A Continuation-Based Framework for Economy-Driven Grid Service Provision

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 management of computational resources is a crucial aspect in grid computing because of the decentralized, heterogeneous and autonomous nature of these resources that usually belong to different administrative domains and are provided in dynamic and changing environments. For this reason more sophisticated computing methodologies are necessary to provide these resources in a flexible manner. In particular, the possibility of controlling the execution of services in grid is a crucial aspect in order to change service execution policies at run-time.

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 control the execution of services according to the changing conditions of the environment where they operate in. The infrastructure is based on *continuations*, a programming paradigm that allows to control the state of program execution at application-level without involving the operating system stack. This approach makes the proposed infrastructure a flexible and easily programmable middleware to experiment different scheduling policies in service-oriented scenarios.

Keywords: Grid service provision, continuations, quality of service.

1 Introduction

Computational grids represent the new research challenge in the area of distributed computing. They aim at providing a unified computational infrastructure composed of networked heterogeneous resources that makes effective use of the computational power delivered by each resource.

A computational grid is a pool of resources that are not subject to centralized control (i.e. that live within different control domains and that do not rely on a central management system), that use standard, open, general-purpose protocols and interfaces (i.e. not application-specific). Resources can be combined in order to deliver added value services so that the utility of the resulting system is significantly greater than that of the sum of its parts. Users will be able to access and share these computing resources 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

D.J. Veit and J. Altmann (Eds.): GECON 2007, LNCS 4685, pp. 112–123, 2007.

users. 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].

In addition, in grid environments resource providers (that can be individuals, organizations, groups, government, and so on) are independent and autonomous entities that need to be motivated to make available the resources they provide.

A market-oriented 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 [2]. Adopting a computational economy-based view [3,4] where resources are provided at a given cost constitutes *per se* a mechanism for encouraging resource owners to contribute their resources 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 production-oriented (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.

In this scenario, service providers need to have control on the execution of the services they provide in order to accommodate for the changing *Quality-of-Service* (QoS) [5] requirements service consumers can ask for.

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 and control in traditional operating systems.

The infrastructure relies on *continuation* programming paradigm [6] in order to provide execution state saving and restoring mechanisms for services. These mechanisms will support the possibility of explicitly controlling the execution of services to allow providers to decide "how" to fulfil a service request, i.e. what Quality-of-Service to provide at run-time.

The rest of the paper is so organized: in section 2 the economic-based service-oriented scenario is described as the reference application domain; section 3 reports the proposed service provider architecture together with its functionalities and interfaces; in section 4 some use-cases are outlined to show the applicability of the proposed infrastructure in commercial computational grids; finally section 5 reports some conclusions.

2 A Service-Oriented Approach for Economic-Aware Grids

In the present work a service-oriented approach is adopted as described in [1], where grid resources are exposed to the network as *grid service*s, i.e. computational capabilities defined through a set of well-defined interfaces, and a set of standard protocols used to invoke the services from those interfaces, and it has to be identified, published, allocated, and scheduled. 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, a service provider represents the interface between a service consumer and a required functionality, i.e. a grid service.

It is worth noting that in our scenario we refer to a grid service as any type of computational resource made available through the network according to platform independent interfaces and protocols, so it can also be a web service compliant with OGSA [7]. In the rest of the paper we refer to "web services", or "grid services", or simply "services" assuming they have, from our point of view, the same meaning.

It is well recognized that in a market-based service-oriented grid, services will be provided with some user-dependent *Quality-of-Service* (QoS) requirements, i.e. with characteristics that meet expectations and obligations agreed between the provider and the consumer. Service providers may want to optimize utilization, i.e. their profit, whereas service consumers may want to optimize response time while minimizing cost. So, the same service could be provided with different QoS.

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, that is mainly a domain-specific problem. In general, we assume that a service request is fulfilled when the user requirements on the Quality-of-Service can be met by the service provider that received the request. According to the current research directions, the match is stated in Service Level Agreements (SLAs) [8], that represent bilateral agreements typically between a service provider and a service consumer established before service execution.

