Experiment and Workflow Management Using Cyberaide Shell

Gregor von Laszewski*[1], Andrew Younge*, Xi He*, Kumar Mahinthakumar†, and Lizhe Wang*
*Service Oriented Cyberinfrastructure Laboratory, GCCIS, Rochester Institute of Technology, Rochester, NY 14623, USA
[1]Email: laszewski@gmail.com
†Department of Civel, Construction, and Environmental Engeneering, North Carolina State University, Raleigh, NC, USA

***Abstract*—In recent years the power of Grid computing has grown exponentially through the development of advanced middleware systems. While usage has increased, the penetration of Grid computing in the scientific community has been less than expected by some. This is due to a steep learning curve and high entry barrier that limit the use of Grid computing and advanced cyberinfrastructure. In order for the scientists to focus on actual scientific tasks, specialized tools and services need to be developed to ease the integration of complex middleware. Our solution is Cyberaide Shell, an advanced but simple to use system shell which provides access to the powerful cyberinfrastructure available today.**

Cyberaide Shell provides a dynamic interface that allows access to complex cyberinfrastructure in an easy and intuitive fashion on an ad-hoc basis. This is accomplished by abstracting the complexities of resource, task, and application management through a scriptable command line interface. Through a service integration mechanism, the shell's functionality is exposed to a wide variety of frameworks and programming languages. Cyberaide Shell includes specialized experiment management and workflow commands that, with the scriptable nature of a shell, provide a set of services which where previously unavailable. The usability of Cyberaide Shell is demonstrated using a Water Threat Management application deployed on the TeraGrid.

I. Introduction

Grid computing is a complex and diverse field where different technologies are constructed and combined to enable the use of distributed resources under administratively separate domains. Applications of Grid computing include access to large-scale computing power and storage resources harnessed from otherwise unconnected resources in an efficient and organized manner. This is commonly facilitated by middleware solutions such as the Globus Toolkit [1] and g-Lite [2]. While sophisticated middleware solutions are necessary for today's Grid computing, they add complexities to its use and seldom take into account user preferences. Thus, scientists and researchers are finding the development of applications on the Grid can be quite time consuming. We believe this is one of the reasons why Grids have not yet been as widespread as one would have hoped for.

The biggest problem with today's Grid middleware is that it does not support the infrastructure related concepts projected by Grid computing. Instead, the Grid computing environment leaves it up to a dedicated research team to develop custom tools to fulfill this need. The absence of such teams in smaller projects requires the scientist to become an expert in Grid computing. In a typical Grid environment there are obstacles that need to be dealt with. This includes task management, job scheduling, resource monitoring, and organization management. Many of these utilities could be abstracted to simplify the scientific user's experience and enable them to perform their research in a more efficient and productive environment. However, it is important that the abstraction doesn't interfere with the Grid infrastructure's overall usability.

When we go back to the roots of job coordination we find abstractions and tools that have been accepted for decades and are familiar to most scientists. One of these tools is a *shell*. A regular shell can support a variety Grid computing as tools such as Condor [3], the Globus Toolkit, and the CoG Kit [4] as they already provide command line interfaces that can be reused within any shell. However, a regular UNIX shell typically does not support the common tools of a distributed environment projected by Grids and advanced cyberinfrastructure.

Historically, a shell provides a layer of abstraction for a complicated system (such as an Operating System) to make it's use streamlined and efficient. The concept of the Cyberaide Shell is to provide this same level of abstraction to Grids and advanced cyberinfrastructure. Thus, Cyberaide Shell is a novel combination of old and new technologies so it is important to simultaneously investigate existing shells and current grid environments together.

The goal of this paper is to describe the current efforts of Cyberaide Shell. First we present some related research in Section 2, followed by our architectural design in section 3. In section 4 we describe our implementation and we give an example use case in Section 5.

II. Related Research and Technologies

To design such a system as described, we need to first understand the basics of shells, workflows, grids and advanced cyberinfrastructure.

A. Shells

One of the oldest and most common tools in computing is a shell [5]. Typically, a shell is used to abstract the details of an operating system kernel by providing a unified interface and a set of tools to the system's users. By doing so, a shield is created for the user to protect them from the intricate details of the underlying system.

