SDSC TR-2006-5

Opal: Simple Web Services Wrappers for Scientific Applications

Sriram Krishnan[*†], Brent Stearn[†], Karan Bhatia[†], Kim K. Baldridge[*†‡], Wilfred Li[*†] and Peter Arzberger[*]
[*]National Biomedical Computation Resource
[†]San Diego Supercomputer Center
UC San Diego MC 0505, 9500 Gilman Dr, La Jolla, CA 92093
[‡]Institute of Organic Chemistry, University of Zurich
Winterthurerstrasse 190, CH-8057 Zurich, Switzerland
{sriram, flujul, karan, kimb, wilfred, parzberg}@sdsc.edu

Abstract— **The Grid-based computational infrastructure enables large-scale scientific applications to be run on distributed resources and coupled in innovative ways. However, in practice, Grid-based resources are not very easy to use for the end-user. The end-user has to learn how to generate security credentials, stage inputs and outputs, access Grid-based schedulers, and install complex client software to do so. This has proved to be an effective deterrent for a number of scientific users. There is an imminent need to provide transparent access to these resources so that the end-users are shielded from the complicated details, and free to concentrate on their domain science. Scientific applications wrapped as Web services alleviate some of these problems by hiding the complexities of the back-end security and computational infrastructure, only exposing a simple SOAP API that can be accessed programmatically by application-specific user interfaces. However, writing the application services that access Grid resources can itself be complicated, especially if it has to be replicated for every application. In this paper, we will present Opal, which is a toolkit for wrapping scientific applications as Web services in a matter of hours. Opal provides features such as scheduling, standards-based Grid security, and data management in an easy-to-use and configurable manner. We will present some of the scientific applications that have been deployed using Opal, and describe the steps involved in doing so. Furthermore, we will demonstrate access to the Opal-based scientific applications via a number of different clients. With the help of Kepler, a scientific workflow tool, we will also show how Opal-based Web services can be used to orchestrate complex scientific pipelines.**

I. Introduction

Modern computational infrastructures for scientific computing are comprised of thousands to tens of thousands of individual processors clustered together with low-latency networks with a total capability of a few hundred teraflops [12]. These clusters are themselves aggregated into an infrastructure that connects large computational clusters, online and offline data storage, and visualization resources. The NSF funded Teragrid [10], for example, consists of nine sites connected by a 40Gbs networking backbone with an aggregate compute capability of over 20 TeraFlops, and over 1 PetaByte of online disk storage. Modern scientific methods require access to resources of this scale, while the economics of these infrastructures favor the distributing of the raw resources across multiple sites with collaborations that span across the sites and across particular specialties. Such collaborations are called *Grids* [28], *Virtual organizations* [29], or *Cyberinfrastructures*.

Programming and using these distributed infrastructures, however, is quite a challenge with multiple hardware configurations, software configurations, and administrative policies that affect different parts of the distributed system. There is hope that *Service-Oriented Architectures* (SOA) can address this problem by providing platform-independent, language-neutral service interfaces that hide the complexity of the implementations while providing a well-defined and high-performance Quality of Service (QoS). From an end-user's perspective, the services that are most relevant are services that perform a scientific operation and where the semantics of the operations are defined in terms of the domain science. For example, a *Blast* [19] service can provide biologists with the capability to compare multiple DNA sequences against each other or against standard publicly available datasets. From the end-user's perspective, the particular infrastructure used for its calculation is irrelevant. The service implementation may use sophisticated scheduling techniques that maximize job throughput, minimize data movement, or optimize in a host of other ways. Service-oriented architectures also support generalized workflows, hierarchical composition and, when used with strong types, support type checking and easy-to-use data translations.

The Grid Computing community has been moving towards services and service interfaces for some time: the Globus toolkit [32] began to refactor its capabilities into the *Open Grid Service Infrastructure (OGSI)* and standard higher-level architecture developed as part of the *Open Grid Service Architecture (OGSA)* [29] through organizations such as the Global Grid Forum. This evolution continues with recent Globus versions 4.x which conforms to the *Web Service Resource Framework (WSRF)* [25] that better aligns itself with emerging Web services specifications, such as WS-Addressing [24]. Moving to this new framework brings scientific computing in line technologically with current commercial and enterprise computing, and has benefited through leveraging of available commercial tools and standards. But this has also required significant time and effort to rewrite much of the functionality the of Grid infrastructure tools.

