Using Continuations for Flexible Provision of Grid Services

Maurizio Giordano and Claudia Di Napoli

Istituto di Cibernetica "E. Caianiello" - C.N.R.
Via Campi Flegrei 34, 80078 Pozzuoli, Naples - ITALY
{m.giordano,c.dinapoli}@cib.na.cnr.it

Abstract. The main challenge of grid computing is to provide a unified computational infrastructure composed of networked heterogeneous resources that makes effective use of the computational power delivered by each resource. To reach this objective management of computational resources is a crucial aspect because of the decentralized, heterogeneous and autonomous nature of these resources that usually belong to different administrative domains. As such they cannot be managed by adopting a centralized approach, but more sophisticated computing methodologies are necessary. In this context, the possibility to manage the execution of services is advisable to control their provision in dynamic and changing environments. In the present work an infrastructure to model service providers is proposed to allow for flexible provision of grid services, i.e. to allow providers to dynamically adapt the execution of services according to both the changing conditions of the environment where they operate in, and the requirements of service users. The infrastructure is based on *continuations*, a programming paradigm that allows to control the state of code execution at programming level without directly using operating system facilities.

1 Introduction

Computational grids represent the new research challenge in the area of distributed computing. They aim to provide a unified computational infrastructure composed of geographically distributed heterogeneous resources cooperating with each other through middleware software to enable usage of the collection of these resources in an easy and effective manner.

In the present work a service-oriented approach is adopted as described in [1], where grid resources are abstracted as *grid service*s, i.e. computational capabilities exposed to the network through a set of well-defined interfaces and standard protocols used to invoke the services from those interfaces, and they have to be identified, published, allocated, and scheduled. Services are not subject to centralized control (i.e. they live within different control domains and they do not rely on a central management system), they use standard, open, general-purpose protocols and interfaces (i.e. not application-specific), and they can be combined in order to deliver added value functionalities so that the utility of the resulting

system is significantly greater than that of the sum of its parts. In order to provide such a computational infrastructure, grid technologies should support the sharing and coordinated use of diverse resources in a dynamic environment [1].

A service is provided by the *body* responsible for offering it, we refer to as *service provider*, for consumption by others, we refer to as *service consumer*s, under particular conditions. In this view, service providers (that can be individuals, organizations, groups, government, and so on) are independent and autonomous entities representing the interface between a service consumer and a required functionality, i.e. a grid service. Users will be able to access and share these computational capabilities on demand over the Internet, relying on an infrastructure that is expected to be resilient, self-managing, and always available, and above all that is perceived as a unified framework by end users.

A service request is fulfilled when the consumer requirements can be met by the service provider that received the request, i.e. when consumers's Quality-of-Service requirements can be met by provider's Quality-of-Service capabilities [2]. The term Quality-of-Service is used in a general sense referring to a very wide range of non-functional service characteristics. It is beyond the scope of the present work to study how complex the quality of a service can be, and how to characterize it, i.e. how many parameters should be considered to express the quality of a service, and how it can be represented. This is mainly a domain-specific problem.

In this approach, service providers must be equipped with mechanisms within their architectures to allow for the provision of known quality levels and for the possibility to change quality levels when necessary. So, providers need to have control on the execution of the services they provide in order to accommodate for the changing conditions under which a service could be provided. In such a way providers are able to decide at run-time "how" to fulfill a service request, i.e. what Quality-of-Service they can provide the service with.

In this work we propose an infrastructure to model service providers to control the execution of services by allowing for service suspension and resuming in a way similar to process preemption in traditional operating system design.

The infrastructure relies on *continuation* programming paradigm [3] in order to provide execution state saving/restoring mechanisms for services. These mechanisms will support the possibility of dynamically controlling the execution of services to allow providers to change at run-time parameters affecting service provision either driven by consumer or system requirements.

The rest of the paper is organized as follows. Section 2 gives an overview of the notion of continuation and its deployment in some classes of programming languages. Section 3 describes the high-level design of the proposed service provider infrastructure and its functionalities, together with the APIs to use it. Section 4 gives details on the infrastructure implementation and Sec. 5 describes the reference application scenario of the proposed work. Finally Sect. 6 reports some conclusions and comparison to related work.

