PythonCLServiceTool: A Utility for Wrapping Command-Line Applications for The Grid

David E. Konerding and Keith R. Jackson

Distributed Systems Department

Lawrence Berkeley National Laboratory,

Berkeley, CA 94720

California, USA

[dekonerding,krjackson]@lbl.gov

WWW project page: http://dsd.lbl.gov/gtg/projects/PythonCLServiceTool/

Abstract.

The international science community has invested large amounts of money in developing numerical and computational codes for everything from basic math to application specific codes. These codes are now a vital part of the scientific process. However, running these codes can be challenging. Many require a highly specialized environment, and may only run in a few locations. To maximize the usage of these codes, it is necessary to enable network access to them. We discuss our recent work in developing automated tools to enable network access to command line applications using Grid[1] tools.

Please use the following format when citing this chapter:

Konerding, D. E., Jackson, K. R., 2007, in IFIP International Federation for Information Processing, Volume 239, Grid-Based Problem Solving Environments, eds. Gaffney, P. W., Pool, J.C.T., (Boston: Springer), pp. 195-211.

1. Introduction

The international science community has invested large amounts of money to develop numerical and computational codes for everything from basic math libraries to application specific codes. These codes are now a vital part of the scientific process. However, their usage often requires a specialized environment, and may only run in a small number of locations. To maximize the usage of these codes, it is necessary to enable remote network access to them. In this paper we will discuss our work to use Grid technologies to expose numerical/computational codes as Grid services. A Grid service is simply a network accessible service that uses standard protocols to describe its interface and for access. Typically today these protocols are based on industry standard Web Services[2].

We will begin the paper by looking at related work in the Web Service area. We will then look at the architecture of our system, PyCLST (Python Command Line Service Tool). After examining the overall architecture, we will look at a concrete example of PyCLST usage. Following a brief look at performance, we will discuss our future plans and conclusions.

2. Related Work

There are several existing software products which provide similar functionality to PyCLST. SOAP::Clean[3] (written in Perl) and O'SOAP[4] (written in O'CAML) served as the initial inspiration for PyCLST and provide a limited set of the functionality in PyCLST. The primary difference between PyCLST and SOAP::Clean/O'SOAP is that PyCLST is architected around Grid standards rather than plain SOAP[5]. This has a number of implications including:

1. Grid services enhance the security support in SOAP. SOAP lacks a standard authorization mechanism. Many Grids use a standard authorization mechanism based on a *gridmap* file. Grid security[6] also provides single-sign-on to reduce the number of times a user must enter their password. Another important feature Grid services offer is *delegation*. Delegation allows a user to grant some sub-set of her rights to a third-party. For example, after a computational job has run, output files might need to be moved to tertiary storage. While the user could manually move the files, it would be easier to have the computational service move them for her. Delegation allows the user to grant the service the right to interact with the tertiary storage system on her behalf.
2. SOAP has limited support for data transport. Binary data must be base-64 encoded which adds a 33% overhead, and large files must be broken into chunks for efficient transfer. Grid services provide efficient, high-performance, and secure data transport. We will use the standard GridFTP[7] protocol to transfer large input and output files between clients and services.
3. SOAP has no standard support for building stateful web services. Each SOAP service typically does this in an ad-hoc manner. Some use cookies,

others pass session identifiers with each SOAP message, but there is no standard way to manage state in Web Services. Grid services adopt the WS-Resource Framework[8] set of standards to provide state management. In particular, the Resource Context subset of WS-RF provides a unique "handle" shared between the service and client that refers to a particular request stream. The WS-Lifetime subset is used to manage the lifecycle of the service (create and destroy a resource context) and WS-ResourceProperty is used to manipulate meta-data associated with the service and provide a substrate for state change notification.
4. SOAP has no support for asynchronous notifications, so the command line client needs to periodically poll the server for result data or keep a request open for a long time. Grid services adopt the WS-Notification[9] to provide asynchronous notifications.

Each of these features is used by PythonCLServiceTool to provide enhanced functionality relative to SOAP::Clean and O'SOAP. Further, because client computers commonly have firewalls that prevent WS-Notification data being retrieved, we include code that detects and works around firewalls using a polling-based, rather than callback-based, response mechanism.