From an application standpoint, rewriting the functionality of the tools is not possible - the end-users have come to trust the calculations of these applications only over many decades of development and refinement. The significant challenge is to bring these legacy applications into the service framework, thereby leveraging all the advantages of such a framework, without requiring changes to the applications itself and without requiring significant effort. This is the problem that the Opal toolkit addresses.

Opal is one among a suite of tools being developed by the National Biomedical Computation Resource (NBCR) [7] at the University of California, San Diego. NBCR is building a Services Oriented Architecture for *Grid-enabling* biomedical codes and providing access to distributed biological and biomedical databases. This will allow biomedical researchers to harness the computational power of the Grid, and securely access very large data resources and specialized instruments available on the distributed resources. Opal is a toolkit that automatically wraps legacy applications with a Web services layer that is fully integrated with Grid Security Infrastructure (GSI) based security [30], cluster support, and data management. This paper describes the architecture and implementation of the Opal toolkit, and how it has been used for Grid-enabling scientific applications for the NBCR community.

The rest of the paper is organized as follows. Section II describes the related work, their shortcomings, and our motivation for developing Opal. Section III describes the overall end-to-end architecture of the NBCR services. Section IV presents the technical details of the Opal implementation. Section V describes a few use-cases where Opal has been used to Grid-enable scientific applications, and compose workflows from them. Section VI describes areas of future research, while Section VII presents our conclusions.

II. Motivation

There are several toolkits that attempt to abstract out the interactions with the complex back-end cyberinfrastructure. Two concrete examples are the Globus toolkit, and the GridLAB project [2]. The Globus toolkit provides the Grid Resource Allocation Manager (GRAM) [23] for submitting jobs to the Grid in a scheduler-agnostic way. The GridLAB project uses the Grid Application Toolkit (GAT) to provide uniform access to Grid middleware via a standard set of APIs that can be used by clients.

However, neither of the above projects treat scientific applications as first class Web services. In other words, clients interact with the Grid resources and their client libraries to submit specific scientific jobs. They do not interact with the applications directly. We believe that wrapping a scientific application as a Web service is a better approach for the following reasons:

- Some of the scientific applications are quite complicated to install and deploy. If they are made available as Web services to authorized users via a simple SOAP API, it obviates the need for every user to install the same application. Deployment of these applications can be delegated to the experts (possibly, the developers of the scientific codes), while it can be easily used by the entire community.
- User accounts do not have to be created for every user who wishes to access the application - the services can ensure that multiple users can be supported concurrently by performing every run in a separate working directory. Authentication and authorization can be performed using standard GSI-based mechanisms.
- Data management for all runs can be performed by the services themselves - otherwise, it is typically a user's responsibility to make sure that all inputs are at the correct locations, and that outputs are not overwritten during subsequent runs. This can be quite tricky to perform, especially if it has to be done for every application that is being used, and if these applications are being used repeatedly.
- Users do not have to be concerned with the Grid schedulers bring used at the back-end. The services can be configured to leverage appropriate ones, completely shielding the users from the complicated details.
- Since the applications can be accessed via a SOAP API, clients are not limited to traditional shell and Portal-based interfaces. Interfaces can be written in a variety of languages, and run on a number of different platforms. Furthermore, these interfaces can be application-specific - the underlying SOAP APIs can be completely hidden from the end-user.

The area of Web service wrappers to scientific applications is certainly not without precedent. The Java CoG Kit [37] provided a way to expose legacy applications as Web services as early as 2002. It uses a *serviceMap* document to generate source code for the Web service implementation, along with the WSDL's for the same. However, source code generation is not very flexible - it is highly dependent on the version of the Web service software being used, and not very conducive to the addition of new features. Furthermore, the above is only a prototype, and does not support complex features such as Grid scheduling, concurrent and asynchronous job submissions, data management for jobs, authentication and authorization, etc.