2 Continuations in Programming Languages

A *continuation* relative to a point in a program represents the *remainder of the computation* from that point [3], so a continuation is a representation of the program current execution state. Continuation capturing allows to package the whole state of a computation up to a given point. Continuation invocation allows to restore that previous state restarting the computation from that point. Although any programming system maintains the current continuation of each program instruction it evaluates, these continuations are generally not accessible to the programmer.

In functional programming languages, the continuation can be represented as a function and the possibility of explicitly managing it allows to effect the program control flow [4]. In languages like C the current execution state is represented by the call stack state, the globals, and the program counter. Some object-oriented programming languages support continuations by providing constructs to save the current execution state into an object, and then to restore the state from this object at a later time.

Depending on which operations can be done at programming level on the continuation store object, it is possible to classify the type of support for continuations in the given programming language. In particular, a *first-class continuation* is represented by a first-class object as defined by Abelson & Sussman in [5]:

> *a first-class data object is a language element that may be named by variables, passed as arguments to procedures, returned as results of procedures, and included in data structures*

There are Java language extensions supporting continuations as not-first-class objects, like RIFE [6] and Jetty 6 [7], in what they provide constructs to capture and resume continuations, but no mechanisms to store them in data objects that can be passed as functions call parameters.

In what follows we outline a short and non-exhaustive list of programming language extensions that support continuations. Our attention is focused on languages and programming environments providing continuation capturing and resuming constructs to build up lightweight user-level threads that can be easily managed and scheduled at application level.

- JavaFlow [8] is a Java component of the Apache Jakarta [9] project that implements continuations. In JavaFlow a continuation is the state of an application, i.e. the stack of functions calls including local and global variables and the program counter. Continuations are captured and saved into a Java object that allows to restart the processing from the point stored in it. JavaFlow supports continuations as first-class objects: the continuation object interface includes methods to suspend and resume it.
- Stackless Python [10] is an experimental implementation of the Python programming language that uses continuation support to model concurrency in

an easy way. It provides abstractions of microthreads at application level, named *tasklets*, whose implementation is based on continuations. Stackless Python supports *tasklets* as built-in user-level lightweight threads with constructs to control their creation, suspension, resuming and scheduling at application level.
- Ruby [11] language supports natively continuations as first-class data types. The `callcc{|$cc|}` construct is provided to capture continuation of programs at any location, binding it to the `$cc` argument variable. Continuation resumption is performed by the `$cc.call` function invocation. Continuations can be used in Ruby to implement user-level threads with suspension/resuming and scheduling support at application level.
- Rhino [12] is an opensource Javascript implementation written in Java. There is a Rhino additional package [13] which introduces an additional mode for the Rhino (Javascript) interpreter that supports first-class continuations. In Rhino a continuation is a Javascript object that represents the storing of the Rhino script execution state. You can return this object, store it in a variable, assign it to a property of another object. The continuation object is also a function: when you call it the current execution state of the program is discarded and the snapshot of the program represented by the continuation object is resumed in its place. Apache Cocoon web development component-based framework [14] uses Rhino Javascript to implement its controller logic.

3 Service Provider Design

In order to be able to provide services that meet Quality-of-Service requirements both of service consumers (e.g. cost, response-time) and of providers (e.g. throughput, profit, CPU utilization), it is crucial to be able to control the execution of services in accordance with new events occurring in the environment since these requirements cannot be always statically determined.

Service preemption mechanisms are a way to provide full control of service execution and they can be implemented (or simulated) using several approaches, both at application or operating system level. For examples, at application level the Java language provides (deprecated) thread suspension/resuming support. Other approaches [15] use operating systems signals (`SIGSTOP`/`SIGCONT`) available in most operating system infrastructures.

The main objective of the proposed service provider architecture is application-level preemption of services in order to support at programming level the development of dynamic policies for service execution. Service preemption is not provided at operating system level, but at application-level by managing program continuations. This choice makes the framework flexible and easily adaptable for developing and experimenting scheduling facilities, policies and service-control in different service-oriented architecture applications.