Kepler[10], an open source workflow execution tool built on the Ptolemy Framework[11], has a Web Service Harvester. The Harvester inspects WSDL[12] files and builds workflow actors which are capable of accessing the service defined by the WSDL file. This allows users to easily interface Kepler with existing web services without having to write code for new actors. Many other systems provide similar functionality, however, unlike PythonCLServiceTool, these tools are only used to generate client bindings to an existing service. Because PythonCLServiceTool exposes command-line applications using a standard web service interface and provides a WSDL file describing the service, tools like the Web Service Harvester can be used to generate their own clients to the wrapped service.

3. Architecture

3.1 Overview of general architecture

Developers configure PythonCLServiceTool clients and servers using a simple user-written configuration file containing named sections populated by key/value pairs. Because all PythonCLServiceTool clients and servers have a great deal of common functionality with just a small amount of command-specific generated code, a collection of template files are interpolated at client/server generation time. The command line parsing code is generated programmatically.

PythonCLServiceTool uses several existing Python frameworks rather than duplicating existing functionality. PythonCLServiceTool uses an asynchronous I/O framework called Twisted[13] for grid and web service communications, as well as external process management. Because the service can handle multiple simultaneous requests without delay, we needed an asynchronous I/O library. Without this functionality, the service would block on data send/receive operations and on process launching/management.

PythonCLServiceTool co-opts an existing standard Python framework used for packaging and distributing software called setuptools[14] for generating clients and servers. Setuptools provides an extensive plug-in framework normally intended for package installation, but we also use it for template substitution and service configuration and deployment. Optionally, the runtime and configuration files for the server can be packaged into a single executable file for easy server deployment. PythonCLServiceTool uses the setuptools *bdist_egg* and *bdist_wininst* features to build a self-contained installer containing the entire service or client runtimes.

The PythonCLServiceTool server generation process creates a Twisted server, the server configuration file, and the necessary runtime code for running the server. The PythonCLServiceTool client generation process creates a Twisted client, a client configuration file, and the necessary runtime code for running the client. Optionally the runtime and configuration files for the client can be packaged into a single executable file for easy client distribution. The client interface is intended to be exactly like the command line interface. It uses the same flags, and ideally should be a direct replacement for the command line application. However, because the client and server processes run in different file system contexts, and users will want to run jobs on files stored on the client computer, PyCLST adds a special syntax to the command-line shell that indicates that the referred-to file should be transferred to the server before job execution.

To ensure that only valid users may access deployed services, PyCLST supports the standard grid-map-file format which is used by the Globus Toolkit®[15] to authorize users based on the subject name contained in their X.509[16] certificate. Also, if the PyCLST server is run with super-user privileges, it will change identity to run the application as the local user specified in the map-file.

3.2 Configuration file format.

The configuration file is based on the Windows INI file format as interpreted by the Python ConfigParser module. A sample configuration file that wraps the *blast* command is seen in Figure 1. As you can see, it is a simple set of name value pairs separated out into different sections.

The user specifies the name of the executable to be run on the server side. The executable name should be specified as an absolute path to ensure that other executables earlier in the server container's PATH are not executed.

The developer specifies the name of the service, which is normally the same as the suffix of the executable pathname (the name of the binary). The service name is incorporated into the service in a number of ways: it is used to form a unique name for the server configuration files, the name of the generated SOAP parsing and encoding scripts, and the WSDL file.

The developer specifies all of the arguments that the executable is capable of accepting. There are two main types of arguments that command line applications are capable of taking:

- Positional arguments derive their meaning from their position in the command line. Developers indicate the position of an argument through the prefix name of the argument, for example, all position argument definitions starting with "arg1" refer to the argument in the first (left-most) position. The Unix *cat* command outputs the contents of files in left-to-right positional order on the command line. In this example, the configuration file specifies that the cat command can take up to three positional arguments (technically, the *cat* command can take many more but we left that out for brevity). All the positional arguments are specified as 'optional' because the cat command can be invoked without any positional arguments, in which case cat will read from standard input.