The *Generic Factory Service* [33] developed at Indiana University is a much more sophisticated toolkit for wrapping scientific applications, that relies on the XSUL SOAP libraries [13] for Web services support. It also uses a *serviceMap* to generate new WSDL descriptions for every scientific application. However, it does not rely on source code generation. Instead, it uses an XSUL *Message Processor* to intercept the SOAP calls for a particular Web service and route it to a generic class that invokes the scientific application using the information provided by the serviceMap document. The disadvantage of this implementation is that this approach is very XSUL specific. Most scientific communities such as NBCR, GEON [1], etc. prefer to use commodity SOAP toolkits (e.g. Apache Axis) for their Web services development.

SoapLab [5] is another advanced toolkit for wrapper gen-

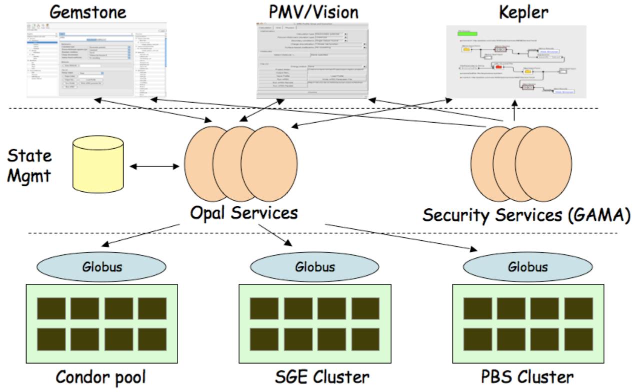


Fig. 1. The end-to-end Web services architecture

eration. They define application configurations using the *ACD* format, and generate the WSDL, and the source code for the Web service implementations from them. They use CORBA for discovering, starting, and controlling applications. As we discussed earlier, source code generation has several disadvantages. Furthermore, the use of CORBA is non-standard in the Grid world - any additional software that needs to be installed and accessed is traditionally met with a lot of inertia by the scientific community. In our experience, the use of standard commodity software to build our applications, and keeping our software requirements to the bare minimum have proven to be the most prudent. This has been our primary motivation in developing the Opal toolkit. With that in mind, we describe the big picture and the important technical details of Opal in the following sections.

III. Architecture Overview

The overall end-to-end architecture of our system has been described in detail in [34]. However, for the sake of completeness, we provide a brief overview in this section.

Figure 1 shows our multi-tiered architecture, with the compute resources in the bottom tier, the Web services layer in the middle tier, and the end-user interfaces in the top tier. The bottom tier is composed of multiple clusters hosted at different sites. NBCR has several clusters located at different geographical locations that run different schedulers of their choice, e.g. Condor [20], the Sun Grid Engine (SGE) [9], etc. The Web services in the middle tier provide access to the scientific applications on these clusters through SOAP APIs that are independent of the infrastructure being used at the back-end. At the top tier, the user-interfaces provide application-specific interfaces that can be easily used by the scientific end-user. These include Gemstone [6], a Mozilla Firefox-based client for dynamically configurable Web service interfaces, the Python Molecular Viewer (PMV) [8], and workflow tools such as Kepler [18], and Vision [11].

Some of the key features of the end-to-end architecture are summarized in the following subsections.

A. Scheduling & Cluster Management

Since different sites run different schedulers, it is mandatory that these be accessed in a uniform way for maximum code reuse. Furthermore, these schedulers need to be accessible programmatically by the Web service implementations, and not via their regular command-line interfaces. We use the Globus Resource Allocation Manager (GRAM) API for submitting and managing Grid jobs. Jobs can be created using a scheduler-agnostic Resource Specification Language (RSL). A scheduler-specific plug-in, called the *gatekeeper*, installed on the Grid resource interprets the RSL and performs a job submission to an appropriate scheduler on behalf of the user. The Java client side libraries to GRAM are provided by the Java CoG Kit. This makes the Web services code easily portable across sites that use different schedulers. Using a different scheduler is as easy as changing the URL for the Globus *gatekeeper*, assuming that the Globus GRAM service is properly configured on the Grid resource.