Shells are commonly found as part of open source developments. Most notably SourceForge [6] has almost 600 system related shells. There are noteworthy features in some of these shells that are not part of the more common system related shell in dominant use today. These features include distributed execution, object oriented shell manipulation and semantic parsing. Some of the more elementary feature comparisons for a small subset can be found at [7].

There are a number of shells in use today. We classify these shells as system shells, distributed shells, or scripting shells. System shells, typically found in a Unix environment place emphasis on job control and file system management in a Unix like environment, distributed computing shells have extensions to execute commands easily on distributed resources, and language shells emphasize the use of a given scripting language as part of the scripting abilities.

1) System Shells: Current computer systems owe their success of usability to the original UNIX system shells. At the time, *sh* and *csh* were innovative and new ideas which made computing systems easier to use for both developers and non- developers. Many technologies today are based on shells, especially in the UNIX and Linux environments. Microsoft Windows owes much of its success on the original DOS shell of the 1980's that is now continued in the PowerShell.

A system shell provides a scripting framework to automate tasks in a coordinated way. A user can automate a number of tasks by writing an initial script and then using and modifying the code as needed.

2) Distributed Computing, Grid, Cloud, Cluster Shells: Traditional system shells were developed for use on a single system. Over time, some shells have been enhanced to execute jobs on a variety of remote systems. The frameworks to address this distributed execution are numerous, so we focus only on listing a few examples and highlighting features that are important to the design of Cyberaide Shell.

In order to easily utilize many resources, distributed computing and cluster shells provide functionality to execute jobs simultaneously on many machines. This allows a user to specify a single command, but use all of the machines in parallel. This is beneficial not only for an end-user but also for the deployment of a larger compute system.

There have been a number of efforts to try to simplify Grid computing for the scientific community. Early on, the Globus Toolkit and the Java CoG Kit provided a set of command line tools that could be used to manage jobs in a Globus-enableds Grid. This includes commands for authentication, file transfer, and job submission. In the CoG Kit we even developed a python-based Cyberaide Shell prototype, which we will replace with the effort described in this paper.

One of the important efforts in making Grids more accessible is the introduction of an ssh client that uses the Grid Security Infrastructure (GSI) as part of the authentication process [8]. However, it is not a shell and thus limited in its scope. The benefit of GSISSH is its ability to authenticate to a remote machine using GSI.

A GridShell developed by Texas Advanced Computing Center (TACC) [9], [10] is as an extension to the *tcsh* and *bash* shells. It supports access to elementary Grid commands within the shell and integrates it into the shell language. For example, the use of the file redirection syntax allows users to access GridFTP servers. To execute a command on a remote machine the keyword *on* followed by the machine name can be used to execute a job *on* another Grid resource.

One of the major deployments of Grid technologies is the TeraGrid [11], which is sponsored by the National Science Foundation (NSF). While the TeraGrid has a portal that provides project and resource management tools to the user, actual interactive job management on the TeraGrid remains an unsolved task. TeraGrid provides access to all of the tools listed in this section. In addition, it allows the access to remote machines through a terminal using mindterm [12] integrated into the portal.

Typically, a portal is orthogonal to a shell, while providing an easy method of viewing the Grid at a glance and a large array of information in an easy to use system. As such, ad-hoc scripting is often not supported easily. Instead users may achieve this by using sophisticated workflow frameworks that have a steep learning curve.

A new term used in the community is Cloud Computing. In cloud implementations, services are available to the user through predefined APIs and Web services and hide all underlying hardware technologies. This is typically accomplished by the use of virtualization [13] and exposing functionality as services. A number of tools have been developed to access clouds from the command line or from scripting languages, such as the Amazon EC2 AMI tools [14].

3) Scripting Language Shells: A wide variety of shell interpreters exist in scripting languages such as Python, Ruby, and Groovy. These languages feature an interpreter by default and can function as basic system shells if used correctly. Third party shells such as iPython [15] and IRB [16] also exist which use these languages to provide an easier interface and added support for customization through objects.

B. Workflows

While relatively little has been pioneered in the connection between Grids and shells, there is a wide array of research in constructing Workflows in Grid environments [17]. These workflow systems are overlayed on top of existing grid middleware to streamline execution of parallel and distributed processes. One such example is Karajan [18], [19] workflow engine which is effective at orchestrating jobs in a complex Grid environment. While such workflow tools are effective, there is a need to easily define and automate workflow systems.

