GWD-TYPE
Category: Informational
GCE-RG

Marlon Pierce, Indiana University
Geoffrey Fox, Indiana University

02-14-03

Grid Computing Environments Shells

Status of This Memo

This memo provides information to the Grid community on the topic of Grid Computing Environment Shells. It does not define any standards or technical recommendations. Distribution is unlimited.

Copyright Notice

Abstract

This document presents an overview, requirements, and features for a shell environment that may be used to access Grid technologies. Shell execution and programming environments have proven useful for providing a useful, flexible interface for users of both standalone and distributed computing systems. We thus propose this model for Grid Computing Environments, treating the emerging Grid infrastructure as a globally scalable operating system. The GCEShell then serves as a scriptable user layer that hides the underlying Grid system-level interfaces that are being developed as part of the Open Grid Service Infrastructure. Such a shell environment should possess a number of features. First, there should be a shell engine capable of loading, executing, and managing the lifecycle of GCEShell commands. The shell should also be capable of connecting multiple shell commands through such constructs as pipes and redirects. Finally, the shell itself depends upon a collection of useful commands, many of which should mimic familiar Unix shell commands as ls, cat, mv, etc. These commands are actually OGSA client applications that are loaded and managed by the shell. We anticipate the creation of useful libraries of shell commands to be an open process.

Contents

1. Overview

The user shell environment has shown itself to be a very useful approach for insulating the user from the low level APIs needed to interact with various systems. We propose that this provides the correct model for building a user environment to interact with Grid technologies. A simple command-line shell can encapsulate a number of Grid functions in a set of commands patterned after the familiar UNIX shell. Likewise, the shell architecture can serve as the foundation for graphical interfaces.

The shell environment possesses an important concept: it provides a general purpose environment for executing arbitrary shell commands, which can be written by anyone. The shell commands themselves are built on top of a well-defined, specified, and circumscribed set of interfaces to system level commands, which may be termed the "Grid kernel." After some transitory phase, we expect the Grid kernel's core system interface API to become essentially complete and fixed, with rare updates and extensions added after much debate. On the other hand, shell environments support a more informal set of user commands that may be developed with little oversight: there is nothing wrong with having two groups develop very similar shell commands (the Grid equivalent to UNIX's *less*, *more,* and *cat*, for example).

Going beyond single commands, the shell should support (via pipes and scripting) the composition of arbitrary, complex commands that are suited to particular purposes. This raises the issues of both "programming the Grid" and service composition and orchestration. As we will describe, we wish to clarify that "programming the Grid" has multiple meanings, which arise from one's location in the Grid Computing Environment stack. Likewise, service composition is an area of many possible solutions and, as yet, no clearly distinct final solution. We discuss here the possibility of a scripting-style approach to these problems that arises naturally from the proposed shell buffer layer that separates the Grid from users.

This brief report reviews the shell environment approaches of UNIX (Kernighan and Pike, 1984), JXTA (Gong, 2000), and Legion (Natrajan, Humphreys, and Grimshaw, 2001) in order to make recommendations for a proposed GCE shell environment. This shell constitutes the lowest level of the GCE and is the foundation on which we may build both command line and graphical user interfaces. The command line version of the shell would serve as a prototype and testing environment and could also be appropriate for Grid "power users". One possible realization of the graphical version may include a portlet-managing web application.

We assume that there will be a standard reference implementation for the OGSA (Tuecke, et al, 2002), and the initial capabilities of the next generation of the Grid will be basically the same as now: a collection of capabilities currently available from Globus (http://www.globus.org), SRB (http://www.npaci.edu/DICE/SRB/), Legion (http://www.cs.virginia.edu/~legion/) and other Grid infrastructure projects. The important initial difference is that these services will be described in standard ways using OGSA extensions to WSDL (Christensen, et al, 2001).

2. Grid Computing Environments and Programming the Grid

Before discussing shell examples and potential features of the GCEShell, we wish to briefly overview Grid Computing Environments and their relation to grids. GCEShell commands are intended to be useful by themselves, but we ultimately view them as the core objects that can be manipulated programmatically using one or more workflow languages. We thus wish to distinguish the ways in which one may "program the grid": by using low level APIs, which we argue are similar to operating system calls, and at a high level, developing programs that manipulate high level shell command objects.