- Option arguments start with one or two dashes and may be followed by a value. "Option" in this context is taken from the Python *optparse* module, and does not mean that it is optional whether the argument is required. It merely indicates that this type of argument is typically used to indicate an optional different behavior. The developer indicates whether an option argument has a value associated with it and whether it is "optional". Option arguments with no value look like "-X" while arguments with a value look like "-X something". In this example, we specify two of cat's option arguments, -n and --version. The -n option to cat causes it to print line numbers at the beginning of each line. The "—version" option causes cat to output its version number.

There are a number of complex issues associated with argument parsing. Although the vast majority of applications follow simple conventions of allowing a mix of positional and option command line arguments, several perverse applications have much more complex rules. A comprehensive review of all applications (or even just standard Unix commands) is far beyond the scope of this document. Nevertheless, there are commands such as *tar*, *dd*, and *find*, each of which violates some of the conventions supported by PythonCLServiceTool. Tar allows multiple options to be coalesced into a single option (such as -xvf, which means extract verbosely from a file). PythonCLServiceTool has no configuration file syntax to support expressing when variables can be coalesced, and would interpret this as a single option called "xvf", which would not be recognized. The user can work around this restriction by specifying the options separately, "-x -v -f". dd is an odd Unix command which does not prepend its option arguments with dashes, so options look like "count=BLOCKS" instead of "-count BLOCKS". The developer can simply define these as positional arguments, losing some of the power of option arguments. The find command has a complex, stateful command line in which order matters. There is no support for checking for valid ordering at the client-side in PythonCLServiceTool; in this case, the find command run on the server will report an error. We reasoned that the *tar*, *dd*, and *find* behaviors are relatively rare and did not justify additional configuration file syntax or option parsing code. Ideally, all applications would conform to the GNU standard for command line options, and we would support exactly that standard, but our support covers the vast majority of cases without being unduly complex. Fortunately the scientific applications we are targeting typically follow the GNU standard anyway.

3.3 Grid Toolkit Support

PythonCLServiceTool uses the pyGridWare[17] toolkit to support all of its grid- and web-services functionality. The pyGridWare toolkit includes code to generate Python grid services from WSDL files, as well as runtime libraries for constructing and running grid services and clients. PyGridWare provides support for the WS-Resource Framework collection of standards used by grid services, as well as WS-Notification. When building a grid service, pyGridWare generates a text file that contains all the configuration details for the service (such as encryption, authorization, and authentication, as well as logging and other common service functionality), a script containing all the code to start the service on a deployment host, and the generated code for a specific grid service.

The code generated by PythonCLServiceTool follows standard WSRF practices in using a standard web service to manage multiple stateful instances. This ensures a deployed service can handle multiple simultaneous requests, each with its own “context” that ensures individual results are delivered to the correct user.

3.4 Template files and Code generation.

When the developer requests server or client code generation, a collection of template files which contain all the generally required common functionality are string interpolated using the configuration file substitution values for *name* and *executable*. The command line client stub, pyGridWare server container startup scripts, client and server runtime libraries, server configuration file, WSDL[12], and XML schema[18] corresponding to the interface of the service are generated. These WSDL and XML schema files can be re-used by other applications that can re-implement either the client or service sides of a PythonCLServiceTool instance. Users are not tied to using our Python clients. The usage of standard web service protocols means that new clients can be generated in any language that supports the WSRF/WS-N protocols suites. The output filenames from the template substitution are based on the input filenames, but are string interpolated to customize them to the service instance. This allows co-installation of the runtime code for several services in the same directory.

Some Python code specific to a PythonCLServiceTool instance is generated automatically (not from template files). The command-line parser and encoding and decoding routines are generated automatically from the argument definitions in the configuration file, because this generated code is highly dependent on the specific details of the command line application arguments.

PythonCLServiceTool utilizes nearly all the basic features of a WSRF based Grid client and service. It has been carefully documented to make it clear how basic Grid functionality (such as web service code generation, notifications, security, and deployment) can be used from Python. Developers are welcome to use and adapt the template file and code generation features of PythonCLServiceTool to develop their own grid applications.

3.5 Tool Usage and Deployment

PythonCLServiceTool leverages an existing Python project-building installation framework known as setuptools. It uses setuptools to string interpolate the template files, build deployable service instances, and build distributable clients. We chose to use setuptools because Python users are familiar with the standard setuptools commands; a simple

```
% python setup.py install
```

will generate the code, configure the service, and deploy it.

