Designing a Grid Computing Environment Shell Engine

Mehmet Nacar, Marlon Pierce, and Geoffrey Fox
Community Grids Lab, Indiana University
501 N. Morton Street
Bloomington, IN 47404
{mnacar,marpierc,gcf}@indiana.edu

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

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.

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 [2] 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) [3,4] and Web service [5] infrastructures as providing a global operating system, extending ideas such as originally incorporated in the Legion system [6]. 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. GCEShell 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 [3].

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 [7] 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 [8].

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 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 [9]. 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 [10] and Java implementations of the DOM [11].

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. GCEShell 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 [12] 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.

6. Status of Implementation and Future Plans

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.

Future plans include support for shell command composition and scripting. One of the powers of the shell environment is that new, specialized commands may be created as needed from the basic library of shell commands. We consider this to be one possible solution for Web Service orchestration. Simple command composition can be done using redirects, pipes and tees will be interesting applications in GCEShell because these require file transfer and sophisticated lifecycle management because of commands running on base shell and services are remote and distributed.

Embedding a scripting language is also planned. This scripting language may include support for existing scripting languages such as Python, as well as XML-based workflow languages such as BPEL4WS.

7. References

1. "A Summary of Grid Computing Environments," G. C. Fox, D. Gannon, and M. Thomas. *Concurrency and Computation: Practice and Experience,* Vol 14, No 13-15 (2002).
2. B. W. Kernighan and R. Pike. The Unix Programming Environment. Prentice Hall, Englewood Cliffs, NJ (1984).
3. "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Open Grid Services Infrastructure Working Group, Global Grid Forum, June 22, 2002. Available from http://www.globus.org/research/papers/ogsa.pdf.
4. "Grid Service Specification," S. Tuecke, K Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, and P. Vanderbilt. Global Grid Forum Recommendation Draft. Available from http://www.gridforum.org/ogsi-wg/drafts/draft-ggf-ogsi-gridservice-04_2002-10-04.pdf.
5. "Web Services Architecture," M. Champion, C. Ferris, E. Newcomer, and D. Orchard. W3C Working Draft 14 November 2002. Available from http://www.w3.org/TR/ws-arch/.
6. "Grids: Harnessing Geographically-Separated Resources in a Multi-Organisational Context" G. Stoker, B.S. White, E. Stackpole, T.J. Highley, M. Humprey. Presented at High Performance Computing Systems, June 2001. Available from http://www.cs.virginia.edu/~legion/papers/HPCS01.pdf
7. "Web Service Description Language (WSDL) 1.1", E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. W3C Note 15 March 2001. Available from http://www.w3c.org/TR/wsdl.
8. "Simple Object Access Protocol (SOAP) 1.1", D. Box, et al. W3C Note 08 May 2000. Available from http://www.w3.org/TR/SOAP/.
9. "Compilers: Principles, Techniques, and Tools". A. V. Aho, R. Sethi, J. D. Ullman. Addison-Wesley. 1986.
10. "ANTLR: ANother Tool for Language Recognition". T. Parr. Available from http://www.antlr.org
11. "Document Object Model Level 2 Core." A. Le Hors, P. Le Hegaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne, W3C Recommendation 13 November 2000. Available from http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.
12. "Specification: Web Service Inspection Language (WS-Inspection) 1.0." K. Ballinger, P. Brittenham, A. Malhotra, W. A. Nagy, and S. Pharies, (2001). Available from http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html.

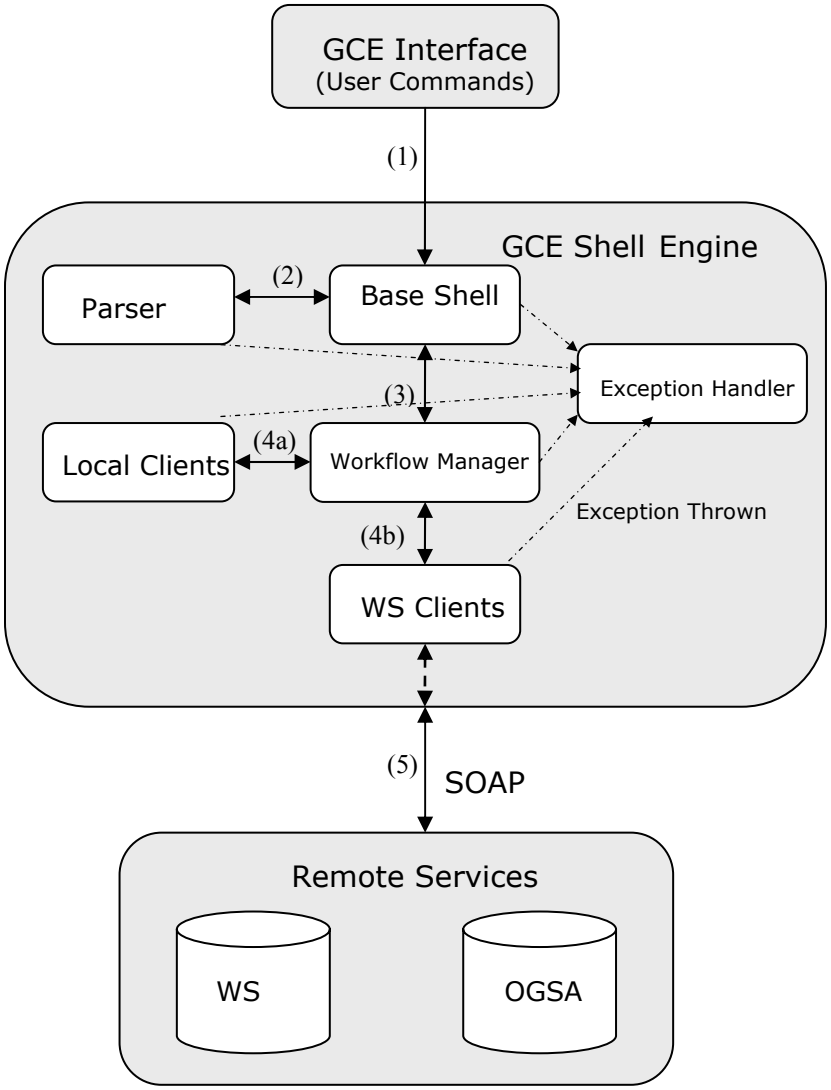

Figure 1: GCE Shell execution steps and block diagram