Grid Computing Environments fulfill (at least) two functions –

- Controlling user interaction – rendering any output and allowing user input in some (web) page. This includes aggregation of multiple data sources in a single portal page.
- “Programming the Grid”

We can divide “programming the Grid” into two parts: preparation of the individual application nuggets which are associated with a single resource and then programming the integration of the nuggets together into a complete “executable”. The nugget could be the SQL interface to a database, a parallel image processing algorithm or a finite element solver. Programming the nugget is currently viewed as outside the Grid although projects like GrADS (http://grads.iges.org/grads/) are looking at integration of individual resource (nugget) and Grid programming. Here we will assume that each nugget has been programmed and we “just” need to look at their integration. This integration generalizes what is familiar from

- “Shell/Perl…” scripts in UNIX case
- Microsoft Com/ActiveX/…. Interfaces
- Possibly the programming seen in AVS, Khoros, CCA (Common Component Architecture), SCIRUN etc.

The above examples indicate that “programming the Grid” has overlaps with (distributed) object technology but in this note, we are not trying to “push a particular programming model” but rather

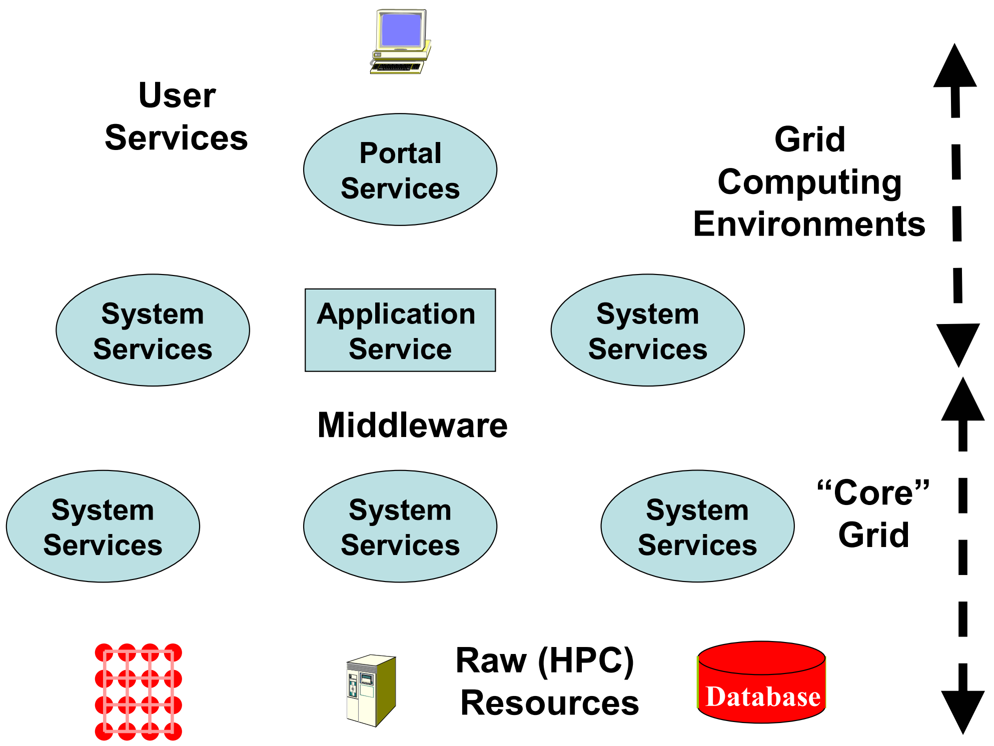

Figure 1: The logical layers of the Grid present different views.

to clarify the issues to be addressed. Although related to tasks familiar from programming PC’s or workstations, “Programming the Grid” is significantly more complicated. As illustrated in Figure 1, the “executable” (integrated nuggets) is a mixture of both system and application services; one uses system services on a single workstation but the meta-OS services of the Grid are currently expected to have programmable interfaces, whereas many of the corresponding workstation (Windows, UNIX) services are more opaque. Not only do we have the richness of both system and application nuggets, but also many Grid systems separately maintain both “real” entities (such as a software nugget) and entities representing the meta-data describing the “real” entity. We expect this separation to continue and indeed expand in use for there is a clear need to define more meta-data and it seems likely that this metadata will often be stored separately from the resource it describes.