The PythonCLServiceTool generated service is typically deployed as a collection of files: the server configuration file, server startup script, and server runtime implementation code. The service configuration file is a simple text file with the same syntax as the PythonCLServiceTool configuration file used to generate the client and service. It defines logging, SSL security, authorization, the server port and interface, URLs to the grid services and the name of the executable. The startup script is either a shell script (on Unix-like platforms) or a DOS batch file (on Windows). The server runtime contains all the string-interpolated template files

PythonCLServiceTool uses the Twisted framework and pyGridWare toolkits to provide a web server that hosts the specific service instance. Multiple service instances (corresponding to different command-line applications) can be co-located into a single web server directory; this simplifies using a single machine to host multiple services (it is also possible to have multiple separate servers each using a different port).

When a client requests the service to run the command line, the service parses the SOAP-encoded command line, and uses the Twisted Framework to fork an external process that executes the command line. Twisted's internal asynchronous reactor support provides standard output, standard error and process return code to the service instance without blocking the service container. As bits of standard output and error become available from the process, the service instance pushes that data to the client via notifications. If firewall or other considerations disallow client notification support the client will detect it and poll the server.

The client side script is designed to be easily distributed to users. It uses the setuptools feature *bdist_rpm* to produce an RPM on RPM-based Linux systems. It can build self-contained executables for DOS- and Unix-like systems which include the client scripts, runtime libraries, and, configuration file. From the perspective of a user, the client script operates identically to the original command line (insofar as the configuration file is an accurate representation of the original command line's arguments).

3.6 File staging

Because users frequently have files stored locally that they need to process remotely, PythonCLServiceTool provides the ability to specify input files that are to be staged onto the server. PyCLST allows the user to specify local files, which are to be transferred to the server, through a special command-line syntax. The filename of the local file is enclosed in an '{' and '}'. For example:

```
% cat.sh {/tmp/localInputFile}
```

will transfer the file */tmp/localInputFile* from the client machine to a temporary directory on the server, and when the job runs on the service, it will have the temporary file name substituted into the command line.

Files are currently staged by base64 encoding by default but the staging mechanism could also be implemented using GridFTP, RFT, or other file movement mechanisms. This support is necessary for moving large data files. In the future PyCLST will optionally create a server working directory for each job instance and transfer all the files created in the directory during job execution back to the client.

4. Example Usage of PythonCLServiceTool

4.1 Wrapping the NCBI BLAST application *blastpgp*

We now demonstrate a concrete example of using PythonCLServiceTool: wrapping a very popular bioinformatics tool called BLAST[19]. Specifically, we will demonstrate wrapping the *blastpgp* command distributed with NCBI BLAST. *blastpgp* searches a protein sequence database against a protein query sequence, permitting gaps in the alignments between query and database sequences. Biologists use this application to identify novel genes based on their similarity to existing, well-characterized genes. BLAST is significantly faster than other sequence-search algorithms, although it is not as accurate as methods such as Hidden Markov Models.

In this example we will assume that the host on which the service is deployed already has the BLAST databases installed and properly formatted, while the query sequence is stored on the client machine.

A minimal blastpgp command line looks like the following:

```
% blastpgp –i query.fa –d database
```

The –i option lists a local filename which contains the query sequence, while the –d options lists the name of a database which is located using the BLASTDB variable. Because this option does not refer to an actual full path name but rather a collection of files in the BLASTDB directory which are prefixed by the database name, PythonCLServiceTool is not currently able to stage BLAST database files. This is generally not a problem because normally users would use a database preinstalled and formatted at the service. Blastpgp takes a number of options that affect its behavior, although except for –i and –d, all of these are optional. It takes no positional arguments.
The blastpgp wrapper config file is given in Figure 1. We have wrapped the two required options, -i and –d, and also wrapped several other commonly used options: -o, which allows the output to be redirect to a local (local to the service) file, -e which defines a different statistical cutoff threshold than the default, and –m which allows the output file format to be adjusted. Each of the options takes a value, and other than –i and –d are optional (not required). To create the service and client, enter

```
% python setup.py install.
```

In this example, running *python setup.py install* deploys both the service and the client to the local Python installation.