Nevertheless, 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, e.g. they may want to break or change some agreements in the case a new consumer comes with a more remunerative request. Also service consumers may decide to change some requirements on service execution, e.g. they may want to pay more to have a service delivered earlier. In such cases, it is advisable to control the execution of services on demand by suspending and resuming their execution according to decisions made at run-time.

3 A Continuation-Based Service Provider

In order to be able to control the execution of services, we propose a service provider architecture supporting web service *preemption* facilities for the suspension and resuming of service instance execution based on *continuation* management [6].

The main feature of this architecture is the possibility to use a set of primitives to control service execution, i.e. to submit, suspend, resume web services. The primitives allow to specify QoS parameters affecting the service scheduling

decisions (priorities, cpu resource access, and so on) and to change them at runtime. These primitives are also exposed as web services that can be invoked by any client program acting on behalf of service brokers and/or service consumers and they are accessed in a distributed setting through XML/SOAP messaging.

In this way we provide a uniform mechanism to control service execution policies at two levels: a *low level* where service providers control their own service execution according to local policies, and a *meta-level* where global decisions need to be made for the coordination of services provided by different providers. In the latter case, we foresee that the primitives to control service execution can be used by a *metascheduler* [9,10] in charge of coordinating the local schedulers of different providers.

The primitives are based on the possibility to capture the state of a computation by means of continuations, so the computation can be suspended and resumed later on.

3.1 The Notion of Continuation

A *continuation* relative to a point in a program represents the *remainder of the computation* from that point [6], 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 [11]. 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.

In our implementation we used Stackless Python [12], 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. Furthermore, Python is one of the languages that provides a satisfactory support of libraries and tools for the development of web services [13].

3.2 The Service Provider Architecture

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.

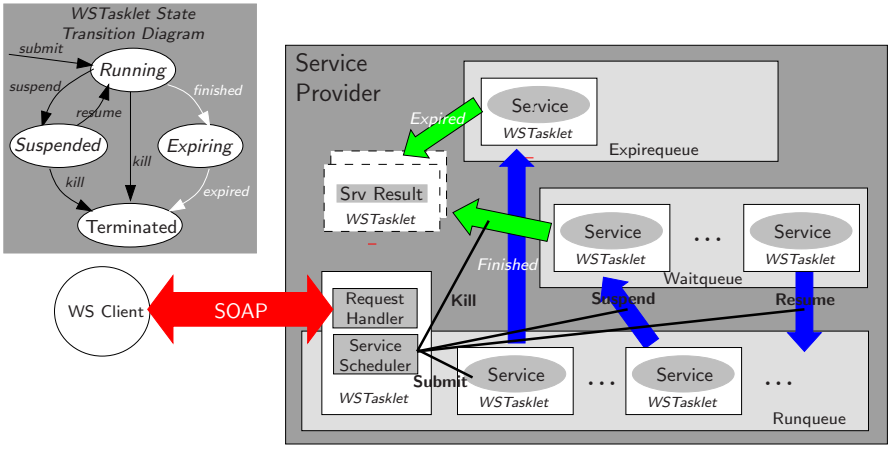

Fig. 1. Services provider architecture and service state transition

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 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 [14] 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 [15,16] 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 they usually obey to the *Inversion of Control* (IoC) programming pattern [17,18] 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.

The client program can represent either a service consumer that requires a service result, or a metascheduler or a service broker 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.

The service provider architecture is depicted in figure 1 and it is implemented in Stackless Python. The provider is represented by a *service container* consisting of a pool of lightweight threads, named *WSTasklets*, implemented by using continuations. WSTasklets execute concurrently in the same Python interpreter process.

WSTasklets are threads wrapping up service functions that represent web service *operations* in WSDL [19]. Web service operations are given as parameters to a WSTasklet wrapper and executed within its context (see figure 1). The wrapping guarantees the required functionalities to suspend/resume web service operation executions by means of the Stackless Python continuation storing and resuming support.