Currently several projects in this area have a rather unclear relationship because there is no consensus as to meaning or perhaps even existence of "programming the Grid". In particular there are (at least) 3 different slants one can use in discussing "programming the Grid"

- There is the view of the user at the client device
- There is the view of the user programs running on the back-end resource(s). Note that in figure, we have the user NOT the application at the top of the hierarchy. We take a peer-to-peer view with application software thought of as resources represented by meta-data in the middle tier and an executable running on a backend computer/data resource.
- There is the view of the various system (middleware) services

One can integrate these different views in the service model – user interfaces (portals), system capabilities and applications are all services communicating via messages. The communicating services have WSDL ports but we also assume that any communication between ports is subject to some negotiation. For instance, a user device and content source must taken into account both content and client constraints (user profile) to define appropriate rendered content. Two application nuggets must take into account both needed latency/bandwidth of application and network constraints (firewalls) to decide most appropriate communication mechanism. This typically runtime specification of the implementation of a particular service-service interaction has no agreed approach. There are of course many examples of its use with particular implementation strategies. "Agents", "brokers" and "profiles" are typical of the language one often uses to describe this adaptive mechanism.

Programming the Grid consists of at least two important considerations

- The programming paradigm and within a paradigm one can choose particular languages – this could be scripted, visual, or compiled.
- The run-time library which could be largely shared between different paradigms in functionality but might be expressed rather differently

The expression of "workflow" is (part of) programming the Grid. This could be visual, an XML specification file such as BPEL4WS (Cubera, et al., 2002), Python or compiled Java. The UK e-Science effort (http://www.research-councils.ac.uk/escience/) has an ontology effort and another AI (with DPML expressing the discovery process) project. These collectively can be thought of as different and possibly very fruitful paradigms for programming the Grid. We can expect it to be useful to have multiple paradigms and multiple languages. It is perhaps useful to think of all of them as "just programming the grid" so one can more easily compare them.

3. Shell Overviews

In order to extract characteristics needed for a GCEShell system, we first review other shell systems that have proven useful.

3.1 UNIX Shells

The most widely known examples of the shell idea are the UNIX shells, which are used to hide low level system calls to the operating system kernel behind a set of user commands. While the system API for interacting with the kernel is more or less standard (Unix System V and Linux being two examples), the user environment is a catchall for various tools created on top of the kernel that have been developed by different groups. These tools may have overlapping purposes

We highlight the following UNIX concepts:

1. Shells invoke a user's commands by forking and spawning new processes.
2. The shell's basic command line features--pipes, redirects, terminators, and groupings—provide a syntax for a primitive workflow.
3. Shells use scripting languages to express workflows of shell commands, creating new, composite commands.

4. Environment variables and alias can be set to specify custom settings (such as search paths) and user defined short hands.
5. Regular expressions and wildcards are used for pattern matching within commands and scripts.

An important aspect of the UNIX shell is that it is used to handle interactions with both the file system and processes. UNIX also provides an important convention that commands that take input and generate output do so either from/to the terminal (standard I/O) or from/to files or from/to other commands.

Pieces of a UNIX pipeline are all run at the same time: ls –l | wc runs both the commands at the same time and the kernel manages scheduling and synchronization as necessary (according to Kernighan and Pike). This would be quite difficult for distributed systems. JXTA, discussed below, has unidirectional pipes to avoid synchronizing issues.