Existing web service frameworks [16,17] make it difficult to implement a service provider architecture with preemption mechanisms of web services without a deep changing of the control patterns usually implemented as a built-in feature. This is because most frameworks obey to the *Inversion of Control*

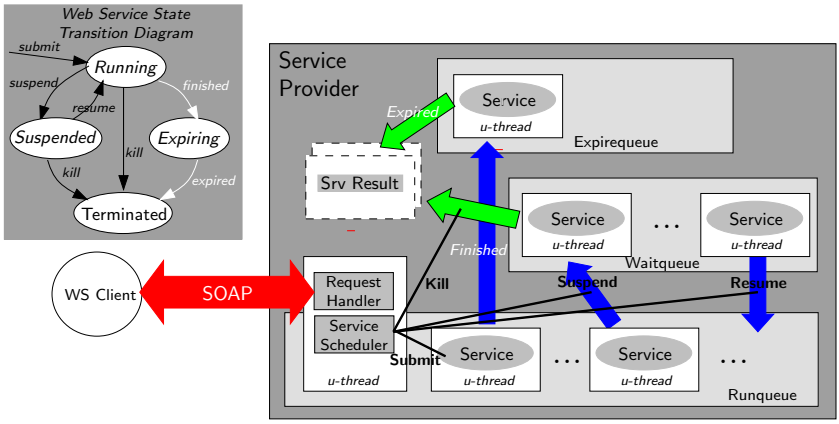

Fig. 1. Services provider architecture and service state transition

(IoC) programming pattern [18,19] widely used in most Java and object-oriented web-application environments. So, web service instantiation and life-cycle management cannot be fully controlled by programmers who develop and add web services to the framework.

For this reason existing web service frameworks are not suitable to provide an application-level control of service execution supporting service suspension and resuming.

For this reason we designed a service provider equipped with mechanisms to process suspension and resuming notifications. The service provider should process, from time to time, arrival of notification messages in order to suspend/resume the execution of a service it is providing by capturing/restoring its continuation. The control of service execution can be driven both by the service provider itself and by any client program. Service preemption, driven or not by client requests, is carried out by the provider storing at the preemption points the execution state of the specified service.

A client program can represent either a service consumer that requires a service result, or a metascheduler or service broker [20] trying to adapt local service execution policies so that resources can be shared in a reliable and efficient way in a heterogeneous and dynamically changing environment like the grid.

3.1 User-Level Control of Services

Our proposal of a service provider architecture supporting user-level preemptible service execution is depicted in Fig. 1. The provider is represented by a *service container* consisting of a pool of lightweight user-level threads, named *u-threads*, whose implementation should support suspension and resuming facilities of threads at application level.

In our design u-threads are wrapper functions that embody and control the execution of web service WSDL *operations* [21]. Web service operations are supplied as parameters to u-threads and executed within their contexts (see Fig. 1). Thus the wrapping guarantees the required functionalities to suspend and resume web service operations.

A u-thread, and hence the enveloped service instance, can be in the following states:

- *running*, i.e. the service instance is executing or ready to be scheduled for execution;
- *suspended*, i.e. the service instance is not yet terminated, but cannot be scheduled for execution;
- *expiring*, i.e. the service instance terminated, but the wrapping u-thread descriptor is still alive to make the service result available to successive requests;
- *terminated*, i.e. the service instance terminated and the wrapping u-thread descriptor is freed and no longer available because either a specified expiration time elapsed, or the client requested and obtained the service result before the expiration time.

The service state transition diagram is showed in the leftmost part of Fig. 1.

A main u-thread, always in *running* state, represents the service provider execution context. The main u-thread program interleaves messaging and scheduling activities by running two modules: the *Request Handler* and the *Service Scheduler*. The *Request Handler* deals with probing incoming SOAP messages; the *Service Scheduler* controls the state transitions of u-threads wrapping service instances. This is accomplished by means of the following primitives: *submit*, *suspend*, *resume*, *kill*. The primitives are reported as black thin lines in the service provider architecture of Fig. 1, and as black arrows in the transition state diagram in the same figure. The *submit* primitive creates a new u-thread, wrapping up a specified service operation and puts it in the *running* state.