The Web services are themselves hosted inside a Jakarta Tomcat container, which is responsible for providing desirable qualities of service such as scalability and reliability (with the possible use of load balancing over a set of Tomcat servers).

B. Data Management and Persistence

The Web services provide the requisite data management for user jobs. When a user requests a job run, a new working directory is created for the same. All the inputs are transferred to this directory, and the scientific executable is run in this working directory. The user immediately receives a unique *jobID* that can be used later to query for job status and retrieve outputs. Long running jobs are easily supported since the client does not have to block (and possibly time out) until the job is completed. Furthermore, multiple jobs submitted by different users can be run concurrently since they are executing in separate working directories. However, this approach makes the Web service *stateful*. If the Web service happens to crash during a run, its state can possibly be lost. However, these Web services can be optionally configured to store their states to a PostgreSQL database, accessed via JDBC. Apart from job status and metadata about job inputs and outputs, the service state also includes user information and job history. In case the Web services are to go down, they can be simply restarted and are able to resume almost seamlessly using the state stored in the database.

C. Security

One of the key contributions of the Globus toolkit is the Grid Security Infrastructure (GSI) [30]. GSI is a public-key system that uses X.509-based [27] user and host certificates signed by trusted third parties called Certificate Authorities (CAs). Typical usage models require that each user is assigned a user credential consisting of a public and private key. Users generate *delegated proxy* certificates with short life spans that get passed from one component to another and form the basis of authentication, access control and logging. However, GSI-based systems are known to be very difficult to administer and use.

We use the Grid Account Management Architecture (GAMA) [21] to address this problem for two reasons - first, it provides a simple portal interface for an end-user and site-administrator to create GSI credentials. This interface abstracts out several software packages for building CAs into a single set of services for easy deployment and administration. Second, it provides Web service APIs to securely retrieve proxies from the back-end server that can be used by a variety of clients on different platforms, without the installation of any complicated CA software.

Authentication between clients and the Web services is performed at the *transport level* for performance reasons. It relies on the creation of a secure point-to-point connection between the client and server, using a GSI-based Secure Sockets Layer (SSL) implementation that can be set up using the security libraries provided by the Java CoG kit. Once the client is authenticated, a call-out is performed by an Apache Axis *Handler* to perform authorization before the Web service is invoked. Currently, a *grid-map* (access-control) based authorization is used; however, Axis Handlers that can perform call-outs to authorization services such as Virtual Organization Membership Service (VOMS) [16], or the Community Authorization Service (CAS) [26] are reasonably easy to implement.

D. User Interfaces

The application services that wrap the scientific applications are accessible via programmatic APIs; however, most users do not prefer to access their scientific applications through programmatic interfaces. Instead, users interact with these services with the help of a variety of science-oriented interfaces, as appropriate for their needs. For example, portal interfaces accessible via Web browsers provide ubiquitous access to the services, but are not very flexible. Other tools provide richer desktop environments, but tend to be more heavy weight. We do not mandate a single user interface to be used by all end-users - instead, the Web service APIs for the services are meant to be able to be sufficiently flexible to enable access via a number of different clients.

One of the user interfaces being used in the NBCR community is Gemstone. Although a detailed description of the Gemstone interface is beyond the scope of this paper, the key idea is that it provides a *shell* in which different service panels (application user interfaces) can be loaded dynamically and executed in order to interact with the back-end services. The user interface elements are described using an XML syntax called the XML User-interface Language (XUL) [22]. It defines all the GUI elements such as buttons, text fields, and menu items. The GUI elements are tied together using Javascript which is natively interpreted by the Mozilla SpiderMonkey Javascript interpreter. Gemstone enables the access of remote Web services dynamically since the presentation logic (XUL), as well as the business logic (Javascript), can be loaded appropriately at runtime.