3.2 The JXTA Shell

The JXTA shell project provides a simple user environment for interacting with JXTA core platform objects. The JXTA shell provides the following:

1. A list of common shell commands
2. A simple syntax for combining commands in a single command line (pipes).
3. A history of previous commands

The shell is also extensible: new commands can be added in a well defined way: by extending the ShellApp superclass and implementing startApp and stopApp methods (along with a few other simple rules).

The JXTA Shell was designed to superficially resemble UNIX shells. However, because the JXTA Shell provides an interface to distributed peers, there must be some differences. The primary difference comes from the pipe concept, which is needed to connect applications. JXTA JXTA pipes are unidirectional and can be dynamically reconnected, as is required in a distributed, asynchronous, fault tolerant system. Also, JXTA shell provides two different types of pipes: simple pipes denoted by the "|" operator, and real JXTA pipes as described above. The simple pipes are used to connect the output of commands with local-only applications.

3.3 The Legion Shell

The Legion project, developed at the University of Virginia, researches distributed computing and Grid infrastructure. In contrast to Globus, which has aspects of client/server systems, Legion is a distributed object system in which all parts of the Grid are equal peers.

Legion's user environment is modeled after the UNIX shell, obviously extended to support distributed peer hosts. There are Legion versions of ls, rm, mkdir, and so on. Legion provides a "Context," a logical collection and organization of files that may be physically distributed among many different members of the Legion virtual computer. So the Legion shell commands act on this distributed file system. The GCE shell, on the other hand, may not want to hide this detail. The GCE ls command can be directed at a particular machine, while the Legion ls is directed at a logical collection.

Legion and Globus present contrasting views of the Grid. Globus does not hide heterogeneity of resources but instead provides a universal way of accessing them and finding out info about them. Legion provides a more homogeneous view of resources. For example, the Legion file system consists of a collection of files that might be distributed over many different resources. The user interacts with this virtual file system as if all files were local. It is debatable which approach should be adopted by a GCE shell: for example, the gce-ls command may either point to one or more physical resources, or it may point to an abstract collection that is mapped to more concrete endpoints.

4. OGSA Web Services and GCE Shells

The Open Grid Service Architecture provides some extensions to WSDL for handling services one may realistically find on a computing grid. OGSA's heart is a set of additional WSDL PortTypes and Operations that provide interfaces for creating services, registering the service instances, finding out information about services, and communicating with services. Any Web Service that aspires to be a Grid Service must include some or all of these additional PortTypes in its WSDL definition.

The basic interaction of an OGSA (or generally a Web Service) client with services is as follows: a service instance is created and runs on some host environment. The service publishes its existence and invocation interface (WSDL) to a service registry, where it is later discovered and invoked by a client using any allowed protocol binding. The client invocation may be done statically, by creation of client-side stubs from the WSDL interface that are used to generate the appropriate RPC command in the chosen protocol; or dynamically, by inspection of the service interface at runtime.

The GCE shell's basic operation is to manage OGSA client applications from the above scenario. These may be drawn from the general purpose shell toolkit commands, but the shell should be flexible enough to handle any OGSA command.

5. GCE Shell Engine

The shell engine is the core application that parses command lines, runs client commands, communicates with the servers (applications and registries), and manages application lifecycles. The GCE shell engine essentially serves as a container for client applications, analogous to server-side containers. Like server-side containers, the shell engine must have abstract containers (contexts) that can hold ShellCommand objects (see below). These abstract containers can contain other contexts and can manage the lifecycle of these objects.

Just as there are at least a half dozen UNIX shells, we shouldn't expect the GCE shell to be singular. Likewise, it seems somewhat inefficient to invent a new scripting language to support GCE shell control structures when there are so many extensible scripting languages already around. So we should expect the shell to be implemented in different languages and probably support scripts written in well-know scripting languages.

Let's now consider some of the shell engine features. The GCEShell engine should responsible for discovering the server side binding points needed by GCEShell commands. In this case, the command would contact the GCE shell in a service discovery phase and communicate only indirectly with the remote service, with direct communications filtered through the shell. The advantage of this is in customization. For example, the user may create a profile that says "I am only interested in these registries and no others, so limit the search for services to just those." Or, by setting an appropriate "environment variable" the user may specify that only a specific resource gets used.