The Service Scheduler maintains three queues to manage u-threads, and the wrapped services instances, in the different states:

Runqueue - it contains all service instances running or ready to be scheduled for execution. Services in this queue are by default executed in time-sharing mode by assigning to each u-thread a *time quantum* that can be changed by the Service Scheduler (also in response to incoming SOAP requests).

Waitqueue - it contains all service instances suspended and thus removed from the Runqueue. The provider may decide to suspend/resume service execution according to both its own scheduling policy, and upon receiving specific SOAP requests from an external application, e.g. a metascheduler.

Expirequeue - it contains all u-threads' descriptors wrapping up terminated service instances whose results are not yet requested by clients via SOAP messages. U-threads are maintained in an inactive state in this queue to temporarily store unused service results until a certain *expiration time* is elapsed. The expiration time is not necessarily a system specific parameter, and it could be specified as a QoS parameter at the submitting phase.

It should be noted that in the Service Scheduler module different scheduling policies can be implemented at application-level overriding the default one both by changing the *time quantum* and by accessing and managing the Runqueue in different modes. In this way the service provider is able to change its own local scheduling policy at run-time directly invoking the primitives to control service execution.

3.2 Service Control APIs

As outlined earlier, the proposed infrastructure allows also to access the primitives to control service execution as web services to be invoked by any external client program. In such a case, a client-provider interaction takes place and it is implemented as an asynchronous request/response operation with polling [22]. Asynchronicity allows the client to proceed the computation concurrently with the web service execution until the operation result is required: at this point the client needs to synchronize with the provider and establishes a new communication to retrieve the result.

We extend the asynchronous request/response operation mode with functionalities to suspend and resume web service execution. The client-provider asynchronous interaction pattern is described in Fig. 2 where a client invokes a web service operation, named "Operation A", offered by the continuation-based service provider.

The primitives to control service execution are exposed as the following WSDL operations: `submit`, `suspend`, `resume` and `probe`. They represent *meta-operations* because they are invoked by clients to control and to monitor web service operation executions.

The client-provider interaction pattern is started by clients invoking the `submit` WSDL operation to request a service execution. The `submit` request invokes the "Operation A" on a set of input arguments and starts its execution (see the syntax in Fig. 3(a)). The provider sends back to the client a reply with an acknowledge that the submission is done together with a *correlation ID*. The correlation ID is unique and is set by the provider to be used together with the client to associate subsequent requests and responses belonging to the same client-provider transaction.

Correlation tokens to embed multiple messages in transactions are widely used in most asynchronous web service protocol proposals [23,24]. Approaches differ for the particular protocol adopted (JMS, SOAP, and so on) and/or the mechanisms used to implement message correlation. In our approach correlation is explicitly included in SOAP message bodies as shown in Fig. 3.

The `submit` request includes a set of `qos` parameters. QoS attributes are specified by clients to drive or affect scheduling policy of the web service operation execution.

The client starts the execution of "Operation A" and continues its computation so that it may also decide to suspend the web service execution, to resume it later on up to completion.

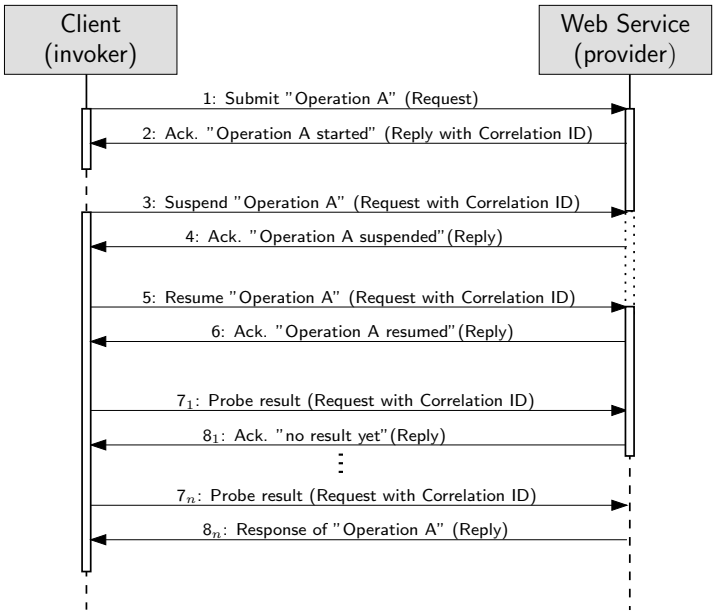

Fig. 2. Asynchronous request/response operation with polling and suspend/resume facility

To perform suspension and resuming actions the client uses the `suspend` and `resume` meta-operations.