4.2 Running the Example

After the service and client have been deployed, start the service container. The name of the service container startup script on Unix-like systems is *start-container_blastpgp.sh* and on Windows is *start-container_blastpgp.bat*. The server can be started in any directory; that directory will act as the “current directory of the server”, meaning executable files will be run from that location and relative file path references will be computed from that location. Create a protein sequence query file in the current directory. Call it *test.fa* and have it contain the following query sequence:

```
>gi|33357914|pdb|1P85|M
MDKKSARIRRATRARRKLQELGATRLVVHRTPRHIYA
QVIAPNGSEVLVAASTVEKAIAEQLKYTGNKDAAAAV
GKAVAEALEKGIKDVSFDRSGFQYHGRVQALADAARE
AGLQF
```

Download the sample database pdbaa from
`ftp://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/pdbaa.gz`

and decompress it in this directory. Next run the blastpgp command

```
% formatdb –o T –I pdbaa
```

to create the BLAST index file.

Run the client command, which is called blastpgp.sh on Unix-like systems and blastpgp.bat on Windows, with the following options: *-d pdbaa –I test.fa*. If it is successful, *blastpgp* will print many lines of useful information including which sequences in the database match the query, the alignments of those sequences to the query, and statistical scores.

Finally, test the client with a local file. Move the test.fa file created in the service deployment directory to /tmp or some other folder, and run the following command (if your shell (most Unix shells do) has its own use for { and }, quote those characters with \):

```
% blastpgp.sh –d pdbaa –I {test.fa}.
```

5. Performance

A natural question arises when exposing numerical/computational codes as Grid services. Is the performance adequate? Despite the obvious benefits of exposing your codes over the network, if the performance is not sufficient it will not be used. Because PyCLST is based on the Python WSRF toolkit, pyGridWare, we will focus on examining the performance of pyGridWare. We will discuss the basic XML parsing performance and the additional overhead of various security options.

In pyGridWare the overwhelming majority of the overhead is in XML parsing. Messages must be serialized into an XML format before crossing the wire, and then de-serialized back into Python on the receiving end. Clearly the choice of an underlying XML parser is very important. We were able to gain a factor of 20 performance gain by switching to a C based XML parser with a Python interface from a pure Python XML parser.

We recently tested pyGridWare using a 2.0GHz dual opteron Linux machine. When running both client and server on the same machine to avoid network round trip time, we see that basic WSRF operations take a little less then 10ms per operation. Over 90% of this time is XML parsing. This is the baseline for what is possible, but in a real-world application security will become important.

While security is essential to running production level Grid services, it does add significant overhead. Using the same testing environment as before, we examined two different authentication mechanisms. The first is based on the widely used IETF standard TLS[20] protocol. This is the same protocol used on the WWW to interact with your bank, or place an order on Amazon. The second authentication mechanism uses XML security primitives to sign the SOAP messages being exchanged. In both cases, a Python binding to the open-source OpenSSL[21] toolkit provides the cryptographic primitives. Tests using the TLS protocol take approximately 25ms per operation. By using the support in the TLS protocol to re-use a security context, it is possible to amortize this overhead over a number of calls. Message level security based on XML security is significantly slower. Operations take approximately 65ms per round-trip.

While there are many applications where introducing a 10ms overhead would be unacceptable, we have found that for the applications we are targeting the performance of PyCLST is acceptable. For example, a typical BLAST query may run for 30 seconds. When you amortize the 25ms overhead over the entire 30 second run, the PyCLST overhead is negligible.

6. Future Work

6.1 Wrapping Command Line Applications for Grid Workflows

We anticipate that PythonCLServiceTool will be used to wrap command lines that are integrated into workflows. When multiple grid services are orchestrated together as a workflow, it is desirable for the user running the workflow to be able to delegate their credentials to services which carry out work on their behalf. This is an important feature for enabling an important optimization, which is that services will communicate in a third-party manner without sending the results of jobs back to the client, and also eliminates the need for users to type their passwords for each individual operation in the workflow. For example, BLAST output is frequently parsed by a secondary program to produce a compacted representation. It would be inefficient for a workflow to run an external BLAST program, collect the standard output, then forward it on to the standard input of the parsing program; instead, it is much more efficient for the standard output of the BLAST program to be connected directly to the standard input of the parsing program. This sort of third-party communication requires that the two services have the appropriate permissions, which is enabled through credential delegation. Credential delegation is also essential for access to other grid services such as WS-GRAM and RFT, support for which will be included in a future version of PythonCLServiceTool. These features will allow PythonCLServiceTool to add important functionality, including the ability to run jobs through an external scheduler, and reliably manage transfers of collections of large input/output files.