The Shell Engine's primary responsibility is to run the "base shell context" (BSC). The BSC is responsible for creating child shell contexts to hold individual commands and for managing the lifecycle (create and destroy) of these child contexts. The BSC also manages communications between the child contexts; that is, the pipes and redirects are functions of the base context. Finally, the BSC creates default 'standard input' and 'standard output' contexts, which provide the (default) standard I/O mechanisms for other shell commands. That is, the default stdin context might be an application for handling keyboard input. As will be discussed in the section on pipes and redirects, all parts of the command line exist in individual child contexts that communicate with each other through the base shell, so default contexts can easily be replaced with other contexts.

The child contexts are analogous to UNIX processes and must present an interface for starting and stopping themselves, with these methods being invoked by the base context. The child contexts should also contain and be responsible for managing the lifecycle of individual shell commands. The child contexts are also responsible for converting wildcards, regular expressions, etc, into full expressions suitable for the shell commands.

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.

6. GCE Shell Commands

Shell Commands (GCE Shell versions of mv, cp, rm, and so on) are the actual OGSA client applications. These must also implement a common interface (see below) that contains methods for I/O and so on. Third-party OGSA clients that do not themselves implement the Shell Command interface must be placed in a generic shell command holder that minimally implements the Shell Command interface.

GCE shell commands can be grouped into several categories, including the following, with some possible examples:

1. Commands that work with local and remote files: cp, mv, rm, ls
2. Commands that work with shell processes: ps, top, kill
3. Commands that work directly with host resources: ping, netstat

GCE shell versions of type (1) and (3) commands must take URIs as arguments. For example,

```
mv  http://…/file1 http://…/file2
```

or

```
ping http://…/somehost
```

where we must make use of some URI managing system external to the shell. One point to be made is that the URI is not for the resource itself (computer or file) but for a service that interacts with the resource. Implicit also in this is that the URI system is tied to some access control mechanism: the user must have the right to move the file from one resource to the next, for example. Since there is no global DNS for URIs, we will need to provide URI support as a plug-in through an interface.

In addition to taking full XML input for instructions, the commands must also generate output. Following the JXTA example, this output should be XML-formatted. We must go a step further, however. The output of each command is really metadata about what it did, which can in turn be used as input for a second command. This is necessary in order for commands to be linked with pipes and tees and will be discussed more below.

GCE shell commands will have some additional requirements over their UNIX counterparts. The probability of failure is much higher than a shell command. A single 'move file' command relies on networks, host computers, URI resolvers and so on to work, so if any part of this fails, the whole command fails. So GCE shell commands must be both reliable (did the command *really* get executed correctly out in server-land?) and provide verbose output and logging as well as regular Boolean return conditions. The latter will be needed for the shell commands to be embedded in traditional scripting languages.

7. Basic GCE shell commands

The GCE shell is to be built out of a set of common run-time primitives. It would have some features in common with UNIX shell, as for instance file manipulation is critical both in UNIX and the Grid. There are some interesting differences. For instance the Grid must express

- The negotiated (profile-based) interaction
- Files and services at all levels of system – local client, middle-tier, backend resource
- Distinction between an object and its meta-data

Looking at primitives needed, the Grid needs to add several features such as:

- Search
- Discovery
- Registration
- Security
- Better workflow than pipe or tee in UNIX shell
- Groups and other collaboration features as in JXTA
- Meta-data handling
- Management and Scheduling
- Networks
- Negotiation primitives for service interaction

One can simplify the discussion by using a uniform service model so that files and executables are both services and not distinct as in UNIX. One probably needs a "virtual service" concept so that an individual file access is a service in the Shell even though it could be implemented differently.

Quality of service negotiations need to take place between all interacting parts. For example the shell command "mv http://…/file1 http://…/file2" could be implemented by a number of different grid service methods: gridftp, reliable file transfer, as a SOAP attachment, or even as a simple UNIX mv if both files are on the same file system. The actual service used would need to be negotiated between the parties. Web services don't currently provide this notion, but examples of negotiation protocols abound, from SIP and H.323 in AV to SSL handshakes in security.