The `suspend` request uses the correlation ID to refer to the web service operation (instance) to be suspended. Upon receiving the request, the provider captures and saves the execution state of "Operation A", and it sends back to the client an acknowledge.

The `resume` request uses the correlation ID to refer to the web service operation (instance) whose execution must be resumed. Upon receiving the request, the provider retrieves the execution state (continuation) stored and tagged with the specified correlation ID. It then resumes the web service operation and sends back to the client an acknowledge.

As described in Fig. 3(b), also the `resume` request includes specifications of QoS parameters. This means that in our framework a service execution could be resumed by changing at run-time the web service operation scheduling policies.

Client-provider synchronization is implemented by the `probe` request. The `probe` checks if "Operation A" is finished. If the request occurs before the web service operation exits (the first `probe` of Fig. 2), the client is acknowledged that the result in not ready yet. After a successful `probe` request, the client synchronizes with the provider and gets the result.

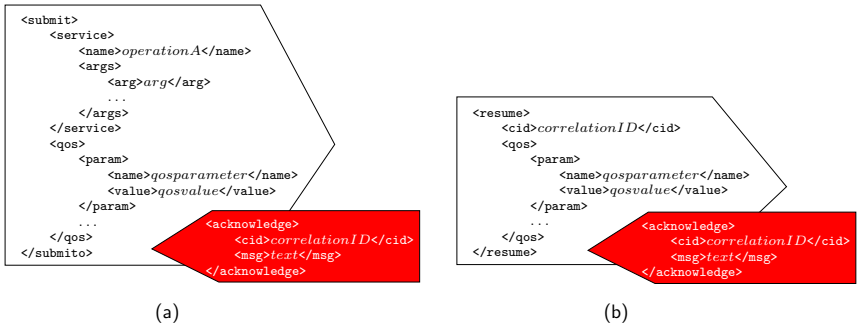

Fig. 3. Service control primitives syntax: (a) `submit`; (b) `resume`

4 Implementation

We outline here that our proposal of a service provider architecture with support to user-level preemptible service execution can be implemented with one of the continuation-based languages mentioned in Sec. 2. In fact user-level threads with suspension and resuming control functions can be implemented in a hosting language supporting first-class continuation management.

Our service provider current implementation is in Stackless Python [10]. The choice is due to two main motivations. First, the language directly supplies the support of user-level threads based on continuations, named *tasklets* that are built-in user-level lightweight threads with constructs to control their creation, suspension, resuming and scheduling at application level. Secondly, the Python scripting language offers a fast prototyping and testing programming environment for the proposed SOA framework, with minor performance penalties compared to other languages like C. Furthermore, Python is one of the languages that provides a satisfactory support of libraries and tools for the development of web services [25].

We implemented our user-level threads as extensions of *tasklets* objects (see Fig. 4), named *WSTasklets*. WSTasklets are threads wrapping up functions whose code implements web service *operations* (in WSDL specification). Web service operations are given as parameters to a WSTasklet constructor method and executed within its context (see Fig. 4).

Our u-threads inherit by tasklets the required functionalities to suspend and resume web service operation executions by means of the Stackless Python continuation storing and resuming features.

The service provider implementation is a `ServiceContainer` object managing and controlling a pool of WSTasklets. It executes in the context of the main WSTasklet, which is always in the running state. Its execution is interleaved with other tasklets that are present in the Stackless Python systemwide runqueue. Note that this queue contains all tasklets in the system (Phyton in-

terpreter) while the service provider Runqueue defined in Sect. 3.1 is used to manage and control WSTasklets wrapping service instances in the context of the service container.

As described in Fig. 4, the `ServiceContainer` object, once started, executes a never-ending loop interleaving the execution of the *Request Handler* and the *Service Scheduler* (see respectively the `handle_request_noblock()` and `timeshared_scheduling()` functions).

