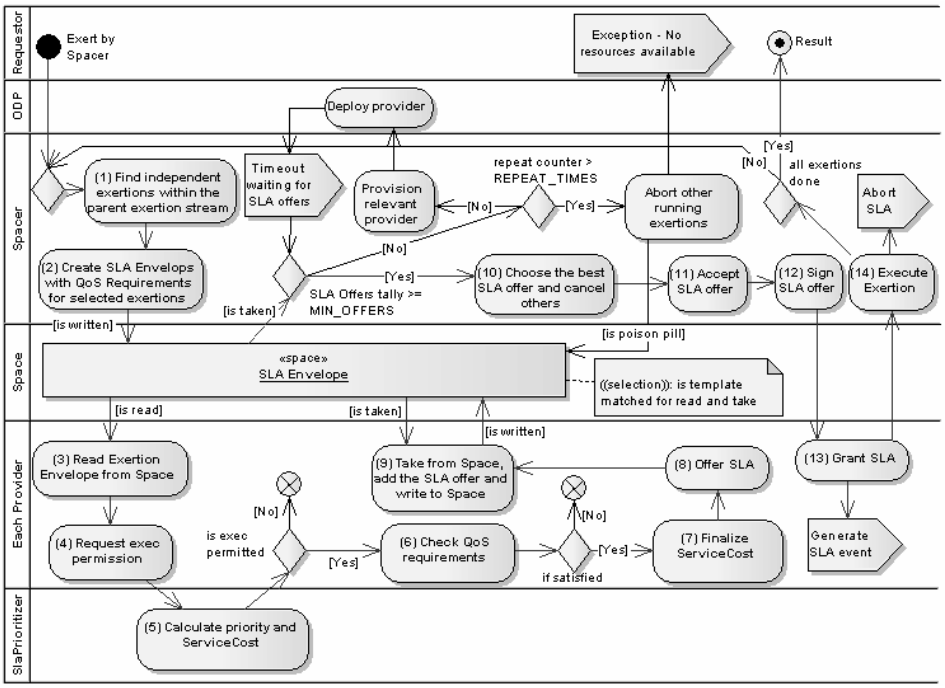

Figure 2. SLA Negotiation for Pull jobs

at this moment the requested number of offers is reached then Spacer passes to the next stage of selecting the most appropriate offer. Otherwise, Spacer calls the *OnDemandProvisioner* service and tries to provision the requested service provider. The aforementioned sequence is repeated as many times as necessary to collect enough offers but no longer then until the REPEAT_TIMES parameter is reached. However, if the number of collected offers is not sufficient during the last round of the algorithm and provisioning succeeds, Spacer gives the exertion another chance by allowing the sequence to be repeated once more.

4.2 Selecting the Best SLA Offer

When the number of collected offers reaches the value specified in the MIN_OFFERS parameter the control flow is passed to the SLA optimizer component within the Spacer service. The optimizer selects the most appropriate offer based on estimated time and cost parameters specified in every offer and the requested priority contained in the QoS requirements for a given exertion. This selection may be performed according to one parameter (the shortest time or the lowest cost) or it may involve multi-objective optimization (the shortest time at constrained cost or vice-versa), however, in case of *Pull* jobs a full upfront multi-objective optimization is not possible as it undermines the basic assumption of asynchronous operation and time independence of tasks within the *Pull* job.

For the same reason there is no possibility for the Spacer to select an updated SLA offer, that is, one were the original QoS requirements are not entirely fulfilled. In such case, however, Spacer returns to the requester the set of updated offers and the requester may choose to lower its expectations and rerun the job.

4.3 Leasing SLA Offers

SERVME uses a leasing mechanism to avoid blocking resources unnecessarily as well as a distributed garbage collector. In case of *Pull* jobs, the mechanism is designed in the following way. A *Lease* is created when a service provider issues an SLA offer. This lease has a fixed timeout. The offer is written to the space. When it is collected by the Spacer service it is immediately passed to a lease renewal service which takes care of extending the *Lease* automatically until it is released or the service is disposed. When offers are collected and the Spacer selects the best one, its *Lease* is extended while all other ones are canceled allowing resources allocated for these SLA offers to immediately become free.

The *Lease* for the chosen offer is monitored and extended by the renewal service for the whole duration of the exertion's execution that begins shortly after the selection of the SLA offer.

4.4 SLA Acceptance, Signing and Execution

This step should be performed by the service requester and that is the case with *Push* jobs, however, in case of *Pull* jobs such behavior would impose significant overhead needed to asynchronously pass the selected SLA offers to the requester for acceptance and signing. Furthermore, the com-

munication model imposed by the federated method invocation does not allow to communicate with the requester without stopping the control flow of the entire exertion and passing it to the requester. As a result, it is proposed to allow the Spacer service to perform the signing of SLA offers on behalf of the requester.