A WSTasklet, and hence the corresponding service, can be in the following states:

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

There is a special WSTasklet, always in *running* state, that is the main thread of the service provider. It interleaves messaging and scheduling activities by means of two modules: the *Request Handler* and the *Service Scheduler*. The *Request Handler* deals with probing incoming SOAP messages; the *Service Scheduler* module controls WSTasklet state transitions by means of a set of primitives: *submit*, *suspend*, *resume*, *kill* (black arrows in the diagram shown in figure 1). The *submit* primitive creates a new WSTasklet, wrapping up a specified service operation and puts it in the *running* state.

The Service Scheduler maintains three queues to manage WSTasklets in different states:

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

Waitqueue - it contains all WSTasklets 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 WSTasklets that finished executing and that are waiting to be garbage-collected by the system. They are maintained in this queue in order to keep the computation results that can be later on collected by service consumers with SOAP requests within a given *expiration time*. 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 re-organizing the Runqueue. 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.

3.3 Asynchrnonous Client-Provider Interaction

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 [20]. 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/reply operation mode with functionalities to suspend and resume web service execution. The client-provider asynchronous interaction pattern is described in figure 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 figure 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

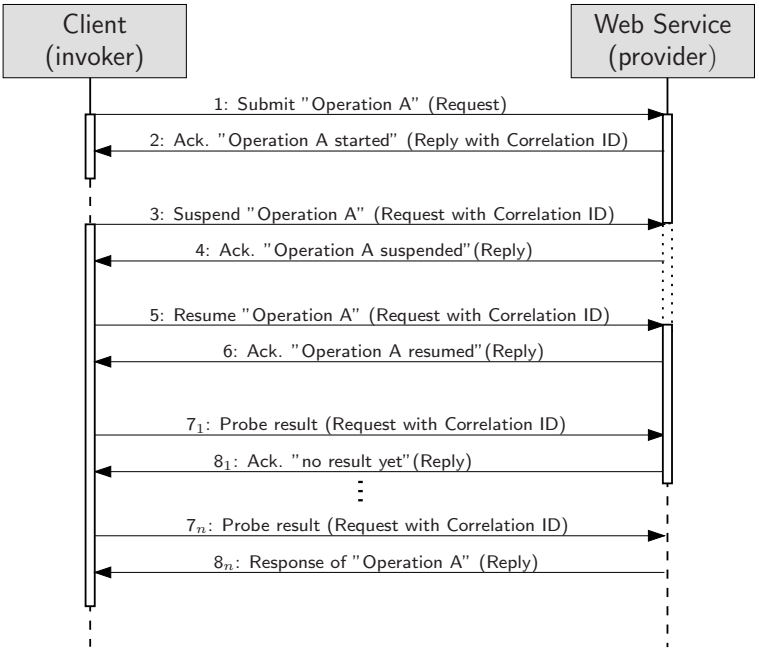

Fig. 2. Asynchronous request/response operation with polling and suspend/resume facility

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 [21,22]. Approaches differ for the particular protocol adopted (JMS, SOAP, etc.) and/or the mechanisms used to implement message correlation. In our approach correlation is explicitly included in SOAP message bodies as shown in figure 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.

To perform suspend 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

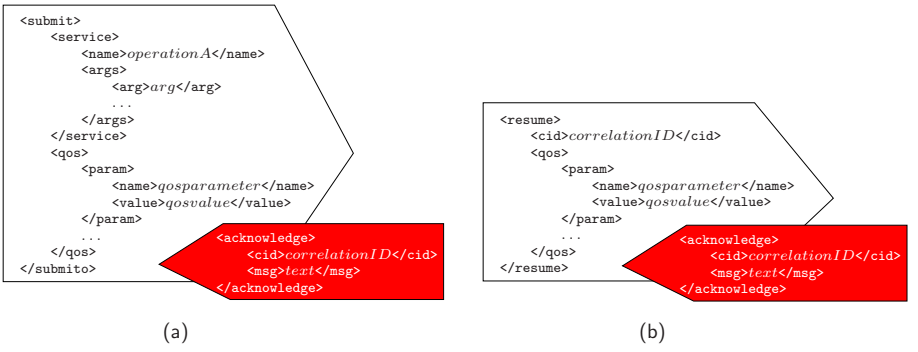

Fig. 3. Service control primivites syntax: (a) `submit`; (b) `resume`

the specified correlation ID. It then resumes the web service operation and sends back to the client an acknowledge.

