Developing a Secure Grid Computing Environment Shell Engine: Containers and Services

Mehmet A. Nacar[1], Marlon Pierce[1] and Geoffrey C. Fox[1]
[1]Community Grids Lab, Indiana University
Bloomington, IN 47404

Abstract

We describe the design and features of our Grid Computing Environments Shell system, or GCEShell. We view computing Grids as providing essentially a globally scalable distributed operating system that exposes low level programming APIs. From these system-level commands we may build a higher level library of more user-friendly shell commands, which may in turn be programmed through scripts. The GCEShell consists of a shell engine that serves as a container environment for managing GCEShell commands, which are client implementations for remote Web Service/Open Grid Service Architecture services that resemble common UNIX shell operations.

Keywords - grid computing environments, web services, grid portals,

1. INTRODUCTION

Grid Computing Environments (GCEs) [1] provide a user view of computational Grid technologies. GCEs are often associated with Web portals, but in general may be any type of client management environment. GCEs also come in two primary varieties: Problem Solving Environments (PSEs), which provide custom interfaces for working with specific sets of applications, visualization tools, etc; and shell-like system portals, which provide direct access to basic commands such as file manipulation and command execution. In Ref. [1] these latter portals are referred to as "GCEShell" portals. This paper is extended version of ref [2].

GCEShell environments may however be separated from specific user interface rendering. We consider here a general engine for managing Grid Web Service clients. This GCEShell engine, which we initially implement as a command line interface, is inspired by the UNIX [3] shell environments, which provide a more user friendly environment for interacting with the operating system than programming directly with system level libraries. We view the emerging Open Grids Services Architecture (OGSA) [4,5] and Web service [6] infrastructures as providing a global operating

system, extending ideas such as originally incorporated in the Legion system [7]. As with other operating systems, most users should not be expected to program at the system level. Instead, we see the need for a command hosting and management environment that supports a number of useful shell-like commands: commands for listing and manipulating remote files, commands for listing system resources, and so on. These shell commands should also support simple composition and workflow through linkages (such as pipes and redirects) and ultimately through scripting environments.

2. GCE SHELL ENGINE

The shell engine is the core application that interprets commands, runs client commands, communicates with the servers (applications and registries), and manages application lifecycles. The GCEShell engine essentially serves as a container for client applications, analogous to server-side hosting environments [4].

Our initial implementation of the GCEShell interface is as command line interface similar to UNIX shells. It manages user command entries and gives results back to standard output. Also, command instances and each command's status are stored at this stage. The shell engine spawns a new thread for each shell command so that user can coordinate each single command entry by itself using several commands like kill, ps, history, exit.

The GCEShell involves both local and remote commands. The shell engine makes several different manipulations, like if the given URI is local, the engine directly makes related command calls. Otherwise, WSDL interfaces [8] are discovered at the given URI that gives service description. According to that information, service requests are made at given SOAP endpoint by using SOAP protocol [9].

The shell engine aggregates all the objects that perform functionality to the shell container. These objects are commands, tasks, communication with servers, and workflows. In other words, the shell engine negotiates with servers, manages application lifecycles, discovers services and communicates with remote services. If the worst case happens, like a service provider is down, there are several cases to overcome that situation. First, the request may be repeated according to service priority and importance. This time slice should be restricted in terms of system performance dynamically. Finally, if a part of command arguments fails, either entire command fails or partial result given on demand.

Figure 1 shows the shell engine's principal components. Each component in our implementation is a Java interface with an implementing class. Arrows in the figure indicates communications between modules. Broken arrows show relationship with Exception Handler. Bold arrows indicate execution steps. Following this design simplifies development of the more complicated components (such as the command line parser) and also allows future reimplementation by other developers.