6.2 Authentication and Authorization

Currently, PythonCLServiceTool does not carry out any special checks to ensure that a client request is from an authorized user. This could lead to denial of service and other attacks on the service. PythonCLServiceTool will adopt an authorization model analogous to Globus based on the *gridmap* file. This model uses public-key cryptography combined with a file that maps user certificates into user names. When the client connects to the server, the server requires it to provide a certificate which states the identity of the user. The server validates the certificate, ensuring that it comes from a trusted source, and then uses the user name in the certificate to map to

a local user. If a client request is presented without a valid certificate, the server will immediately terminate the request. This file adopts the same format as the Globus gridmap file and performs effectively the same functionality. For example, this line in the gridmap file:
"/DC=org/DC=doegrids/OU=People/CN=David E. Konerding 692119" dek
indicates that a client who presents a certificate with the distinguished name in quotes will be mapped to the local user dek. Multiple certificates can be mapped to a single user on the system which is convenient if it is desired not to add any "extra" user accounts on the system.

In addition to supporting grid-map-file based authorization, we will also implement a standard authorization interface. This will allow others to plug in other more flexible means of authorization, i.e., SAML[22], VOMS[23], etc.

6.3 Advanced File and Job Support

Many applications output collections of files in the run directory of the service, and these files will need to be available to the client. Therefore, PythonCLServiceTool will be enhanced to support several higher-level data transfer mechanisms to facilitate high-performance, reliable file transfer as implemented by RFT and access to storage resource managers and brokers including SRM[24] and SRB[25].
The current PythonCLServiceTool model is to execute an application on the service host. However, in many situations the service host is not the most applicable location to execute the application. Therefore, PythonCLServiceTool will be enhanced to include submission to external batch schedulers using the standard WS-GRAM interface.

6.3 Fault tolerance

There are a number of aspects which could cause a job to fail while running on the service. Further, there are events which could cause the service container to crash. To ensure that the service container is reliable, and outstanding requests persist beyond a crash, the service will store its internal state in a durable disk file using a lightweight but powerful embedded RDBMS, “sqlite”[26].

Another aspect of maintaining a reliable service is to instrument the service with logging functions. This logging is invaluable when, inevitably, something goes wrong with the service. We will integrate support for the NetLogger[27] library, a lightweight but high-performance network logging toolkit designed for use in distributed systems like a Grid. NetLogger integration will enable service developers to get fine-grained views of their service’s operation, which is invaluable both during debugging and when diagnosing server failures.

6.4 Programmatic interfaces to legacy programs.

As we stated earlier, PythonCLServiceTool was initially targeted at wrapping command line applications, exposing the standard input, standard output/error, and return code of an application. However, some legacy applications expose their functionality at a library rather than application level. We will enhance PythonCLServiceTool so that it can be used to wrap libraries in addition to applications. These libraries will be accessed through a small command-line client stub which can be used to create service library instances, create instances of data structures defined by the library, invoke functions in the instance, and ultimately destroy the library instance.

Many Fortran applications can be automatically wrapped using the f2py[28] application. This application parses Fortran source code for a library and generates Python modules that can call the Fortran functions in the library directly.
C and C++ applications can be wrapped using SWIG[29]. SWIG parses C and C++ source code for a library and generates Python modules that can call the C/C++ functions in the library directly.

6.5 Config file format

We have identified a number of problems which cannot be addressed using our existing configuration file format. The format, while simple, is cumbersome for applications with large numbers of option arguments, unbounded number of positional parameters, and complex command line parameters that interact with each other. Future versions of PythonCLServiecTool will switch to an XML-based file format that will allow for much more extensive specification of command line behavior and will expose many more implementation details in PythonCLServiceTool to the developer. Since the existing format is useful for many simple applications, a translator from the existing format to the XML format will be provided.

