pyGlobus: A Python Interface to the Globus Toolkit™

Keith R. Jackson
Lawrence Berkeley National Laboratory
1 Cyclotron Road
MS: 50B-2239
Phone: 510-486-4401
Fax: 510-486-6363
Email: KRJackson@lbl.gov

This work is supported by the U.S. Dept. of Energy, Office of Science, Office of Advanced Scientific Computing Research, Mathematical, Information, and Computational Sciences Division under contract DE-AC03-76SF00098 with the University of California.

Abstract

Developing high-performance problem solving environments/applications that allow scientists to easily harness the power of the emerging national-scale "Grid" infrastructure is currently a difficult task. Although many of the necessary low-level services, e.g. security, resource discovery, remote access to compute/data resource, etc., are available, it can be a challenge to rapidly integrate them into a new application.

To address this difficulty we have begun the development of a Python based high-level interface to the Grid services provided by the Globus Toolkit [1]. In this paper we will explain why rapid application development using Grid services is important, look briefly at a motivating example, and finally look at the design and implementation of the pyGlobus package.

Introduction

The emergence of large-scale "Computation/Data Grids"[2] offers the promise of dynamically constructing domain specific problem solving environments [3] to support high-end science. Many science projects today, e.g., high-energy physics and observational cosmology, require the coordinated use of organizationally and geographically distributed simulation codes, data archives, instruments, and research teams. While the community has made great progress in providing the basic Grid services, it can be challenging for non-computer scientists to access them. This difficulty impedes the goal of allowing application scientists to rapidly develop applications that utilize Grid services.

To address this problem, we have begun the development of higher-level object-oriented toolkits that support rapid application development through the use of modern software engineering techniques. In addition to the Python based toolkit that is our focus, there are also related projects to provide similar toolkits in Java [4], and CORBA [5]. We believe that Python has several properties that make it well suited for Grid programming.

- It's a high-level object-oriented programming language with; automatic memory management, dynamic typing and binding, simple, easy to learn syntax, and support for packages, modules and classes.
- It has a wide variety of built in data structures including, lists, hash tables, and through the NumericPython package, high-performance multi-dimensional arrays.
- There exist a wide variety of modules to perform various tasks including support for XML [6] processing, SOAP [7], etc.
- It's portable across any platform that supports ANSI-C. The Python interpreter is written in ANSI-C and compiles under all flavors of UNIX, Win32, and MacOS.
- Python code is easy to develop and maintain due to the simplicity of the syntax.
- It is easy to integrate native C/C++ or Fortran code into Python as an extension module.
- Excellent performance. Through the use of extension modules for key areas, it is possible to achieve performance within one or two percent of optimized C code.
- Python is currently extensively used in the high-performance computing world; so many application scientists are already familiar with it.
- Platform independent GUI toolkits and Open Source IDE's are available.
- Support for meta-programming. Python offers built-in support for introspection, which allows for automatic discovery of interfaces by applications and GUI builder tools.
- The Python language is Open-Source.

In the rest of this paper, we discuss a motivating example that illustrates some of the problems in Grid computing, and then present a technical overview of the Python CoG Kit.

Motivation

We are currently in the midst of a fundamental shift in the way science is done due to the growing ability to dynamically couple heterogeneous compute, data, instrument, and collaboration resources. NASA's

Information Power Grid (IPG) and the NCSA Alliance's National Technology Grid have demonstrated the feasibility of providing persistent Grid Services to the scientific community. One similarity amongst these Grids is the use of the Globus toolkit to provide many of the underlying Grid Services. The Globus toolkit provides a number of modules that implement Grid Services for security, resource discovery, data transfer/management, etc. It has become the most popular solution to providing these services.

Although many of the necessary services are becoming available, they can still be very challenging to use. To fully realize the goal of allowing application scientists to routinely use Grid Services, more must be done to ease the burden of Grid application development. The careful use of appropriate abstractions and higher-level constructs such as objects and components can help hide much of the complexity of Grid programming from the application scientist.

We now consider a motivating example from the field of observation cosmology that illustrates many of the requirements we see for 21st century science.

Recent studies of distant supernova have shown that the expansion of the universe is accelerating under the influence of a new force, called *dark energy*. Current studies are conducted by geographically distributed research teams, and involve the coordinated use of several ground-based observatories, the Hubble space telescope and multiple distributed compute and storage resources. For example, the Supernova Factory at LBNL utilizes instruments in Hawaii and California, and storage and compute resources at Cal Tech and LBNL. As the program progresses, it will incorporate resources in Chile and the Canary Islands. This program is a stepping-stone to the next generation search, the space-based Supernova Acceleration Probe (SNAP).