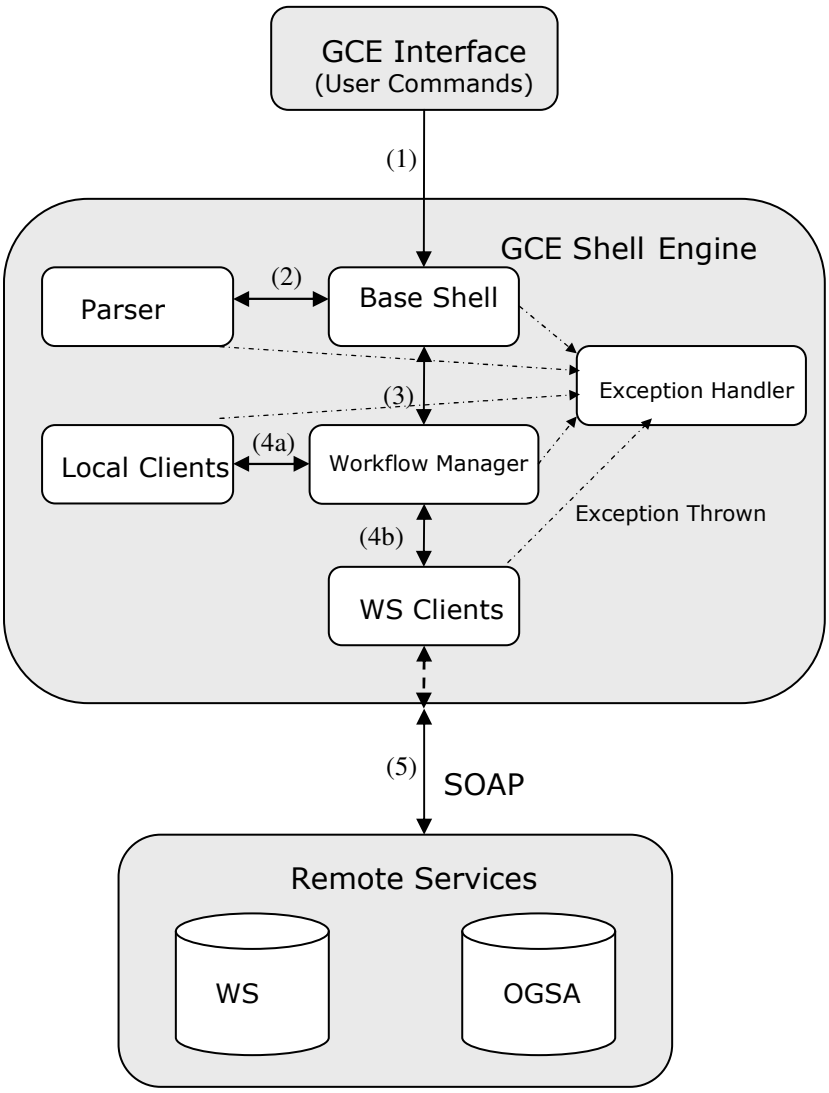

Figure 1: GCE Shell execution steps and block diagram

Figure 1 also indicates the steps followed after a command is issued to the shell. Here we summarize the execution steps for processing a user command. In Step 1, the user enters a command. That command is caught by the shell engine and divided into the tokens by the Parser in Step 2. The Parser is responsible for checking the syntax of command line. We follow a typical shell-like syntax for command the command line: commands, attributes and options. The Parser can also distinguish delimiters between multiple commands and remove them. In our test implementation we mark each item in the command line as a token and collect them in a hash table. Parsing along syntax rules is a quite well known subfield of computer science and is reviewed in [10]. We are in the process of replacing the test implementation with a more formal parsing engine that

will produce a parse tree at the end of Step 2. We are currently evaluating third party parser packages such as ANTLR [11] and Java implementations of the DOM [12].

The results of the parsing are next passed to the Workflow Manager in Step 3, which is responsible for executing the parsed command line. The test implementation represents this as a hashtable, but we are in the process of converting this to use a "parse tree" object as described above. The workflow manager is then responsible for managing the execution of the clients and their arguments that it receives from the parser, Step 4. These clients may be either local applications (such as shell history commands) as shown in Step 4a, or clients that must connect to remote applications, Step 4b. In the latter case, the client must then interact with the remote Web Service, Step 5. In both cases, the client applications implement a common shell command interface (see next section). Each command line is represented as a single object and is executed by a single thread. The Workflow Manager is responsible for creating new threads for each command line it receives (represented as a parse tree). Each thread in turn must walk the tree and identify commands (nodes that only possess leaves) and create "command objects" (detailed below) to execute the specific shell commands. These command objects are executed in sequence if the command line has more than one command. Exceptions may occur at numerous places in this system and are handled by the Exception Handler. We address these issues below. We implemented event system model for this design that works in between workflow manager and exception handler.