III. Design

The Cyberaide Shell contains four high-level design components that make it unique when compared with any other current technology. The four components are object management, cyberinfrastructure backends, command line interpreters, and services.

A. Design Overview

The design of Cyberaide Shell contains components to enable access a variety of new cyberinfrastructure, Grid, and Cloud toolkits and services. This includes Globus, Condor, BOINC, Amazon EC2, and the CoG Kit. We have designed an abstraction framework that will make it possible to integrate these and other backends for future cyberinfrastructure needs.

The advanced Cyberaide Shell functionality is exposed as a service. This allows other computing frameworks to easily access our shell through independent entry point via another tool or even another programming language. Naturally, an API based binding can also be provided through our abstractions. However, we envision that most interactions will be conducted through secure Web services.

Users and high-level applications interface with Cyberaide Shell through its standardized command line interface (CLI). This interface allows users to have an easy way to manage jobs, resources, and users. In reality, Cyberaide Shell is an application that launches separate sub-applications. Each sub-application or command is accompanied by a manual page that accurately outlines its usage. Using this CLI in a consistent manner allows for detailed scripts to be executed, which further enriches the end user's experience. These scripts can contain workflows for one or more experiments a user may want to execute. Using experiment management commands a fully functional workflow system could be assembled with ease.

By using a standardized command set, we can make Cyberaide Shell easily extendible in two ways. First, we enable developers to create their own shell applications to provide specialized functionality to scientists where a general shell would leave off. Second, it enables scripts to be created that can automate complex tasks with ease. These two components are key for cyberinfrastructure developers to leverage when creating their own scientific applications.

B. Architectural Design

Based on the high-level design goals we want to interact with the Cyberaide Shell in a variety of ways. To support this, we follow a layered architectural service and component based design where different services interact with each other in predefined channels. These services are provided through a number of components. An overview of this architectural design can be seen in Figure 1.

1) Resource Layer: Cyberaide Shell is designed to use a variety of resources through different service abstractions. This may include a wide variety of Grid and Cloud infrastructure. However in this paper we focus mainly on the integration with the TeraGrid, which uses the Globus Toolkit and simple remote execution using SSH. In addition Cyberaide Shell is able to utilize Web 2.0 services such as calendars, address books, and other services provided for example by Google, and Facebook.

2) Shell Implementation Layer: Our shell is based on a simple abstraction model that has proven to be useful based on the lessons learned from the CoG Kit [20]. This includes abstractions for job submissions, file transfers, and

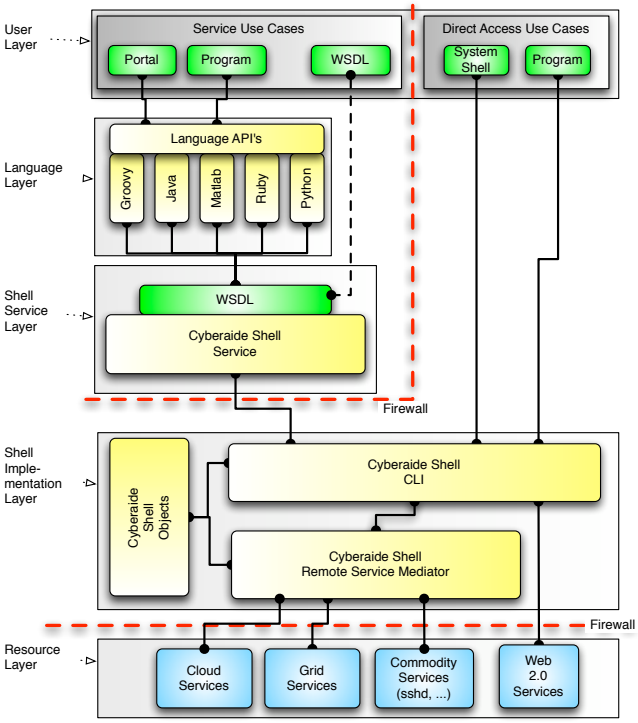

Fig. 1. Use diagram of Cyberaide Shell

authentication. In addition to these traditional abstractions we have introduced a new set of abstractions that are related to managing people of a virtual organization and jobs that are based on real time requirements that can be easily incorporated as part of a calendar system.