As the scale of these searches has increased, a number of new requirements have emerged. The first is the shear scale of the data handling and compute tasks involved. Raw, uncorrected sky images must be transferred nightly from the remote observatories to compute facilities. The images are then corrected and calibrated to remove any atmospheric effects or tracking errors. The results are then compared to baseline sky catalogs to eliminate asteroids and man-made satellites. Finally algorithms are applied to the images to search for increases in stellar magnitude that may indicate a supernova event. The resulting data is then analyzed manually be researchers to find the most promising candidates to observe. This process involves approximately 50 gigabytes of data in 500 files to be transferred, processed, and archived daily for the life of the project - 5 to 10 years.

Secondly, as the accuracy of supernova models increases, it should be possible to allow the tight integration of simulation data with experimental data to help filter out candidate supernovas. As more accurate supernova simulations are developed over the next year, it should become possible to use these to filter out candidate supernovas for further observation. This process of comparing simulation with experiment must happen within a 24 hour time period to be of use in filtering out candidate supernovas.

To create large scale science applications such as those described above requires the close coupling of a variety of technologies. It is necessary to move and access large amounts of data, access high-performance compute resources, and integrate one-of-a-kind scientific instruments. Many of these functions are provided by a combination of open-source and proprietary software that must be used as an integrated whole. The Python language offers excellent support for this type of "glue" coding.

pyGlobus Overview

The rest of the paper will focus on the Python CoG Kit, pyGlobus, and explain; what the high-level goals for the project are, and how Globus concepts are mapped to the Python idiom. It will also discuss some of the underlying implementation details before examining a number of the most commonly used interfaces.

This project began with a number of important high-level goals in mind. First, we wanted to ensure performance levels at or near the native Globus C code. To do this we have relied on the use of native extension modules in Python. This allows Python code to cleanly interface with the underlying C code. By using Python solely as a very thin control proxy, we can minimize the performance cost associated with the wrapping. Second, where possible we have mapped the underlying C code to a natural Python idiom. For example, in C it is normal to return an *int* status code and use pointers to pass in other output variables. In Python functions may return multiple values. The wrapper functions take care of mapping between these two styles. Another important example is the use of exceptions. Python provides support for catching and throwing exceptions to indicate error conditions. The pyGlobus wrappers convert the underlying Globus error codes into Python exceptions, allowing for much cleaner error handling at the Python level. The third goal was to minimize the complexity of Grid programming as much as possible by the careful use of object-oriented programming techniques such as abstraction, encapsulation, and polymorphism. We have taken the approach of using abstraction and default arguments to provide a simple clean interface for most users, while still providing access to a more rich set of capabilities for the advanced user.

While it is possible to generate wrapper functions by hand to interface C and Python code, in practice this is a very mechanical and time consuming process. A number of tools exist to help automate this process. We have chosen to use the Simple Wrapper Interface Generator (SWIG) [8] to generate our interfaces. SWIG supports the mapping of built in and user defined C types into Python types, including the ability to override the default type mappings. Although the use of SWIG does not eliminate the need to write wrapper code, it does minimize this.

We have found it useful to distinguish between two categories of code in pyGlobus. The first provides a low-level mapping between Globus functions and Python methods. Although the Globus toolkit is written in C, it is still an object-oriented architecture. Hence it was possible to do a fairly direct mapping into Python proxy classes. For example, the Globus ftp client module provides a number of functions that take a *globus_ftp_client_handleattr_t* pointer as their first argument. In pyGlobus, we have an *ftpClient.HandleAttr* object that acts as a proxy for all of these functions. In addition to these proxy classes, we intend to build a set of higher-level components that build upon, and extend the basic functionality provided by the Globus toolkit. For example, we are working with the supernova group to develop a set of components to help manage the shepherding of 500 files a night from three different locations. This will use the underling GridFTP [9] protocol to transfer the data, but will add support for automated performance tuning, logging, and fault recovery.

Package Overview

In this section we will look at a number of the major modules that provide the basic interface to the Globus toolkit. Although we will look at several code examples, this is not intended as a complete introduction to the pyGlobus package. Instead we hope to provide an overview of the general functionality provided by pyGlobus. For further information consult the online API documentation (http://www.doesciencegrid.org/projects/pyGlobus/api_doc/).

Exceptions

The pyGlobus package makes extensive use of exceptions for error handling. It provides a base class for all package exceptions, *pyGlobus.util.GlobusException*, which inherits from the built-in exception base class, *exceptions.Exception*. Each of the other modules in pyGlobus defines its own sub-classes of *GlobusException*, e.g., the gramClient module defines a *GramClientException* that extends from *GlobusException*. This provides for a great deal of flexibility in error handling.