As described in figure 3(b), also the `resume` request includes specifications of QoS parameters. This means that in our framework a service execution can 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 figure 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.

4 Application Scenarios

Economy-based grid environments require more sophisticated scheduling approaches able to deal with several optimisation functions: those provided by the user with his/her objectives (e.g. cost, response-time) as well as those objectives defined by the resource providers (e.g. throughput, profit, CPU utilization). It is also important to be able to change these objectives according to new conditions occurring when fulfilling service requests.

Our framework provides this flexibility since 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 service execution suspension and resuming.

For example, let's suppose that a client requests the execution of a service with an associated *cost* representing an estimate of the amount of resource the client expects the web service will consume (e.g. CPU time). Thus the *cost* may correspond to the maximum execution time guaranteed by the provider, according to a previous agreement with the consumer. In this case the service request can be `submit`ted with a `qos` parameter value corresponding to the *cost*. The Request Handler module of the service provider receives the request with the associated cost and the Service Scheduler starts its execution in time-sharing.

While the time-sharing quantum is fixed for all services, the amount of quanta available for the service is limited by the *cost* parameter. If the service has not completed before this time, the Service Scheduler suspends it.

Depending on the client-provider agreement, the suspended service can be rescheduled for execution only after all tasks have finished spending the time slices they paid for, or it can be resumed only if the client pays an additional cost. In the latter case the client `resume`s the operation with a new value of the `qos` parameter guaranteeing an additional execution time slice for it. In this scenario the Service Scheduler makes scheduling decisions according to the received requests and it controls the execution of the required services accordingly.

The possibility of changing the scheduling policies at run-time can be exploited also when the cost of service execution is dependent on the priority the provider assigns to the service it provides. We assume that when consumers request a service execution, the provider charges for the service a cost according to the priority which the service will be executed with. In this scenario it is possible for consumers to increase/decrease, at any time, the money they are willing to pay for the the required services. In such a case the provider should respectively increase/decrease the priority it assigned to the service execution requested by that consumer. In this scenario the `qos` parameter included in service submission represents the priority assigned to it; if the consumer wants to change it, it request the provider first to suspend the service and then to resume it with a changed priority value.

Of course, in both scenarios it is up to the local Service Scheduler, according to the adopted policy, to account for the cost/priority change request and to fit it in the multitasking environment.

5 Conclusions

In this work we propose a service provider architecture based on continuations storing and management to provide primitives to control web services execution and to implement service scheduling policies. The primitives are also offered by the service provider to external client applications via web service (SOAP/WSDL) interfaces.

With this approach we may implement the service execution policies at two levels: the lower level relies at 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.

Community Scheduler Framework (CSF) [23] is an infrastructure providing facilities to define, configure and manage *metaschedulers* for the grid. Metascheduling is conceived as a higher level of scheduling decisions to coordinate local schedulers (PBS [24], LSF [25]) on hosts and clusters in a grid environment.

Our approach has similarities to CSF. Both solutions pursue the scope of providing new high-level scheduling functionalities both to service consumers and to the development of metascheduler middleware. CSF functionalities mainly target configuration and management of scheduling policies and their coordination in a grid environment.

CSF provides scheduling functions both via web service (SOAP/WSDL) interfaces and by means of client interfaces or shell commands. Like CSF, our framework allows service execution control and scheduling queues configuration and management through SOAP-based messaging interaction.

Although both approaches support suspension/resuming facilities, CSF applies them to control jobs, i.e. processes running in the hosting OS environments. CSF defines high-level scheduling services (in Java) to drive and translate consumer requests into job-control commands implemented at lower level in the scheduler running on the target hosts or clusters (as PBS and LSF). Therefore CSF job-control functionality depends on the underlying OS layer service compatibility.