Naturally, the Cyberaide Shell includes a command line interface through which all commands are controlled and managed. We defined literate command objects to interface between the various components and provide a convenient mechanism of access even from other languages or frameworks.

3) Shell Service Layer: An important feature of the Cyberaide Shell is its ability to be used within a Web service. This allows the shell functionality to be also exposed to other services and application interfaces using this service. Thus a very flexible way exists to integrate Cyberaide Shell through a service-oriented architecture. This includes the possibility of accessing Web 2.0 services to significantly enhance the functionality of the shell and make these services accessible easily through a command line or other interface that is provided as part of the Cyberaide Shell.

4) Language Layer: The application interface and the service provide an ideal way to expose the functionality of the shell and its internal components through language bindings. Thus it is possible to create a direct interface for new target languages to a Cyberaide Shell Web service by providing a useful application interface. This can be implemented in many different languages such as Java, Python, Ruby, Matlab, C#, or others.

5) User Layer: Through the interplay of components and services our design enables interacting with the command shell in a variety of ways. This includes an *interactive* command line interface used by the end users, a Web service interface used by portal developers, and an application *programming interface* used by developers to interface directly with the command line shell.

IV. Implementation

In this section we present a prototype implementation of the design outlined above. The current state is a functioning Cyberaide Shell with the main components in place. Additional features such as language bindings and additional cyberinfrastructure frameworks are currently under development. We will focus only on the features that currently exist in Cyberaide Shell.

A. Command Line Interface

The Command Line Interface (CLI) is the central component of Cyberaide Shell. All incoming commands, either directly through the CLI or through the Cyberaide Shell Web service are interpreted through the CLI. Cyberaide Shell is built using Apache CLI 1.1 [21].

We have implemented enhancements to Apache CLI that make it easier to develop and manipulate command lines from within our code. This includes the definition of a framework that requires the creation of short and long command line arguments as well as a mandatory documentation for all commands.

Apache CLI is based on three stages of command line processing: (a) definition, (b)parsing, and (c) interrogation. The definition stage is based on a convenient Java API that allows us to define options and parameters to commands. The parsing stage takes a given command and splits it apart to gather precise information about the options and parameters specified. The interrogation stage is defined by the programmer and allows querying options and parameters to execute the appropriate actions.

We have provided an additional stage named the *registration stage* allowing commands to be loaded at runtime. This enables us to load in commands at startup via a specification in the cyberaide properties file, or during the execution of the CLI. Cyberaide Shell users are able to integrate commands created by others that are either distributed with cyberaide, or placed into a runtime repository.

Similar to other UNIX environments, Cyberaide Shell uses the notion of a PATH variable that allows users to place scripts in a directory accessible as part of the PATH. While using these features, we have defined a number of basic commands that simplify the use of the CLI. These commands include *man, history, cat, alias, sysinfo, and exit*. Furthermore, we provide a shell path to the underlying operating system within another shell's customary *!* character proceeding the command.

A key component of Cyberaide Shell is the explicit use of nested shells, or a shell-within-a-shell. This means that each command has its own shell interpreter and that Cyberaide Shell CLI handles the execution of subordinate CLI's. These CLI instances are dynamically loaded upon startup of the shell. This means that the implementation of dynamic loading and the creation of a nested shell is a straightforward extension.

Thus, within the main shell the user can specify either the full command with arguments, or just the command name. If a full command plus arguments are given, the CLI will interrogate the options and will execute in the way specified. If just the command name is given, a new shell for that command will open within the main shell, allowing users to issue sequentially different parameters while not leaving the context of the command. This can lead to a significant reduction in scripting for repetitive commands. Users will be familiar with the following example as it is similar to most other shells:

```
cybershell> set add -f parameter_set_a
cybershell> set add -f parameter_set_b
cybershell> set submit
```

Users can also take advantage of the nested shell feature:

```
cybershell> set
set> add -f parameter_set_a
set> add -f parameter_set_b
set> submit
set> return
cybershell>
```

In the first case execution is achieved by starting two separate set commands with an internal quit that is not visualized to the user. In the second case the contents of the super command are preserved and multiple invocations of the commands with different parameters can be issued. This allows for advanced users to write scripts that save both time and space, as well as provide context between commands.