WS Clients cover inspection of services, discoveries and service requests for grid services. First of all, the service in the specified URI is inspected and WSDL interface is found. After that the engine creates client stubs for that service and makes remote procedure requests from SOAP endpoint. If an exception is thrown, the exception handler deals with that. Unless it is succeeded, the service and so the command fails and gives an exit code and message as error.

The Workflow Manager traces all parts of command to be completed successfully. It combines completed parts in accordance with command line and finally outputs are sent to GCEShell interface.

The base shell context is responsible for creating child shell contexts to hold individual commands and for managing the lifecycle of these child contexts. It also manages communications between the child contexts; that is, the pipes and redirects are functions of the base context. Child context threads must block until the command completes. If this is not implemented in the shell command itself (the client is decoupled from the server and exits before the server process completes) then the child context will need to implement a listener that gets notified when the command completes on the server.

The GCEShell engine's design must provide a simple, well defined mechanism for adding new shell commands. Command prototypes and base shell connections is specified. In future, we need to add dynamic class loading so that grid users could place new features on run time. However, a user can add, remove or replace a command implementation by updating properties file and providing appropriate classpaths.

3. GCE SHELL COMMANDS

GCEShell contains command and context interfaces which must be implemented by new commands. Single command objects are derived from base shell so that singleton carries all requirements needed. To simplify the loading and management of child components by the Base Shell, we define a common interface for both local and remote shell commands. The shell interface has the following methods.

- For each attribute, write accessor (get and set) methods.
- For execution directions, execute() method.
- To kill the command or process, exit() method.
- Allowing process to sleep, suspend() method.

Commands are similar to jakarta-ant tasks and JXTA ShellCommands. A task can have multiple attributes. The value of an attribute might contain references to a property. These references will be resolved before the task is executed. A service command can have a set of properties. These might be set in the properties file by outside the base shell. A property has a name and a value; the name is case-sensitive. Properties may be used in the value of command names.

Commands will be well defined by interfaces, so a developer might want to add more commands. To do that, it is needed to implement classes that inherited from that interface. The only thing is to plug that new command into shell container, updating property file giving command name and package name pairs. Removing a command or replacing new ones are need similar configuration above.

4. EXCEPTIONS AND EXCEPTION HANDLING

The most crucial expectation from any kind of shell is to run forever, unless a user exits it. GCEShell has modular and integrated design, to prevent conflicts in terms of using services and crashes. It is especially important for GCEShell, which interact with distributed resources that may become unavailable for a number of reasons. It is therefore important that the GCEShell have robust exception handling.

ParserException is thrown, when the command stream consist of unknown syntax parameters or characters. Thus, the user can correct words or syntax. If the given command name is not specified in the properties file, CommandNotFound exception is

thrown. LocalClientException is thrown when a local command has an internal error, perhaps caused by improper input. Finally, RemoteClientExceptions may be thrown either if the remote command was sent improper input or the remote server is unreachable for any number of reasons.

Each exception interacted with related module, but most of them are handled by the workflow manager and base shell. There are mechanisms to deal with exceptions. For example, when remote client exception thrown, the request will be made in a loop so far to get service or exceed the timeout. Also, timeouts can be done several times.

5. INFORMATION SYSTEM REQUIREMENTS

The Shell Commands are responsible for discovering the service that they need and for communicating with that service. However, it is possible and perhaps desirable for the shell to take over some of these responsibilities when the command is run by the shell. In this case, the command would contact the GCEShell in the service discovery phase and communicate only indirectly with the remote service, with direct communications filtered through the shell.

Workflow manager coordinates all negations with services and adjust timeouts according to priority of specified services. The purpose of involving WSIL [13] is that the base shell needs to inspect web services instantly. Likewise, gce-ls command is available remote service of the shell. In case of taking URI argument, related web service method being invoked. So, currently up and running web services reported back to the shell container. For example,

```
gce-list http://fuji.ucs.indiana.edu:8080/axis/services
```

The gce-list command examines the inspection.wsil at this location and inspects what WS running and gets back the list of WSDL interfaces.

The shell container is eligible to deal with some possible failures. If a command resulted with error or exception, workflow manager might be able to manage that in different cases. Depending on partial results, either command is terminated or request is repeated until getting the service or timeout.

GCEShell container can support Globus Security Infrastructure (GSI) [14] to access COG [15] services. Also it is compatible with GSI-enabled web services.