The basic operations of the GCE shell should mimic UNIX shell commands but apply to distributed resources. Some examples would include

1. ls: this should list files for a particular node or a collection of nodes.
2. gce_cat/gce_more/gce_less: these should display the contents of a selected resource (if it is a leaf node). Note this is the distributed version of this command.
3. cd: should allow you to set the current working directory on a particular resource.
4. ping: should allow you to test the existence of a particular resource anywhere in the Grid.
5. ps: find out about your processes

Most but not all GCE shell commands will be OGSA clients. A few commands need only to be implemented in the shell itself. For example, the command "gce_cat [<some_resource>] | more" would display some file somewhere, but the pipe and more command are local and used only for formatting the output of the gce_cat command.

Just as the JXTA shell has P2P-specific like "join" and "leave", it will be necessary to identify some GCE specific commands that have no UNIX counterparts. JXTA also provides a nice model for developing new shell commands: they all must extend the same abstract class. GCE shells should do the same.

8. GCE Redirects, Pipes and Tees

The real value of the shell lies in the coupling of primitive shell commands into composite commands for specific tasks. In other words, the GCE pipes and redirects are shorthand for much more complicated Grid interactions (like gridftp). It is the pipe that is probably the most important feature that the GCE can provide. Likewise, UNIX-like tees will be needed since pipes may need to redirect single I/O to/from multiple commands.

Pipes, tees, and redirects serve as the most basic type of workflow. More complicated workflows can be set up by scripting in some well-known language like Perl or Python after appropriate bindings are made (the shell would hand off these scripts to the appropriate interpreter).

In summary, pipes are used to link commands, while redirects link commands with files. UNIX tees split output between files and standard output.

Redirects (< and >) are built-in shell commands that are bound to a particular Grid service for file transfer. The command

```
gce-ls –l http://…/somedir > http://…/somefile
```

means list all the URI children immediately under somedir and put those results in a file located by the URI following the redirect. The redirect is a Web Service client that must look up the file's URI, see which service can be used to place the file there, and then performs the action. In this case the base shell context (see above) must create a new context to hold the redirect command and replace the default 'standard output' context with the context for the redirect.

GCE pipes are more complicated than redirects. UNIX pipes connect commands, but typically these piped commands are filters for parsing the output of the first command in the pipeline. UNIX pipes work because UNIX shell commands (with a few exceptions) always are for working with files. But this is not really what would be important for the GCE shell: while sort and grep commands will have a use in the GCE shell, these are NOT OGSA clients.

JXTA pipes also are a bit different: pipes are used to manage communication between a local peer and some remote peer. The GCE shell, however, consists of local OGSA client commands that connect to distributed OGSA servers. What we really need is a way to pipe two OGSA commands together. The problem is that the interesting problems deal with scientific applications: the venerable execute-move output-visualize sequence. The problem is that the output of the grid-enabled a.out command may be spread out all over the place. So to connect a1.out to a2.out, we must have more than the standard output of a1. We also need to know all sorts of information about the application instance—all of the application metadata.

As a preliminary suggestion, we will assume that the GCE pipe should be unidirectional. We must define a "pipe binding" interface that allows various file moving mechanism clients to be bound as pipes. These means that we need the following sorts of interfaces:

1. PipeService: factory for creating different sorts of pipes
2. InputPipe: waits for input
3. OutputPipe: sends messages to InputPipes.

It is important to realize that shell applications are actually local proxies to remote services. So we need two sorts of pipes: local-to-local and remote-to-remote. The L2L pipe means that the client side output of one command needs to be sent to the client-side input of another command, which then results in some remote operation. L-Pipes should generate output and accept input in Unicode, just as UNIX.