Resource Acquisition

The *gramClient* module provides the main interface to the Globus GRAM [10] protocol and provides resource acquisition and management functionality. It supports the ability to remotely start and manage compute jobs through a uniform interface. The *GramClient* class provides methods to submit, check status of, and cancel jobs. The following example illustrates using the *GramClient* class to submit a simple job.

```
from threading import *
from pyGlobus import gramClient
# Callback function for job state changes
cond = 0
def func(cv, contact, state, error):
    global cond
    ... # handle various job states
    elif state == gramClient.JOB_STATE_DONE:
        print "Job is done"
        cv.acquire()
        cond = 1
        cv.notify()
        cv.release()

condV = Condition(Lock())
try:
    # Construct object, init's globus modules and creates the
    # underlying handle
    gramClnt = GramClient()
    # Set the callback to receive state changes.
    callbackContact = gramClnt.set_callback(func, condV)
    # Submit the request. rm is the Resource Manager to
    # contact. rsl is the RSL describing the job request.
    jobContact = gramClnt.submit_request(rm, rsl,
                 gramClient.JOB_STATE_ALL,callbackContact)
# Now handle any exceptions and wait on the condition
         variable.
```
The Globus toolkit uses callbacks to propagate information back to the application; pyGlobus follows the same model, but allows the callbacks to be written in Python.

Secure IO

pyGlobus provides an easy to use interface to high-performance secure synchronous and asynchronous remote IO using the Grid Security Infrastructure (GSI) [11] to support PKI authentication. The GSITCPSocket class provides the main interface to the remote IO facilities. The io module also contains a series of attribute objects that allow the user to control a variety of settings, including tcp buffer size, authentication and authorization modes, out-of-band data handling, etc. A simple example will illustrate how easy it is to create a secure authenticated server in Python.

```
from pyGlobus.io import GSITCPSocket

    try:
        # Construct a socket object
        soc = GSITCPSocket()
        # Create a listener and return the port
        port = soc.create_listener()
        soc.listen()
        # Accept incoming connections
        childSoc = soc.accept()
        # Retrieve the delegated credential
        cred = childSoc.get_delegated_credential()
        str = "spam, spam, eggs, and spam"
        # Write the sting
        nBytes = childSoc.write(str, len(str))
    # Catch any exceptions here
```

A similar example shows a GSI enabled client.

```
from pyGlobus.io import GSITCPSocket

    try:
        # Construct a socket object
        soc = GSITCPSocket()
        # Connect to the remove server
        soc.connect(host, port)
        # Read in the data
        soc.read(buf, size, size)
    # Catch any exceptions
```

These examples illustrate the simplest use of the GSITCPSocket class. Through the use of attribute classes it is possible to tune network parameters for the connection, set network options and security attributes.

Grid FTP

Access to the GridFTP protocol is provided through two modules. The ftpClient module provides access to the client side functionality, including the ability to set parameters for the underlying tcp connection, get and put files, make and delete directories, list files, control security parameters, third party transfers, partial transfers, parallelism, set plugins, etc. The ftpControl module provides a lower level interface useful for implementing servers. An example will illustrate a simple use of the ftpClient module to initiate a third party gridftp transfer.

```
from pyGlobus import ftpClient

def done_func(cv, handle, error):
    if not error == “NONE”:
        print “The following error occurred: %s” % error
    …

ftpClnt = ftpClient.FtpClient()
ftpClnt.third_party_transfer(srcURL, destURL, done_func, cv)
# Catch exceptions and wait for the callback to be called
```

Gass Copy

The gassCopy module provides a protocol independent interface to transferring files. It supports the ftp, gridftp, http, and https protocols in addition to local files. It also provides access to a number of configurable attributes to control various performance options, and supports both synchronous and asynchronous transfers. The GassCopy class provides the main interface for file transfer, and supports methods such as *copy_utl_to_url* and *register_copy_handle_to_url*. The following code will transfer a file from a web server to a gridftp server.

```
from pyGlobus import gassCopy
from pyGlobus.util import GlobusException

try:
    gCopy = gassCopy.GassCopy()
    gCopy.copy_url_to_url(“http://foo.com/tmp”,
                               “gsiftp://bar.com/tmp/a”)
except GlobusException, ex:
    print ex.str
```

Gass File

The gassFile module provides the ability to manipulate remote files as if they were local. It supports files that can be accessed through http, https, ftp, and gridftp. Either an integer file descriptor or Python file object may be returned. The following example opens a remote file as a Python file object and reads all of the lines from it.

```
from pyGlobus import gassFile

gFile = gassFile.GassFile()
f = gFile.fopen(“http://www.foo.com/index.html”, “r”)
lines = f.readlines()
gFile.fclose(f)
```