5.1. Information System Security

Information systems could have restricted information. To retrieve this information system users should be authenticated. Globus proxy service provides authentication using X509 certificates. When a user logs in to a GCEShell it gets globus proxy

credentials until either proxy expired or logout. Using the proxy, a client can call Cog based commands and secure web services commands.

GCEShell engine supports secure web services to access restricted services. GSI enabled security system uses Transport Layer Security (TLS). Secure connection established after user proxy and host proxiy negotiation. GSI secured channel is used to exchange critical or private information with parties.

6. STATUS OF IMPLEMENTATION

GCEShell interface is a standalone application written in Java. All commands implement the same interface and each command runs in a new thread. We have implemented the following so far: a) we are constructing the GCEShell engine, with initial prototypes; b) we are implementing an initial set of commands, which use the interface described above for clients and implement remote services using Apache Axis (http://ws.apache.org/axis); c) the shell has ability to use environment variables; and d) we are implementing an initial information discovery command, gce-list, based on WSIL. This command can be used to discover available web services and provides information to the user on locations of services. The collection of commands that we have implemented so far includes the following: gce-ls, gce-list, gce-ps, gce-set, gce-history, gce-man, gce-kill, and gce-help.

GCEShell container incorporates with GRAM services using Java CogKit commands. These commands similar to existing shell commands like gce-ls, gce-ps, gce-kill. Additional command gce-run submits remote jobs using 'globusrun' command. Each command is implemented by an RSL [16] script that generated dynamically to match with options and attributes.

There are some configuration attributes when running GCEShell engine, in order to get several optional services. While running a command, a user can choose type of services. Such parameters called service attributes that could be 'ws' or 'cog'. Web services based commands called by ws parameter, GRAM based commands called by cog parameter. By doing such kind of distinctions help to develop new services and plug into the shell engine easily.

7. FUTURE WORK: GRID COMPUTING ENVIRONMENT PATTERNS FOR SERVICE ORIENTED PORTALS

In this section, we summarize our current research efforts to build a more general portal engine suitable for service oriented grid systems and alternative display technologies.

The GCE Shell is intended to be a subset of general portal architectures. One of the limitations in computing portals previously has been that they have been too dependent on presentation technology, specifically browsers. However, Grid Computing Environments may be implemented using Java Swing and similar user interface components, or with Shell like interfaces (as we have discussed here) that interact with

remote services through a service layer. All these computing environment systems have the same architecture [17], and thus in our discussions below, we will refer to all of these computing environments as "portals" and do not limit our discussion to Web browser clients.

It is important that we review all such Grid portal systems (browsers, shells, and GUIs) in terms of two important architectural developments: portlet components and service oriented architectures (SOA) [18]. These provide two complementary developments. Portlets provide reusable portal components that may be shared between projects, with desired user capabilities plugged into the portal in a well-defined way. Web service architectures provide the means for separating the portal's functionality from its display. We next examine the implications of these two trends for portals.

Portlets are software components that process user requests and generate responses in the form of display instructions (typically HTML, but this may be generalized to XML and thus to various display technologies). Each portlet corresponds to a user capability, so for example we may have job submission portlets that manages the user's request to launch an application and ties it to one or more services that will actually implement the action.

Portlets are currently a part of Sun's Java technology and are standardized through JSR 168 [19], but the concept is general and not inherently language specific. General Web Service for Remote Portlets (WSRP) specifications [20] exist for portlets that do not depend on specific programming languages. Portals following the portlet component approach are thus composed of portlet components.

We may take the JSR 168 specification as a fixed point for development, but it is important to understand what is out of scope in the specification. The JSR 168 specification must be placed into the context of two other entities:

1. The Portlet Container: provides the runtime environment and manages the lifecycle of portlet components. Portlet containers manage the requests and responses needed by the portal.
2. The Portal Display Manager: manages the display of the portlets. For browsers and GUI applications, this may be an aggregated interface that combines several different portlet displays into a single display.

Unlike the portlet API, implementation guidelines for two entities are non-standard: we may treat JSR 168 engines such as Jakata-Pluto as black boxes. Clearly also many features out of scope in the current portlet API (such as portal login and inter-portlet communication) may be implemented in various ways that may not be shared between portal implementations.

These two entities are typically combined in current practice: the Open Grid Computing Environments project (OGCE) [21] provides a typical example of such systems. However, as we have pointed out in this paper, we must also support alternative user interfaces (such as shells). Clearly, the Portlet Container may be understood as a generalization of our Shell Engine, while the Display Manager needs to be generalized to the point that it can support a wide range of display mechanisms. A GCE Shell Display Manager will be one such mechanism that we must develop, but we also will draw upon experience building browser portals to define more general Display Managers.