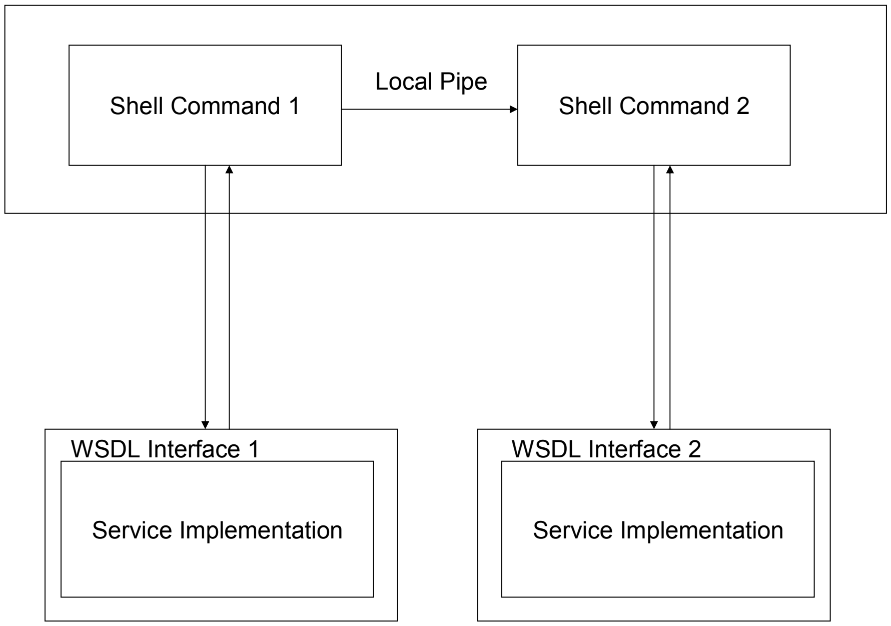

Figure 1 A local pipe transfers client side output of command 1 to input of command 2, which is used to create command 2's request.

The R2R pipe is itself a client application that gets bound to some particular file transfer implementation. An R-Pipe signifies that the server-side output of command 1 should be automatically passed to command 2. The system may be better implemented if it actually passes metadata between applications. The R-Pipe needs to pass instructions (or at least metadata) about what it did, and the second shell command can examine this metadata and decide what to do. That is, first command generates some server-side output file someplace and passes this information (not the file) to the second command. The second command then decides what it needs to do about command 1’s file.

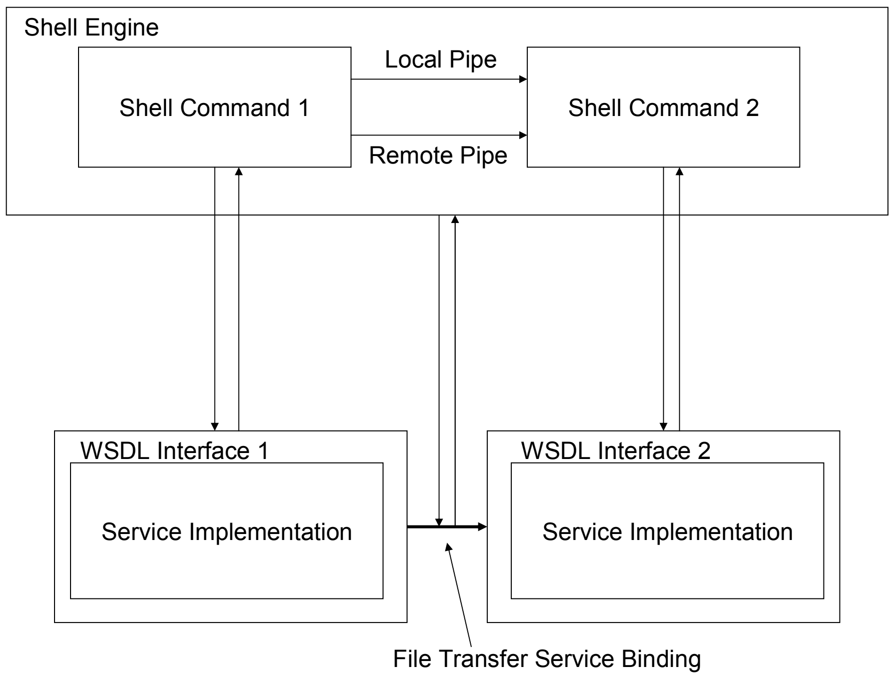

Figure 2 Shell command 1 needs to communicate both locally and remotely with shell command 2. The remote pipe is a proxy to some file transfer service.