The new XML file format will add support for more complex/varied command line formats (such as better support for /option formats used on Windows), including mechanisms for overriding the default file transfer behavior, redirection of I/O via third party interactions, interactions between command line options, and detection of invalid command lines on the client side.

6.6 Better self-contained deployment

We are investigating the use of py2exe[30] on Windows and freeze on UNIX to create a more self-contained deployment including the Python runtime. The current single-executable deployment contains only the service or client runtime libraries and startup script, thus it requires a valid Python installation on the target machine. We will use Py2exe and freeze to take the generated PythonCLServiceTool package, and combine it with a full Python installation and the required runtime libraries to provide a single executable that can be easily deployed to a service or client host.

7. Conclusion

PythonCLServiceTool addresses two existing problems facing the community: how to make applications available to scientists without distributing the entire software package, and how to make legacy application available on the grid without extensive retrofitting.

As shown in the Performance section, PythonCLServiceTool adds marginal overhead compared to typical long-running scientific applications. Although there is some cost to XML parsing and security, these represent only a tiny fraction of the overall time spent running the application.

There are still several remaining features which must be implemented before PythonCLServiceTool can be used in production environments. The authorization functionality needs to be implemented so that only authorized users can invoke the service. Command-line syntax and back-end support for high-performance file transfers is required before large files and standard input/output can be used. Fault-tolerance, recovery and logging must be implemented for the service to be useful in long-running production environments. Finally, the configuration file syntax needs to be significantly enhanced to support these features and to allow for more sophisticated command line support. Nevertheless, PyCLST has shown the value of simple automated tools to help expose legacy applications as Grid services.

8. Figures

Figure 1

```
[main]
name=blastpgp
executable=/home/portnoy/u5/dek/sw/i386/blast-2.2.13/bin/blastpgp

[optionarguments]

arg1option=-i
arg1desc= Query [File In]
arg1hasvalue=True
arg1optional=False

arg2option=-d
arg2desc= Database [String]
arg2hasvalue=True
arg2optional=False

arg3option=-o
arg3desc= Output File For Alignment [File out]
arg3hasvalue=True
arg3optional=True

arg4options=-e
arg4desc=Expectation value (E) [Real]
arg4hasvalue=True
arg4optional=True


arg5option=-m
arg5desc=alignment view options
arg5hasvalue=True
arg5optional=True

## BLAST does not have any position arguments.
[positionarguments]
```

9. Bibliography

1. Foster, I., C. Kesselman, and S. Tuecke, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations.* Intl. J. Supercomputer Applications, 2001.
2. Kreger, H., *Web Services Conceptual Architecture*. 2001, IBM.
3. Cornell.edu. *SOAP::Clean*. 2006 [cited; Available from: http://www.asp.cornell.edu/SOAP-Clean/.
4. Cornell.edu. *O'SOAP*. 2006 [cited; Available from: http://www.asp.cornell.edu/osoap/.
5. *Simple Object Access Protocol (SOAP) 1.1*. 2000, W3C.
6. Globus Project, *Grid Security Infrastructure (GSI)*. 2002.
7. Globus Project, *The GridFTP Protocol and Software*. 2002.
8. *Web Services Resource Framework (WSRF) - Primer v1.2*. 2006, OASIS.
9. *Web Services Base Notification 1.3 (WS-BaseNotification)*. 2006, OASIS.
10. Ludascher, B., et al., *Scientific Workflow Management and the Kepler System.* Concurrency and Computation: Practice & Experience, 2005(Special Issue on Scientific Workflows).
11. *Ptolemy*. 2006 [cited; Available from: http://ptolemy.eecs.berkeley.edu/ptolemyII/.
12. Christensen, E., et al., *Web Services Description Language (WSDL) 1.1*. 2001.
13. *Twisted*. 2006 [cited; Available from: http://twistedmatrix.com/trac/.
14. *Setuptools*. 2006 [cited; Available from: http://peak.telecommunity.com/DevCenter/setuptools.
15. Foster, I., C. Kesselman, and S. Tuecke, *The Globus Toolkit and Grid Architecture*. 2001, In preparation.
16. Adams, C. and S. Farrell, *Internet X.509 Public Key Infrastructure Certificate.* Mar 1999(2510).
17. Jackson, K.R. *pyGridWare*. 2006 [cited; Available from: http://dsd.lbl.gov/gtg/projects/pyGridWare/.
18. Fallside, D.C., *XML Schema Part 0: Primer*. 2001, W3C.
19. *BLAST*. 2006 [cited; Available from: http://www.ncbi.nlm.nih.gov/BLAST/.
20. Dierks, T. and C. Allen, *The TLS Protocol Version 1.0*. 1999, IETF.
21. *OpenSSL*. 2002 [cited; Available from: http://www.openssl.org/.
22. *Security Association Markup Language (SAML) Specification v.1.0*. 2002, OASIS.
23. EU DataGrid, *VOMS Architecture v1.1*. 2003.
24. Gu, J., A. Sim, and A. Shoshani, *The Storage Resource Manager Interface Specification, version 2.1*. 2003.
25. Baru, C., et al. *The SDSC Storage Resource Broker*. in *8th Annual IBM Centers for Advanced Studies Conference*. 1998. Toronto, Canada.
26. *Sqlite*. 2006 [cited; Available from: http://www.sqlite.org/.
27. Gunter, D., et al. *NetLogger: A Toolkit for Distributed System Performance Analysis*. In *IEEE Mascots 2000: Eighth International Symposium on*