Portlet containers must provide the runtime environment for managing portlet components. This effectively means that the portlet container must house all of the "portal services": create and destroy portlets, send messages between portlets, manage access to user requests and responses, manage identity and authentication, manage roles and access restrictions to content and services, and so forth. The portlet container must not only manage portlet lifecycles, but it must also manage service lifecycles and service object lifecycles. "Service objects" here mean entities such as user identification objects created when the user logs into the portal.

The primary challenge for Grid portal services is defining the relationship between the portal services and service oriented architectures. Typically, in systems such as Jetspeed and variants like OGCE, the portlet container engine implements all services locally. Specific implementations may in fact be client applications to remote services (such as web service stubs or Java COG calls). Such systems must be designed carefully, however, as we may easily design tightly coupled systems that share objects all within the same Virtual Machine. Such implementation have short term advantages but suffer, in the long run, from two inherent limitations: the service implementation is too intimately tied to the container implementation and the service implementation is language dependent.

These are critical limitations to the computing portals for two reasons:

1. Portlet container implementations are non-standard, so services cannot be easily reused between different container implementations. In particular, it appears likely that containers will undergo significant development as they seek to provide capabilities outside the scope of JSR 168 standards. Thus it appears likely that portlet container services, if too tightly coupled to the portal container, will have to be frequently reimplemented and cannot be reused between projects.
2. Grid portal systems are by their nature distributed, so it is often the case that portlet services need to be invoked following external Grid events. For example, a user's job completes and the user's job management portlets need to updated to show that this has occurred and data may be downloaded. Portlet services directly

implemented and accessible only through the portlet container make this sort of event updating extremely unwieldy.

Both of these limitations may be overcome if we adopt a Service Oriented Architecture (SOA) for the portlet container. In this approach, the portal container does not and should not implement services directly. Instead, "portal services" inside the portlet container are in fact only "Requester Agents" in SOA terminology. That is, they manage client stubs. The portlet container is thus in part a Requester Agent management system.

By removing service implementations from the engine and specifying them as web services, we may provide a clean separation between implementation (which may change) and interface (which should be less time dependent). Portlet containers should then be programmed to these service interfaces, but as portlet containers themselves evolve, we do not lose our time-tested service implementations. Furthermore, separating the portal services from the container provides a natural solution for Grid systems, in which events may be generated externally to the portal engine. In Figure 2, services such as event logs can have multiple equivalent requester agents (service stubs): those inside the portlet container and those running externally. Incoming Grid event postings thus do not have to go directly to the container's requester agent (a complicated process) but rather through an equivalent external agent. In this case, external agents are utilized by an event broker.

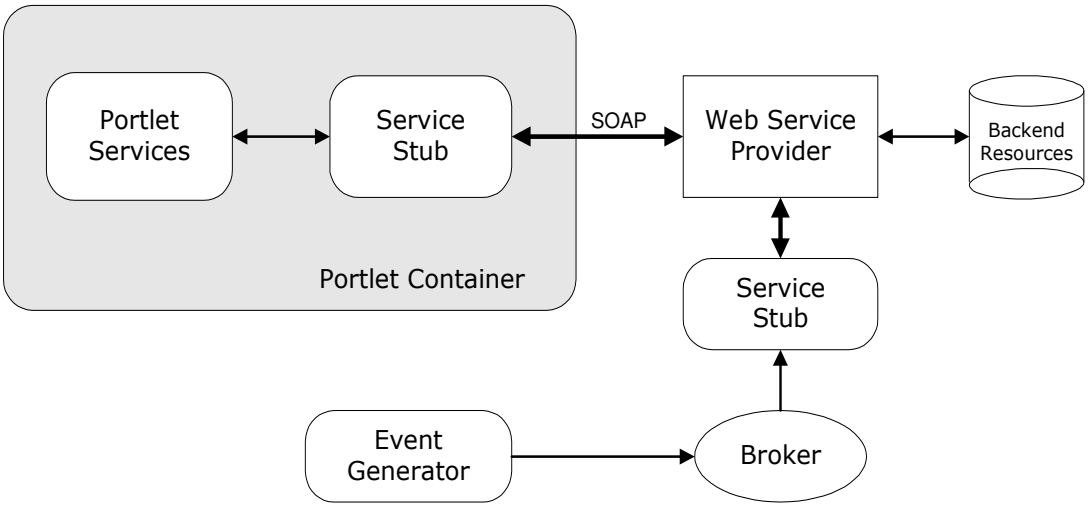

Figure 2: Portlet services interacts with external event generators using web services