Finally, pipe bindings could be used in a couple of different ways: communicate arbitrary output of commands in Unicode or else communicate XML metadata about the command output.

9. Features of GCE Shell User Environment

The GCEShell environment should have the following:

1. A minimal set of GCE shell commands, implementing the GCE Shell Interface/Abstract type.
2. Support for wildcards, regular expression matching.
3. Support for environmental variables
4. Support for GCE pipes and redirects.
5. A history feature
6. Process control features for starting and stopping local and remote processes.
7. Support for scripting, or perhaps bindings to specific scripting languages.

“History” means that the user can get a listing of formerly run commands and re-execute them.

Environmental variables may be extended past the simple name/value pairs of the Unix and JXTA shells. The variable name may actually be a URI, which points to an XML nugget of values, not just a single value.

In UNIX, processes are managed by the kernel, not the shell. The GCE shell, however, will need to implement a process-like system (maintaining each executing command in a Java thread, for example) that allows the user to start multiple commands, run them in background, check their status (running, sleeping, zombie), kill them if necessary, etc.

10. GCE ShellCommand Interface

It is probably a good idea for GCE shell commands to have a standalone mode: they are just OGSA service clients. However, they should also be executable inside a GCE shell context. For this to happen in a Java implementation, the GCE shell needs to load and execute the appropriate class in a new thread. The GCE shell commands thus need to implement some common methods.

Commands running in the GCE should all implement the same interface. Let's call this interface ShellCommand. Then the shell command needs to implement at least the following:

1. readStandardInput(): following Unix model, input will be assumed to be Unicode, so takes a reader.
2. writeStandardOutput()
3. writeStandardError()
4. suspend(): should temporarily stop execution of the process This may be difficult to support throughout the entire distributed system, as the real process somewhere may not have any suspend capability, but it at least can be mimicked in the shell: cache all messages and output from the command and do not deliver until necessary.
5. kill(): A process receiving this signal should terminate and also clean up all of its running instances.
6. exitMessage(): When the shell command exits, it should report an exit status code (in XML instead or else UNIX-style exit codes).

The I/O methods assume Unicode. The shell has to be responsible for figuring out the specific Reader/Writer to create. For example, if we redirect some command output to a local file, the shell would create a PrintWriter object for this. An alternative and possibly better approach is for the commands to output XML as well.

Since the shell must be able to execute any OGSA or Web Service client, there must be a general purpose "wrapper" that implements the above interface and can contain a third-party client application.

11. Security Considerations

There are no security considerations associated with the contents of this document.

Author Information

Marlon Pierce
Community Grids Lab, Indiana University
501 N. Morton Street, Bloomington, IN 47404
Email: marpierc@indiana.edu

Geoffrey Fox
Community Grids Lab, Indiana University
501 N. Morton Street, Bloomington, IN 47404
Email: gcf@indiana.edu

Intellectual Property Statement

References

[BPEL4WS] "Business Process Execution Language for Web Services, Version 1.0," F. Cubera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Available from http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

[OGSA] S. Tuecke, et al. Grid Service Specification. http://www.gridforum.org/ogsi-wg/drafts/draft-ggf-ogsi-gridservice-04_2002-10-04.pdf.

[JXTA] L. Gong. Project JXTA: A Technology Overview. http://www.jxta.org/project/www/docs/jxtaview_01nov02.pdf.

[LEGION] A. Natrajan, M. A. Humphrey, and A. S. Grimshaw. "Grids: Harnessing Geographically-Separated Resources in a Multi-Organisational Context" Proceedings of *High Performance Computing Systems*, June 2001.

[UNIX] B. W. Kernighan and R. Pike. The Unix Programming Environment. Prentice Hall, Englewood Cliffs, NJ (1984).

[WSDL] "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.