Modeling, Analysis and Simulation of Computer and Telecommunication Systems. 2000.

28. *F2py*. 2006.
29. Beazley, D.M. *SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++*. in *Proceedings of the 4th USENIX Tcl/Tk Workshop*. 1996.
30. *Py2exe*. 2006 [cited; Available from: http://www.py2exe.org/.

Q&A – Keith Jackson

Questioner: Brian Smith

Numerical applications often are made up of multiple numeric components. These components frequently pass large amounts of data between components and other applications, the data is returned as part of the service. Are the service tools and techniques able to support such scenarios in a reasonable way?

Keith Jackson

Currently tools are available to support these scenarios, but only in an ad-hoc manner. It is an ongoing area of research to understand what functionality and interfaces services should support to make linking them together into larger applications easier. Today too much of this process must be done manually as part of the workflow. For example, today I would submit a job to a numerical service, and then have my workflow script use GridFTP to move the data to the next service. In the future, it should be possible to tell the first service that it should send its output data to the second service and have it all happen without manually coding the data transfers.

Questioner: Craig Douglas

Where are examples of automated tools in Python that actually generate real, non-trivial services?

Keith Jackson

Currently *SWIG* is the most popular tool to generate Python bindings to C or C++ code. *F2py* is probably the most popular tool for generating Python bindings to Fortran codes. Both have been used extensively to provide interfaces to numerical codes. A good example is the data analysis pipeline for the Hubble telescope. All of the images run through a pipeline that is written in Python, but invokes a large number of services written in C, C++, and Fortran. For more information on wrapping numerical codes in Python, see: http://www.scipy.org/Cookbook.

Comment: Craig Douglas

Visual C 1.0 (circa 1998) compiled and linked 90,000 lines of code to put a window up that said "Hello". Automated tools that generate an enormous number of lines of Grid/Globus code similarly emphasize that Grid services programming is obscenely over-complicated.

Keith Jackson

Indeed Grid service programming can be very complicated. Finding ways to hide as much of that complexity as possible while still providing useful tools to the scientists is an ongoing research problem.

Questioner: Gabrielle Allen

Can you explain more about how services are deployed. Is there one service for each application on the machine? How do you track changes in the application location?

Keith Jackson

Currently there is one factory service for each application on the machine. Each running instance of that service would also have a unique service that encapsulates the running code. Service deployment is still done manually. We plan on leveraging some of the work of Kate Keahey at ANL on automated service deployment. She has been investigating what information you need, and how it should be presented, so that a system can decide where to deploy services in real time.

Comment: Dennis Gannon

We use an application service factory to generate service instances on the fly. This solves the problem of "too many services".

Questioner: Dennis Gannon

How do you handle programs with complex input and output files?

Keith Jackson

Currently the user has to ensure that the proper files are in place before running the service, and then move the output files after the service has completed. Typically this is done using GridFTP. In our visual workflow tool we encapsulate this into a hyper-graph node that uses GridFTP to move the input files in, runs the service, and moves the output files. To the scientist this looks like an atomic operation.