B. Task Management and Job Execution

The current prototype of Cyberaide Shell supports two resource types for remote job execution, performed by the *submit* and *experiment* commands. The submit command simply takes in the appropriate commands to execute a given task on a specified resource. If no resource is specified a default resource is used. A *submit* is directly mapped to the resource's middleware controls to launch a new job. The execution command includes a variety of sub-commands using the nested shell approach. These subcommands include *create, add, dependency* and *submit*. They allow for a workflow to be easily created by the user and all resource management is handled within Cyberaide Shell itself. All serial and parallel tasks are implicitly defined by the nature of the workflow itself. An example of this experiment tool is shown in Section V.

We have implemented TeraGrid-based job submissions as part of our prototype. This is done using a combination of tools that are part of the Globus Toolkit. Login is managed through MyProxy [22], a x509 credential management service that works with the Grid Security Infrastructure (GSI). Job submission and monitoring is performed by the *globusrun-ws*

command, which uses the WS-GRAM protocol to submit jobs to a variety of batch queuing systems, such as Condor or PBS. The other job execution resource in the current prototype is through SSH. The SSH extension logs into a remote computer and spawns off a job as a new process. Monitoring the status and getting the results are done by separate login attempts as requested by the system.

In the design of Cyberaide Shell the difference between a *task* and a *job* are highlighted. A task is defined as an event that needs to take place. A Job is defined as an event that is scheduled to execute or is currently executing on a specific resource. Simply put, a job is a task mapped to a resource. Cyberaide Shell uses a simple state model to represent a jobs status:

- *Queued* A task has been submitted, however it has not been assigned to a specific resource.
- *Pending* A job has been assigned to a specific resource and awaiting execution
- *Running* A job is currently executing on a resource
- *Finished* A job has finished its execution and is returning to the user
- *Done* A job has finished execution and all results have been received by the user
- *Failed* A job has failed for any reason and cannot execute.

C. Service Framework

In order to enable Cyberaide Shell in higher-level services such as programming languages, APIs, or Web Portals, we use a mediator Web service to provide remote connectivity to the underlying CLI. Thus, the shell functionality can be exposed to a variety of frameworks through the mediator service. This includes also a Cyberaide JavaScript interface which we are also developing [23]. Hence, the creation of a Cyberaide Portal based on JavaScript that runs within a browser and remotely contacts Cyberaide Shell is possible.

The Cyberaide mediator Web service is implemented using the Apache CXF platform [24]. CXF uses the Jetty HTTP server [25] as a platform for hosting the Web services. Jetty is designed entirely in Java with Java-specific APIs. These APIs enable standalone Web service applications to be deployed automatically. CXF is compliant with the JAX-WS standard and implements many of the Web Service Interoperability (WSI) standards including WS-Addressing, WS-Policy, WS-ReliableMessaging, and WS-Security. Cyberaide Shell makes special use of the WS-Security standard, which uses SOAP-level encryption to ensure endpoint-to-endpoint security.

The current web services implementation works by passing Cyberaide Shell scripts in realtime via the mediator to a backend service. The following commands make this possible:

- *submit()* – Loads the script into Cyberaide Shell.
- *run()* – Starts the script loaded into the Web service.
- *kill()* – Unconditionally stops the script currently running.
- *getStatus()* – Queries for the current state of the script.
- *getOutput()* – Returns the output generated by the script
- *getError()* –Returns any erroneous output generated by the script

The Web service is deployed as an executable jar. After starting the Web service, the WSDL is published and the web service itself is ready to process requests. Clients connecting need to also implement the WS-Security standard defined by WSI over the secure HTTPS protocol and authenticate using an X.509 certificate. Authentication can be customized through local account management.

V. Use Case

We illustrate the usability of Cyberaide Shell, by applying it to a scientific application that is used for water threat management [**?**]. The problem is to determine the location of sensors in a water distribution system in order to minimize the reaction time in an emergency situation determined by sensory data.

To determine an optimal placement of the sensors, EPANET [26], a widely used water distribution network hydraulic and water quality modeling tool, is used to simulate our placements. This program uses known pipe network topology, link/node physical characteristics, and network boundary and initial conditions, to simulate the space-time variation of flows, pressures, and water quality concentrations using well-established principles [27]. The EPANET engine is available as a C language library with a well-defined API [28]. EPANET is made malleable through adaptive control by the simulation controller.