In the present work we developed a continuation-based service provider featuring programmable and full control of generic web service executions. The service execution control is not provided at operating system level, but at application-level through the use of continuations. This choice makes the framework flexible and easily portable across heterogeneous programming environments with support of continuations since there is no direct dependence with the operating system. The proposed framework represents a programming platform for developing and experimenting with service scheduling policies in different service-oriented applications.

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. Wooldridge, M.: Engineering the computational economy. In: Proceedings of the Information Society Technologies Conference (IST–2000), Nice, France (2000)
3. 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)
4. 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)
5. Foster, I., Roy, A., Sander, V.: A quality of service architecture that combines resource reservation and application adaptation. In: Proceedings of the 8th International Workshop on Quality of Service (IWQOS 2000), Pittsburgh, USA (2000) 181–188
6. Friedman, D.P., Haynes, C.T., Kohlbecker, E.E.: Programming with continuations. In: Program Transformation and Programming Environments, pp. 263–274. Springer, Heidelberg (1984)
7. Foster, I., Kishimoto, H., Savva, A., Berry, D., Djaoui, A., Grimshaw, A., Horn, B., Maciel, F., Siebenlist, F., Subramaniam, R., Treadwel, J., Reich, J.V.: The open grid services architecture, version 1.0. Technical report, Global Grid Forum Informational Document (2005)

8. Czajkowski, K., Foster, I.T., 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)
9. Wäldrich, O., Wieder, P., Ziegler, W.: A meta-scheduling service for co-allocating arbitrary types of resources. In: Wyrzykowski, R., Dongarra, J.J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 782–791. Springer, Heidelberg (2006)
10. Vadhiyar, S., Dongarra, J.: A metascheduler for the grid. In: Proceedings of the 11th IEEE Symposium on High-Performance Distributed Computing, IEEE Computer Society, Los Alamitos (2002)
11. Di Napoli, C., Mango Furnari, M.: A continuation–based distributed lisp system. In: Proceedings of the First International Conference on Massively Parallel Computing Systems, pp. 523–527. IEEE Computer Society Press, Los Alamitos (1994)
12. Tismer, C.: Stackless python (2007), http://www.stackless.com
13. SourceForge.net: Python web services (2007), http://pywebsvcs.sourceforge.net
14. 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), pp. 311–318. IEEE Computer Society, Washington, DC (2004)
15. The Apache Software Foundation: Apache web services project - axis (2007), http://ws.apache.org/axis
16. IBM developerWorks: WebSphere (2007), http://www-128.ibm.com/developerworks/websphere
17. Johnson, R.E., Foote, B.: Designing reusable classes. Journal of Object-Oriented Programming 1(2), 22–35 (1988)
18. Fowler, M.: Inversion of control containers and the dependency injection pattern (2004), http://www.martinfowler.com/articles/injection.html
19. Booth, D., Liu, C.K.: Web services description language (wsdl) version 2.0 part 0 primer (2007), http://www.w3.org/TR/2007/PR-wsdl20-primer-20070523
20. Adams, H.: Asynchronous operations and web services, part 2 (2002), http://www-128.ibm.com/developerworks/library/ws-asynch2/index.html
21. Swenson, K., Ricker, J.: Asynchronous web service protocol (2002), http://xml.coverpages.org/AWSP-Draft20020405.pdf
22. Sun Developer Network: Developing asynchronous web services with java message service in sun java studio enterprise 7 (2005), http://developers.sun.com/prodtech/javatools/jsenterprise/reference/te chart/jse7/asynch.html
23. Platform: Open source metascheduler for virtual organizations with the community scheduler framework (csf). Technical report (2007), http://www.cs.virginia.edu/~grimshaw/CS851-2004/Platform/CSF_architecture.pdf
24. Open portable batch system (2007), http://www.openpbs.org
25. Load sharing facility (2007), http://www.platform.com