The Request Handler listens to the specified socket address/port to probe for incoming SOAP requests. If a SOAP message arrived, it is processed by the `TaskletController` object: this is the Python module providing access to the control primitives of services offered by our framework.

In the bottom of Fig. 4 we report only fragments of the `TaskletController` code: the `tsksubmit` method implements the *submit* primitive WSDL interface. In the same part of Fig. 4 the `foo` method represents an example of a web service of our provider whose execution can be controlled by means of the set of primitives whose semantics and interfaces were defined in Sect. 3. It should be noted that web service code does not need to be rewritten in order to be integrated and provided by our service provider. Only one modification is needed to convert the `return` construct with a system-dependent version: in fact, according to our design, service termination leaves the wrapping thread still alive although in an inactive state (Expirequeue) just for the time required by the service consumer to gather the result stored in the expiring WSTasklet. Remember that no value is returned by the service to clients if a maximum expiration time elapses without no `probe` incoming requests.

The Service Scheduler is the module in charge of managing the user-level thread Runqueue and Waitqueue described in the previous section. In the example of Fig. 4 we report a round-robin scheduler module that assigns at each round a fixed time quantum (`timeshare`) for execution to all threads in the Runqueue. As before mentioned the scheduling module is a component of our framework that can be modified to program and experiment with different scheduling policies of service execution.

5 The Economic-Grid Scenario

In order to reach the full potential of grid computing, it is well-recognized that the grid needs to shift towards production-oriented platforms, so that service providers are motivated to make available the resources they provide.

A computational economy approach can be used to provide the possibility of buying and selling computational resources in the same way as goods and services are bought and sold in the real world economy [26]. Adopting a computational economy-based view [27,28] where services are provided at a given cost constitutes *per se* a mechanism for encouraging resource owners to contribute their resource(s) for the construction of the grid, and compensate them based on the resource usage, i.e. on the value of the work done. So, the ultimate success of computational grids as a production-oriented commercial platform for solving

problems is critically dependent on the support of market/economy-based mechanisms to resource management. In such "commercial" computational grids, resource owners act as service providers that make a profit by selling their services to users that act as buyers of computational resources for solving their problems.

A suitable application domain for the proposed architecture is the economic-aware grids in which Quality-of-Service features may include the *cost* of the service to be provided. In fact, in our framework it is possible to associate to the request of a service execution a `qos` parameter taking into account the cost of a service and to allow both the client and the provider to use its value to drive suspension and resuming of service execution.

It is likely that in very dynamic and changing computing environments like the grid, service providers can make different decisions on the Quality-of-Service they provide their services with, according to the requirements of new service requests. For example, they may want to break or change some previous agreements when a new consumer comes with a more remunerative request. The proposed framework makes it possible to implement dynamic priority-based service scheduling policies by means of suspension and resuming facilities and by taking into account `qos` parameter changes at run-time, so that more remunerative service requests can be executed with a higher priority.

6 Conclusions

In this work we propose a service provider architecture based on continuations management that provides primitives to control web services execution and to implement different service scheduling policies. The primitives are offered by the service provider to external (client) applications through SOAP messages.

With this approach we may implement the service execution policies at two levels: the lower level relies on the service provider layer to implement local schedulers; the higher level can be a metascheduler that interacts with multiple service provider schedulers in a distributed setting by means of SOAP messaging.

The close related work we found in literature is the framework proposed in [15] that allows for two-level approach to process scheduling: the user-level scheduler divides time into equal size *decision intervals* (higher-level quantum) at the beginning of which it decides the processes eligible to run during the interval; the kernel-level scheduler executes the selected processes according to the operating system scheduling policy. The selection of processes to be run is carried out using suspend and resume mechanisms offered by the underlying operating system. On the contrary, our approach relies completely on user-level suspend and resume functions, and it provides scheduling mechanisms in the extent of the service provider application rather than in the operating system one.

Our approach has similarities to Community Scheduler Framework (CSF) [29], an infrastructure providing facilities to define, configure and manage *meta-schedulers* for the grid. Metascheduling is conceived as a higher level of schedul-

ing decisions to coordinate local schedulers (PBS [30], LSF [31]) on hosts and clusters in a grid environment.