GSI SOAP

The GSISOAP module provides support for the SOAP protocol bound to HTTP running over a GSI authenticated connection. This allows authentication, data integrity/privacy, and the ability to delegate GSI credentials. The module supports creating both SOAP clients and servers. This module builds on top of the SOAP.py [12] open-source SOAP implementation. A simple GSI SOAP server that echoes requests is illustrated below.

```
from pyGlobus import GSISOAP
from pyGlobus import ioc

def echo(str, _SOAPContext):
    c = _SOAPContext
    # Get the delegated credential
    cred = c.delegated_cred
    # Export the delegated cred to a file
    credPath = cred.export_external()
    # Return the original sting to the caller
    return str

# Creates an instance of a GSI SOAP server
# listening on port 8088
server = GSISOAP.SOAPServer("foobar.lbl.gov", 8088)
# Request delegation of a full GSI proxy
server.delegation_mode =
                 ioc.GLOBUS_IO_SECURE_DELEGATION_MODE_FULL_PROXY
# Register the "echo" function with the SOAP server
server.registerFunction(GSISOAP.MethodSig(echo, keywords=0, context=1),
                                                        "urn:gtg-Echo")
server.serve_forever()
```

A simple client to access this server is now shown.

```
from pyGlobus import GSISOAP
from pyGlobus import ioc

# Create the SOAP proxy class for the remote method
proxy = GSISOAP.SOAPProxy("https://foobar.lbl.gov:8088",
                                        namespace="urn:gtg-Echo")
# Delegate a full GSI Proxy to the server
proxy.delegation_mode =
                 ioc.GLOBUS_IO_SECURE_DELEGATION_MODE_FULL_PROXY
# Call the remote server and print the result
print proxy.echo("No one expects the Spanish Inquisition!!")
```

Future Directions

Although most of the Globus toolkit has been wrapped, there are a number of areas still to be completed. In particular, the gass transfer and ftp control packages are still only partially implemented. Once these are completed, we will begin to develop a number of higher-level components for developing portals with the WebKit [13] servlet engine or Zope [14], and developing domain specific problem solving environments. We will also be looking at developing a set of common GUI components, using wxPython [15], to support file transfer, job control, etc.

Summary

The Python CoG Kit offers the ability to rapidly develop applications that access Grid Services provided by the Globus toolkit. It provides a simple high-level object-oriented interface, yet offers good performance. pyGlobus currently supports most of the functionality of Globus, and future work will complete the remaining modules. Although more feedback from users is necessary to create more high-level components, pyGlobus has already proven useful to a number of projects.

Acknowledgements

We are grateful to a number of individuals who have contributed to the development of pyGlobus. In particular we'd like to thank: Joshua Boverhof, Dennis Gannon, Dan Gunter, William Johnston, Gregor von Laszewski, Jason Lee, Jason Novotny, and Mary Thompson. This work is supported by the U.S. Dept. of Energy, Office of Science, Office of Advanced Scientific Computing Research, Mathematical, Information, and Computational Sciences Division under contract DE-AC03-76SF00098 with the University of California. Its report number is: LNBL-49370.

References

1. Foster I., Kesselman C., Tuecke S. 2001. The Anatomy of the Grid: Enabling Scalable Organizations. Intl. J. Supercomputer Applications; 15(3) 2001.
2. Foster I., Kesselman C., editors. The Grid: Blueprint for a Future Computing Infrastructure. Morgan-Kaufmann, 1999.
3. Allen G., Benger W., Goodale T., Hege, H., Lanfermann G., Merzky A., Radke T., Seidel E. 2000. The Cactus Code: A Problem Solving Environment for the Grid. Proc. 9th IEEE International Symposium on High Performance Distributed Computing; 253-260.
4. Laszewski G., Foster I., Gawor J., Lane P. 2001. A Java Commodity Grid Kit. Concurrency and Computation: Practice and Experience 2001; 13(8-9):643-662.
5. Verma S., Gawor J., Laszewski G., Parashar M. 2001. A CORBA Commodity Grid Kit. 2nd International Workshop on Grid Computing. November 2001.
6. XML. http://www.w3.org/XML/. January 2002.
7. SOAP. http://www.w3.org/TR/SOAP/. January 2002.
8. SWIG. http://www.swig.org/. January 2002.
9. Allcock B., Bester J., Bresnahan J., Chervenak A., Foster I., Kesselman C., Meder S., Nefedova V., Quesnel D., Tuecke S. 2001. Data Management and Transfer in High-Performance Computational Grid Environments. Parallel Computing, 2001.
10. Czajkowski K., Foster I., Karonis N., Kesselman C., Martin S., Smith W., Tuecke S. 1998. A Resource Management Architecture for Metacomputing Systems. Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing, 1998.
11. Foster I., Kesselman C., Tsudik G., Tuecke S. 1998. A Security Architecture for Computational Grids. Proc. 5th ACM Conference on Computer and Communications Security Conference; 83-92.
12. SOAP.py. http://sourceforge.net/projects/pywebsvcs/. January 2002.
13. WebKit. http://webware.sourceforge.net/. January 2002.
14. Zope. http://www.zope.org/. January 2002.
15. wxPython. http://www.wxpython.org/. January 2002.