Spacer accepts and signs chosen SLA offers independently for each exertion immediately after selecting it. Then the SLA is sent to the issuing service provider and the exertion's service() method is called to start the execution.

When the execution of all inner exertions finishes the Spacer collects the results, calculates the actual overall time and cost of the execution and returns the results and the control flow to the requester.

4.5 SLA Envelops

The SLA Envelop created for each exertion and written to the space contains the following information a) exertion ID, b) SLA contract (requirements or offer), c) SLA state, d) service type, e) optionally the requested provider's name.

When the envelop is written to the space its SLA state is set to SLA_REQUEST. Every service provider creates a template to match requests for SLA offers for every service type (interface) that it can offer. Each of those templates contains the offered service type and the SLA state set to SLA_REQUEST.

4.6 Distributed Arrays in Space

As opposed to the algorithm of exerting non-QoS exertions via space proposed in SORCER that uses independent entries (exertion envelops) in the space, the SLA acquisition and negotiation algorithm in SERVME needs to store collections of entries in the space. This results from the fact that for every SLA offer request written to the space by the Spacer service there may be any number of SLA offers and the ability to determine their current number without reading all these objects from the space is crucial. Therefore SERVME proposes a structure called a distributed array that is an extended pattern based on the structure by the same name proposed by F. Freeman et al. in [5]. SERVME extends the basic distributed array pattern by allowing multiple elements to be taken from the array as well as fixes some issues with transactional handling of the arrays.

During the SLA negotiation process the Spacer service creates two distributed arrays for every SLA offer request: one for SLA offers and another one for offers that do not completely fulfill QoS requirements - updated SLA offers.

Service providers append their offers to the corresponding arrays while the Spacer service monitors the size of the array that contains SLA offers. When the number of collected offers reaches the required number specified in the MIN_OFFERS parameter or when a timeout event occurs both arrays are read from space and deleted.

5 Deployment

The proposed coordination algorithm was implemented and successfully tested within the SORCER MOS.

The reference implementation was written in Java 1.6 and requires Jini 2.1 and Rio 4.0 [10]. The Rio runtime is used for provisioning and as a source of QoS data. The Rio's Service UI is integrated into the SERVME service provider's UI and so, it allows the user to view and monitor QoS parameters at runtime.

6 Performance Analysis

The complexity of distributed systems requires extensive testing since often theoretical models are useless in practice. A real-world use case from bioengineering was selected to validate the proposed solution.

The problem addressed in the use case focuses on protein sequencing using the Rosetta software. Rosetta has become a *de facto* standard for scientists involved with protein sequencing. It has been ported to the Berkeley Open Infrastructure for Network Computing (BOINC) and as Rosetta@home is available for those who want to contribute by devoting part of their CPU time. Unfortunately, not all scientists can benefit from this free CPU time.

The goal behind protein structure prediction is to develop methods and algorithms that will allow us to predict and order any given protein structures of any length of the amino acid chain. The way scientists are trying to achieve this is by generating computed models of proteins, so called decoys, and comparing them with the real structure solved by experimental methods.

The use case scenario involved running multiple simultaneous computations to generate a large number of decoys for a particular protein structure. For practical reasons, since it is easier to observe how well SERVME manages resources on a small example, the chosen structure is rather short.

This protein known as the 2KHT structure has a length of 30 residues and is represented by the following sequence: ACYCRIPACIAGERRYGTCIYQGRLWAFCC

It is important to mention that in this use-case the SORCER MOS is used similarily to a traditional grid: the business logic comes from the provider and the exertion job is only used to coordinate the simultaneous execution of multiple calculations. Although it is a notable limitation it simplifies the use-case and allows to focus on resource management capabilities.

The computations were run using the stand-alone Rosetta software 3.1 on a cluster that consists of 10 AMD Opteron-based servers connected by a Gigabit network and running Centos 5.3 Linux.

6.1 Assumptions

The experiments were run under the following assumptions:

1. Every exertion job contained 20 inner simple exertions (tasks). The goal of each task within this job was to compute one decoy for the protein structure presented above. The QoS requirements requested an exclusive reservation of one CPU's core to any given task.

2. To ensure comparability of results the cluster was not used by any other user and no other jobs were run simultaneously. All experiments were repeated many times. (50 for jobs with a parallel flow and 30 for sequential ones).