Like CSF we provide high-level scheduling functionalities either to service consumers or to metascheduler middleware. CSF functionalities mainly target configuration and management of scheduling policies and their coordination in a grid environment.

Like CFS, our framework allows service execution control and scheduling queues configuration and management through SOAP/WSDL messaging.

Although both approaches support suspension/resuming facilities, CSF applies them to control jobs, i.e. processes running in the hosting operating system environments. CSF defines high-level scheduling services (in Java) to map consumer requests into job control commands implemented at lower level in the scheduler running on the target host (kernel scheduler) or clusters (as PBS and LSF). In both cases, CSF job-control functionality depends on the underlying operating system layer.

Since in our approach service control does not rely on the operating system layer, the framework results in a flexible and easily adaptable programming environment to develop and experiment scheduling facilities, policies and service-control in different service-oriented architecture applications. Furthermore, portability can be guaranteed across heterogeneous programming environments with (explicit) support of continuation capturing and resuming.

We plan to test the proposed architecture by setting up an experimental application where a service provider can vary its scheduling policy at run-time according to the incoming service requests by changing the time quantum parameter representing the cost associated to each request so that it can maximize its income.

References

1. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The physiology of the grid: An open grid service architecture for distributed system integration. Technical report Open Grid Service Infrastructure WG (2002)
2. MacLaren, J., Sakellariou, R., Garibaldi, J., Ouelhadj, D.: Towards service level agreement based scheduling on the grid. In: Proceedings of the second European Across Grids Conference. (2004)
3. Daniel P. Friedman, C.T.H., Kohlbecker, E.E.: Programming with Continuations. In: Program Transformation and Programming Environments. Springer-Verlag (1984) 263–274.
4. Di Napoli, C. and Mango Furnari, M.: A continuation–based distributed lisp system. In: Proceedings of the First International Conference on Massively Parallel Computing Systems, IEEE Computer Society Press (1994) 523–527
5. Abelson, H., Sussman, G.J.: Structure and Interpretation of Computer Programs. 2nd edn. Volume ISBN 0-262-01077-1. MIT-Press (1993)
6. RIFE Web Continuations. `http://rifers.org/wiki/display/RIFE/Web+continuations` (2007)
7. Jetty Continuations. `http://docs.codehaus.org/display/JETTY/Continuations` (2007)