However, EPANET can only simulate one scenario within a given execution. A parallel version of the EPANET application exists [29] to simulate multiple EPANET instances at once. It includes a wrapper interface to add in multiple sources and utilizes MPI [30] to enable concurrent simulations.

The following Cyberaide Shell script describes a water threat management simulation running multiple instances in coordination.

```
cybershell> experiment
experiment> create watersimulation
experiment> add –nodes cyberaide.teragrid
experiment> add –task –f cyberaide.water
experiment> create –workflow simulation1
experiment> dependency init calc_1
experiment> dependency init calc_2
experiment> dependency init calc_3
experiment> dependency calc_1 gather
experiment> dependency calc_2 gather
experiment> dependency calc_3 gather
experiment> submit –workflow simulation1
```

Fist, an experiment is created named *watersimulation* within the Cyberaide Shell. Next, all available nodes listed in the *cyberaide.teragrid* object are added to the resource pool and all tasks within the *cyberaide.water* object are loaded into the system. The water task object contains jobs called $calc_1$ to $calc_3$, and init the initial job to obtain the input. The calculation jobs contain parallel EPANET jobs, and the task

gather aggregates the results from the different runs. Finally, the workflow is submitted and all tasks are scheduled on all available TeraGrid resources.

This example illustrates how creating workflows using Cyberaide Shell becomes a trivial process. This is important for many scientists, as they are able to quickly generate workflow systems without employing a programmer. A portal system such as the one described in the Implementation section can provide a graphical workflow tool to further simplify workflow creation. In contrast to other workflow systems however, interaction with resources and tasks can be done dynamically allowing steering of the workflow at runtime.

VI. Conclusion

Cyberaide Shell is designed to help overcome the challenges scientists face when using advanced cyberinfrastructure in today's complex computing environment. This is accomplished by combining the usability of shells with the power of grids and advanced cyberinfrastructure. Cyberaide Shell leverages a variety of novel concepts such as experiment management tools, extendible APIs using Web services, complex interaction with TeraGrid resources, and a new CLI that is familiar to most users yet easily extendible by advanced developers. A working prototype has been created that outlines the potential of Cyberaide Shell and proves the plausibility of our design.

Acknowledgment

Cyberaide Shell would not be possible without the hard work and dedication of the Service Oriented Cyberinfrastructure Lab at Rochester Institute of Technology. Contributions to this effort include Sanket Patel, Paresh Khatri, and Lihze Wang.

Work conducted by Gregor von Laszewski is supported (in part) by NSF CMMI 0540076 and NSF SDCI NMI 0721656.

References

[1] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *International Journal of Supercomputer Applications*, vol. 11, no. 2, pp. 115–128, 1997, ftp://ftp.globus.org/pub/globus/papers/globus.pdf.

[2] E. Laure, S. Fisher, A. Frohner, C. Grandi, P. Kunszt, A. Krenek, O. Mulmo, F. Pacini, F. Prelz, J. White *et al.*, "Programming the Grid with gLite," *Computational Methods in Science and Technology*, vol. 12, no. 1, pp. 33–45, 2006.

[3] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor - A Hunter of Idle Workstations," in *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*. San Jose, California: IEEE Computer Society, June 1988, pp. 104–111. [Online]. Available: http://www.cs.wisc.edu/condor/

[4] G. von Laszewski, "A Loosely Coupled Metacomputer: Cooperating Job Submissions Across Multiple Supercomputing Sites," *Concurrency, Experience, and Practice*, vol. 11, no. 5, pp. 933–948, Dec. 1999, the initial version of this paper was available in 1996. [Online]. Available: http://www.mcs.anl.gov/~gregor/papers/vonLaszewski--CooperatingJobs.pdf

[5] A. Taddei, "Shell Choice: A Shell Comparison," Guide CN/DCI/162, CERN, Geneva, Geneva, SW, Tech. Rep., September 1994. [Online]. Available: http://www.hep.phy.cam.ac.uk/lhcb/LHCbSoftTraining/documents/ShellChoic%e.pdf

[6] "SourceForge.net System Shell Categroy." [Online]. Available: http://sourceforge.net/softwaremap/trove_list.php?form_cat=294

[7] M. Froomin, "Bourn and Korn Shell Comparision Table." [Online]. Available: http://www.cs.sjsu.edu/faculty/froomin/Handouts/Shell_Ref_Page.html

[8] "GSISSH." [Online]. Available: GSISSH.http://www.teragrid.org/userinfo/access/ssh.php#gsissh

[9] E. Walker, T. Minyard, and J. Boisseau, "GridShell: A Login Shell for Orchestrating and Coordinating Applications in a Grid Enabled Environment," in *Proceedings of the International Conference on Computing, Communications and Control Technologies*, Austin, Texas, Aug. 2004, pp. 182–187.