Portlet containers provide two important capabilities needed by SOAs: session/state management and service orchestration. These are difficult in distributed object and service systems, but implementing then within the portlet container provides an elegant

solution. The requester agents living in the container (all in the same JVM) acting as proxies to remote services, maintain the state that their corresponding remote services do not actually posses. Service orchestration (effectively, distributed programming) is simplified by programming to the requester agents within the container and letting the requester agents manage the remote invocations.

REFERENCES

1. Fox, G. C., Gannon, D., and Thomas, M. (2002). A summary of grid computing environments. Concurrency and Computation: Practice and Experience. Vol 14, No 13-15.

2. Nacar, M., Pierce, M., Fox, G. C. (2003). Designing a grid computing environment shell engine. Proceedings of the 2003 International Conference on Internet Computing.

3. Kernighan, B. W. and Pike, R. (1984). The unix programming environment. Prentice Hall, Englewood Cliffs, NJ.

4. Foster, I., Kesselman, C., Nick, J., and Tuecke, S. (2002). The physiology of the grid: an open grid services architecture for distributed systems integration. Open Grid Services Infrastructure Working Group, Global Grid Forum.

5. Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S., Kesselman, C., and Vanderbilt, P. (2002). Grid service specification. Global Grid Forum Recommendation Draft.

6. Champion, M., Ferris, C., Newcomer, E., and Orchard, D. (2002). Web services architecture. W3C Working Draft. Available from http://www.w3.org/TR/ws-arch/.

7. Stoker, G., White, B.S., Stackpole, E., Highley, T.J., Humprey, M. (2001). Grids: harnessing geographically-separated resources in a multi-organisational context. High Performance Computing Systems.

8. Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. (2001). Web service description language (WSDL) 1.1. W3C. Available from http://www.w3c.org/TR/wsdl.

9. Box, D., et al (2000). Simple object access protocol (SOAP) 1.1. W3C. Available from http://www.w3.org/TR/SOAP/.

10. Aho, A. V., Sethi, R., Ullman, J. D. (1986). Compilers: principles, techniques, and tools. Addison-Wesley.

11. Parr, T. (1989). ANTLR: Another tool for language recognition. Available from http://www.antlr.org

12. Le Hors, A., Le Hegaret, P., Wood, L., Nicol, G., Robie, J., Champion, M., and Byrne, S. (2000). Document object model level 2 core. W3C Recommendation. Available from http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.

13. Ballinger, K., Brittenham, P., Malhotra, A., Nagy, W. A., and Pharies, S. (2001). Specification: web service inspection language (WS-Inspection) 1.0. Available from http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html.

14. Foster, I., Karonis, N. T., Kesselman, C., Tuecke, S. (1998). Managing security in high-performance distributed computing**.** Cluster Computing. 1(1):95-107.

15. von Laszewski, G., Gawor, J., Lane, P., Rehn, N., and Russell, M. (2002). Features of the java commodity grid kit. Concurrency and Computation: Practice and Experience. Vol 14, No 13-15.

16. The globus resource specification language RSL v1.0. Available from http://www-fp.globus.org/gram/rsl_spec1.html.

17. Thomas M., Dahan M., Mueller K., Mock S., Mills C., Regno R. (2002). Application portals: practice and experience. Concurrency and Computation: Practice and Experience. Vol 14, No 13-15.

18. Booth D., Haas H., McCabe F., Newcomer E., Champion M., Ferris C., and Orchard D. (2004). Web service architecture. W3C Working Group Note.

19. Abdelnur A., Chien E., Hepper S. (2003). Portlet specification 1.0. Java Community Process Program. Available from http://jcp.org/en/home/index

20. Kropp A., Leue C., Thompson R. Web services for remote portlets (WSRP). OASIS. Available from http://www.oasis-open.org

21. Gannon D., Fox G., Pierce M., Plale B., von Laszewski G., Severance C., Hardin J., Alameda J., Thomas M., Boisseau J. (2003). Grid portals: a scientist's access point for grid services.