8. The Jakarte Project: Commons Javaflow. http://jakarta.apache.org/commons/sandbox/javaflow (2007)
9. The Apache Jakarta Project. http://jakarta.apache.org (2007)
10. Tismer, C.: Stackless python. http://www.stackless.com (2007)
11. Thomas, D., Fowler, C., Hunt, A.: Programming Ruby. 2nd edn. Volume ISBN 0-9745140-5-5. (October 2004)
12. Rhino: Javascript for Java. http://www.mozilla.org/rhino/ (2007)
13. RhinoWithContinuations. http://wiki.apache.org/cocoon/RhinoWithContinuations (2007)
14. The Apache Cocoon Project. http://cocoon.apache.org/ (2007)
15. Newhouse, T., Pasquale, J.: A user-level framework for scheduling within service execution environments. In: Proceedings of the 2004 IEEE International Conference on Services Computing (SCC '04), Washington, DC, USA, IEEE Computer Society (September 2004) 311–318
16. The Apache Software Foundation: Apache web services project - axis. http://ws.apache.org/axis (2007)
17. IBM developerWorks: WebSphere. http://www-128.ibm.com/developerworks/websphere (2007)
18. Johnson, R.E., Foote, B.: Designing reusable classes. Journal of Object-Oriented Programming **1**(2) (1988) 22–35
19. Fowler, M.: Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html (2004)
20. Vadhiyar, S., Dongarra, J.: A metascheduler for the grid. In: Proceedings of the 11th IEEE Symposium on High-Performance Distributed Computing. (2002)
21. Booth, D., Liu, C.K.: Web services description language (wsdl) version 2.0 part 0 primer. http://www.w3.org/TR/2007/PR-wsdl20-primer-20070523 (2007)
22. Adams, H.: Asynchronous operations and web services, part 2. http://www-128.ibm.com/developerworks/library/ws-asynch2/index.html (2002)
23. Swenson, K., Ricker, J.: Asynchronous web service protocol. http://xml.coverpages.org/AWSP-Draft20020405.pdf (2002)
24. Sun Developer Network: Developing asynchronous web services with java message service in sun java studio enterprise 7. http://developers.sun.com/prodtech/javatools/jsenterprise/reference/techart/jse7/asynch.html (2005)
25. SourceForge.net: Python web services. http://pywebsvcs.sourceforge.net (2007)
26. Wooldridge, M.: Engineering the computational economy. In: Proceedings of the Information Society Technologies Conference (IST–2000), Nice, France (2000)
27. Buyya, R., Abramson, D., Giddy, J.: An economy driven resource management architecture for global computational power grids. In: Proceedings of The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000), Las Vegas, USA (2000)
28. Buyya, R., Giddy, J., Abramson, D.: An evaluation of economy–based resource trading and scheduling on computational power grids for parameter sweep applications. In: Proceedings of The Second Workshop on Active Middleware Services (AMS 2000), In conjuction with Ninth IEEE International Symposium on High Performance, Pittsburgh, USA (2000)
29. Platform: Open source metascheduler for virtual organizations with the community scheduler framework (csf). Technical report, http://www.cs.virginia.edu/~grimshaw/CS851-2004/Platform/CSF_architecture.pdf (2007)
30. Open portable batch system. http://www.openpbs.org (2007)
31. Load sharing facility. http://www.platform.com (2007)

```
class ServiceContainer(ServiceContainer):
    sleepingTasklets = {}
    runningTasklets = {}
    expiredTasklets = {}
    Tasklets = {}
    timeshare = 1000

    def __init__(self, server_address, services=[], RequestHandlerClass=SOAPRequestHandler):
        ...
        return ServiceContainer.__init__(self, server_address, services=[], RequestHandlerClass=SOAPRequestHandler)

    def timeshare_scheduling(self):
        """Round-robin scheduling with fixed time-share"""
        # if the main u-thread is not the only running...
        if stackless.getruncount() != 1:
            t = stackless.run(self.gettimeshare())
            # If we got a tasklet, it was the interrupted one ... we need to reinsert it for rescheduling.
            if t:
               t.insert()

    def serve_forever(self):
        """Service Container: interleave communication and scheduling activities"""
        while 1:
            # handle incoming requests (SOAP) in not-blocking mode
            self.handle_request_noblock()
            # schedule services
            self.timeshare_scheduling()
    ...

def AsServer(port=80, services=(), RequestHandlerClass=SOAPRequestHandler):
    '''services -- list of registered and exposed web service instances'''
    address = ('', port)
    sc = PreemptServiceContainer(address, RequestHandlerClass=RequestHandlerClass)
    # register web services in the service container
    for service in services:
        path = service.getPost()
        sc.setNode(service, path)
        service.setContainer(sc)
    sc.serve_forever()

print "... starting server"
AsServer(port=12321, services=[TaskletController()],RequestHandlerClass=SOAPRequestHandler)
```

```
class TaskletController(ServiceSOAPBinding):
    soapAction = {}
    root = {}
    container = {}
    _wsdl = """<?xml version=\"1.0\" ?> ..."""

    def __init__(self, post='/TaskletScheduler', **kw):
        ServiceSOAPBinding.__init__(self, post)

    def setContainer(self,sc):
        self.container = sc

    def tsksubmit(self, quanta, service, *args):
       if service in dir(self):
          # create the tasklet and use it to wrap service
          t = WSTasklet(eval("self.%s" % service))(*args)
          # set qos parameters of tasklet
          t.setquanta(quanta)
          unID = "%s:%i" % (t.getname(), t.getID())
          # manage scheduling queues
          self.container.Tasklets[unicode(unID)] = t
          self.container.runningTasklets[unicode(unID)] = t
          # schedule tasklet fo execution
          t.insert()
          # if back the tasklet has run for a quanta and now is suspended
          return [ t.getname(), t.getID(), t.getquanta(), "suspended" ]
       else:
          return "\n...no such service %s" % service

    ...

    def foo(self, name):
      n = 1
      while n < 1000000:
         n += 1
      self._return("Hello " + name)

    ....
```

Fig. 4. Continuation-based service provider implementation in Python