[10] "GridShell Website." [Online]. Available: http://www.tacc.utexas.edu/~ewalker/gridshell/

[11] "TeraGrid." [Online]. Available: http://www.teragrid.org/

[12] "MindTerm," Webpage. [Online]. Available: http://www.appgate.com/index/products/mindterm/

[13] VMware, "Understanding Full Virtualization, Paravirtualization, and Hardware Assis," VMware, Tech. Rep., 2007. [Online]. Available: http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf

[14] Amazon, "Elastic Compute Cloud." [Online]. Available: http://aws.amazon.com/ec2/

[15] "iPython." [Online]. Available: http://ipython.scipy.org/moin/

[16] "RubyShell." [Online]. Available: http://www.rubycentral.com/pickaxe/irb.html

[17] I. Taylor, E. Deelman, D. Gannon, and M. Shields, Eds., *Workflows for e-Science: Scientific Workflows for Grids*. Springer, 2007, iSBN: 978-1-84628-519-6.

[18] G. V. Laszewski and M. Hategan, "Workflow concepts of the Java CoG Kit," *J. Grid Comput.*, vol. 3, no. 3-4, pp. 239–258, 2005.

[19] G. von Laszewski, I. Foster, J. Gawor, and P. Lane, "A Java Commodity Grid Kit," *Concurrency and Computation: Practice and Experience*, vol. 13, no. 8-9, pp. 643–662, 2001. [Online]. Available: http://www.mcs.anl.gov/~gregor/papers/vonLaszewski--cog-cpe-final.pdf

[20] K. Amin, G. von Laszewski, R. A. Ali, O. Rana, and D. Walker, "An Abstraction Model for a Grid Execution Framework," *Euromicro Journal of Systems Architecture*, vol. 52, no. 2, pp. 73–87, 2006. [Online]. Available: http://www.mcs.anl.gov/~gregor/papers/vonLaszewski-abstraction-jsa.pdf

[21] "Apache Commons CLI," Vesrion 1.1. [Online]. Available: http://commons.apache.org/cli/introduction.html

[22] J. Novotny, S. Tuecke, and V. Welch, "An Online Credential Repository for the Grid: MyProxy," in *Proceedings of the Tenth International Symposium on High Performance Distributed Computing (HPDC-10)*. San Francisco: IEEE Press, August 2001. [Online]. Available: http://www.globus.org/research/papers/myproxy.pdf

[23] G. von Laszewski, F. Wang, A. Younge, X. He, Z. Guo, and M. Pierce, "Cyberaide javascript: A javascript commodity grid kit," in *GCE08 at SC'08*. Austin, TX: IEEE, Nov. 16 2008. [Online]. Available: http://cyberaide.googlecode.com/svn/trunk/papers/08-javascript/vonLasze%wski-08-javascript.pdf

[24] "Apache cxf: An open source service frameworks," webpage. [Online]. Available: http://cxf.apache.org/

[25] "Jetty," webpage. [Online]. Available: http://www.mortbay.org/jetty/

[26] L. Rossman, "EPANET 2 users manual," US Environmental Protection Agency, Cincinnati, Ohio, Tech. Rep., 2000.

[27] T. Walski, "Analysis of Water Distribution Systems," *Van Nostrand Reinhold Co. New York. 1984. 275*, 1984.

[28] L. Rossman, "The EPANET programmers Toolkit for Analysis of Water Distribution Systems," in *Proceedings of the Annual Water Resources Planning and Management Conference*, 1999.

[29] S. Sreepathi, K. Mahinthakumar, E. Zechman, R. Ranjithan, D. Brill, X. Ma, and G. von Laszewski, "Cyberinfrastructure for Contamination Source Characterization in Water Distribution Systems," in *Proceedings of the INternational Conference on Computational Science, ICCS 2007*, ser. Lecture Notes in Computer Science, vol. 4487. Springer, 2007, p. 1058.

[30] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1999.