3. Every provider used a cost/time approximation model in which the cost is inversely proportional to the execution time – as a result the execution on faster machines is more costly than on lower-end hardware. In these tests the chosen algorithm was rather simple. The cost was estimated according to the following formula: $EstimatedCost = \frac{5000}{EstimatedTime}$, where the EstimatedTime expressed in milliseconds was calculated as the average execution time for previously run exertions with similar parameters: in this case, since the tasks were practically identical, this applied to calculations of all previous decoys.

4. The cost of coordinating a job by the rendezvous peer was constant and set to 100 units.

6.2 Discussion

Despite holding the assumption about exclusive access to the computing cluster used for the experiments, in a distributed system there is always a number of factors that influence the performance and thus the execution times of computing tasks. This property does not allow to perform a complete statistical analysis and, in particular, discourages from drawing quantitative conclusions from the outcomes. However, the results together with the *a priori* knowledge about the applied coordination algorithms allow us to infer qualitative hypotheses and thus concentrate on trends rather than differences expressed in numbers. On the other hand, however, for better credibility the observations were analyzed from the point of view of their distribution. With minor exceptions, the kurtosis is positive or oscillates around "0". This fact together with a relatively small standard deviation allows for assuming that the distribution concentrates around the average and thus the average may be regarded as a credible measure.

The discussion on the results focuses on three trends that may be observed: (1) To show how jobs run with QoS management perform in comparison to those that are coordinated using techniques previously available in the SORCER environment. (2) To prove that SERVME allows for optimization of the execution of composite exertions for the best time or the lowest cost. (3) To show how the chosen parameters (in particular, MIN_OFFERS) influence the actual performance.

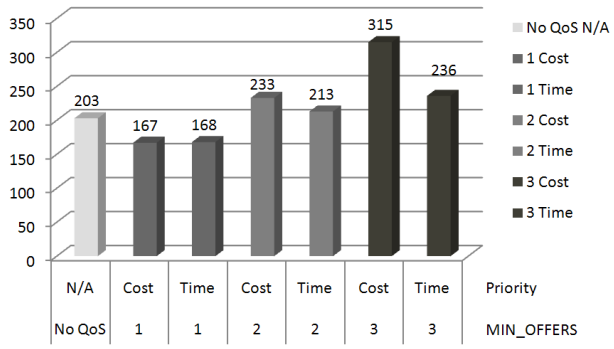

Figure 3. Execution Time (s) for Parallel jobs

Table 2. Execution Time: Parallel jobs

	Job Parameters		Execution Time (s)			Execution Cost		
Pos.	Min Offers	Priority	Average	Kurtosis	Std Dev.	Average	Kurtosis	Std Dev.
1	No QoS	N/A	**206**	-0,16	26	**N/A**		
2	1	Cost	**167**	1,64	14	**1682**	2,13	40
3	1	Time	**168**	0,6	12	**1645**	-0,66	33
4	2	Cost	**233**	1,01	23	**1422**	0,64	40
5	2	Time	**213**	-0,68	25	**1649**	-0,52	40
6	3	Cost	**315**	-1,08	54	**1440**	-1,96	37
7	3	Time	**236**	0,15	38	**1744**	-1,02	26

These trends may be better observed by looking at jobs with a parallel flow therefore due to space limitations the results of sequentially run experiments have been omitted in this paper. The chart shown in Figure 3 presents the average execution time for the non-QoS Pull job vs. QoS Pull. The time is expressed in seconds and the parameters for each job (priority and MIN_OFFERS) are given below every bar.

The first trend may be observed by looking at three figures. The results in Figure 3 show that Pull jobs with QoS perform better only if they use a FIFO type of SLA selection, that is, when MIN_OFFERS is set to "1". The reason why the execution time rises together with the MIN_OFFERS parameter results probably from the waiting time spent on collecting SLA offers. It is worth noting that the task executed in the experiment was relatively short. In case of calculations that take longer the rising waiting time should have a less significant impact.

As far as the second trend: cost vs. time priority is concerned the experiments show that the inverse correlation of time and cost rises increases together with the MIN_OFFERS parameter. This tendency is not very strong and probably results from the fact that the cluster used for experiments is not really heterogeneous – all nodes have CPUs of the same speed. In a more diverse deployment the effects of setting the optimization priority should be more evident.

Finally, some general conclusions can be drawn. Setting a higher MIN_OFFERS parameter makes only sense when the priority is to lower the cost, since it allows for achieving a more precise selection and, consequently a

lower cost (if the priority is set to cost) but adds waiting time required to collect more offers and thus significantly impairs the overall execution time.

The results proved that with appropriate parameters, the new coordination algorithm allows us to significantly shorten the overall execution time for complex computations while adhering to the requested QoS requirements and allowing resources to be used in a fair way. This last point is particularly important for settings where SERVME may be applied to manage resources in a shared environment used concurrently by many users.

7 Conclusion

The paper presents a new SLA-based asynchronous coordination algorithm for Federated Metacomputing Environments. This algorithm integrates the recently developed dynamic SLA negotiation with the space-based computing approach. It was developed as an extension within the SERViceable Metacomputing Environment and thus uses its infrastructure services: QosCatalog, SlaDispatcher, SlaMonitor, SlaPrioritizer, and OnDemandProvisioner. The SLA object model introduced in SERVME is utilized as a common description model for QoS parameters. The presented solution addresses the challenges of spontaneous federations in SORCER and allows for better resource allocation.

As the performance analysis realized on a real-world use case of protein structure prediction shows the new algorithm allows us to optimize the usage of resources according to cost/time priorities and with certain parameters shortens the overall execution time in comparison to the previous non-QoS algorithm used in SORCER. The introduced on-demand provisioning of services provides for better hardware utilization by reducing the number of compute resources to those presently required for collaborations defined by corresponding exertions and therefore allows the architecture to scale very well. 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". Finally, the proposed environment allows for accounting of resource utilization based on dynamic cost metrics, thus it contributes towards the realization of the vision of utility computing.

References

[1] J. Cao, D. Spooner, J.D. Turner, S. Jarvis, D. J. Kerbyson, S. Saini, and G.R. Nudd. Agent-based resource management for grid computing, 2002.

[2] K.-M. Chao, M. Younas, and N. Griffiths. BPEL4WS-based coordination of grid services in design. *Computers in Industry*, 57(8-9):778–786, December 2006.

[3] T. Eymann, M. Reinicke, O. Ardaiz, P. Artigas, L. DÃaz de Cerio, F. Freitag, R. Messeguer, L. Navarro, D. Royo, and K. Sanjeevan. Decentralized vs. centralized economic coordination of resource allocation in grids. In F.F. Rivera, M. Bubak, A.G. Tato, and R. Doallo, editors, *Grid Computing*, volume 2970 of *Lecture Notes in Computer Science*, pages 9–16. Springer Berlin / Heidelberg, 2004.

[4] F. Favarim, J. da Silva Fraga, L.C.Lung, M.Correia, and J.F. Santos. Exploiting tuple spaces to provide Fault-Tolerant scheduling on computational grids. In *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 403–411. IEEE Computer Society, 2007.

[5] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces(TM) Principles, Patterns, and Practice*. Prentice Hall PTR, June 1999.

[6] D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[7] Z. Juhasz and L. Kesmarki. A Jini-Based prototype metacomputing framework. In *Euro-Par 2000 Parallel Processing*, pages 1171–1174. 2000.

[8] D. Khushraj, O. Lassila, and T. Finin. sTuples: semantic tuple spaces. In *Mobile and Ubiquitous Systems, Annual International Conference on*, volume 0, pages 268–277, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[9] E. Kühn, R. Mordinyi, and C. Schreiber. An extensible Space-Based coordination approach for modeling complex patterns in large systems,. In *Leveraging Applications of Formal Methods, Verification and Validation*, pages 634–648. 2009.

[10] D. Reedy. Project rio: A dynamic adaptive network architecture. Technical report, Technical Report, Sun Microsystems, 2004.

[11] P. Rubach and M. Sobolewski. Autonomic SLA management in federated computing environments. In *2009 International Conference on Parallel Processing Workshops*, pages 314–321, Vienna, Austria, 2009.

[12] P. Rubach and M. Sobolewski. Dynamic SLA negotiation in autonomic federated environments. In Robert Meersman, Pilar Herrero, and Tharam Dillon, editors, *On the Move to Meaningful Internet Systems: OTM 2009 Workshops*, volume 5872 of *Lecture Notes in Computer Science*, page 248–258. Springer, 2009.

[13] M. Sobolewski. Federated method invocation with exertions. In *Proceedings of the 2007 IMCSIT Conference*, pages 765–778. PTI Press, 2007.

[14] M. Sobolewski. Exertion oriented programming. In *International Journal on Computer Science and Information Systems*, volume 3 of *IADIS*, pages 86–109. 2008.

[15] M. Sobolewski. Metacomputing with federated method invocation. In *Advances in Computer Science and IT*, pages 337–363. M. Akbar Hussain, In-Tech edition, 2009.

[16] V. Sunderam and D. Kurzyniec. Lightweight self-organizing frameworks for metacomputing. In *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*, pages 113–122, 2002.