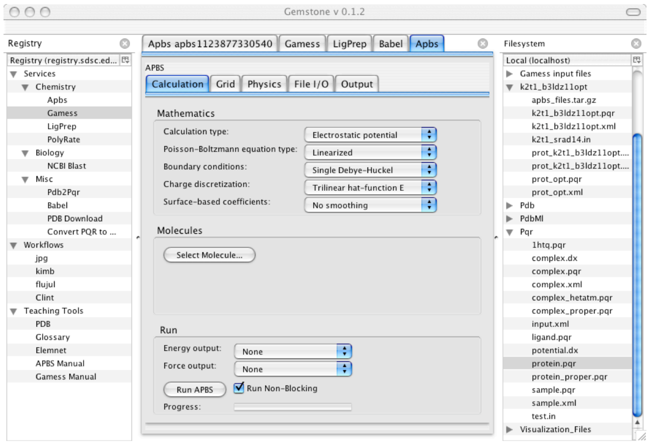

Calculation type: Electrostatic potential
Poisson-Boltzmann equation type: Linearized
Boundary conditions: Single Debye-Huckel
Charge discretization: Trilinear hat-function E
Surface-based coefficients: No smoothing
Energy output: None
Force output: None
☑ Run Non-Blocking
Progress:

Fig. 2. Molecular science through Gemstone

Figure 2 shows the look-and-feel of the Gemstone user interface. The left panel is a registry of services that is loaded dynamically, the middle is the rendering of the application-specific user interface described by the XUL, and the right is a repository of user data that can be dragged and dropped into the middle panel. In Section V, we will demonstrate how the Gemstone interface has been used to access Opal and other services to enable simulations of novel ligand-protein interactions, useful in the field of pharmaceutical design.

Opal services are also being accessed via workflow tools such as Kepler. Kepler uses the concept of *Directors* to define the type of workflow being implemented, e.g. communicating sequential processes, synchronous data flow, etc. It uses the concept of *Actors* to perform certain actions when triggered. Actors have input and output ports which are used to consume and produce data respectively. The use of Kepler for a bio-informatics workflow using Opal services will be demonstrated in Section V.

IV. Implementation Details

In this section, we focus on the Opal services themselves, and describe them in greater detail. We look at how it is implemented, and the steps involved in exposing an existing scientific application as a Web service.

Figure 3 shows the implementation of the Opal services inside the Web services container. The Opal services are developed using the commodity Apache Axis toolkit, and are hosted within the Jakarta Tomcat container. Multiple instances of the Opal services may exist within the same container, each wrapping a different scientific application, and accessible via unique URLs.

A. Container Properties

The container itself can be configured with a static set of properties. These include properties of the computational infrastructure such as Globus and database setup information, the number of nodes in the compute cluster, etc.

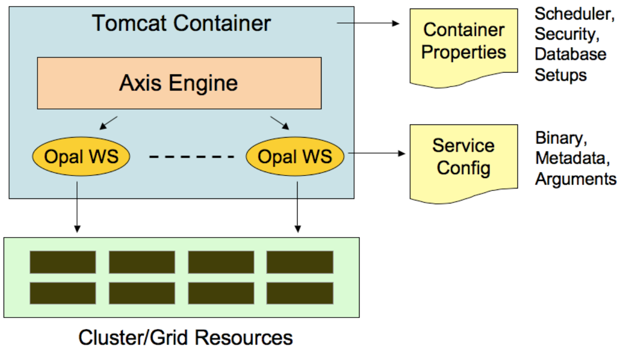
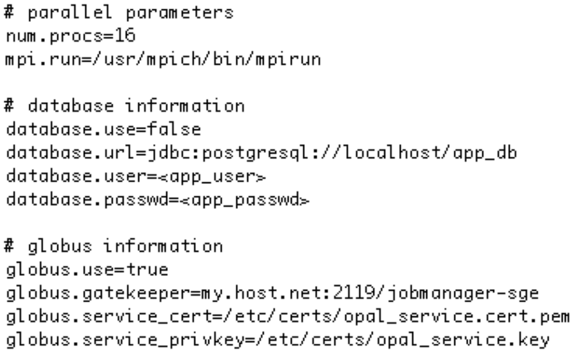


Fig. 3. Opal services - Implementation details



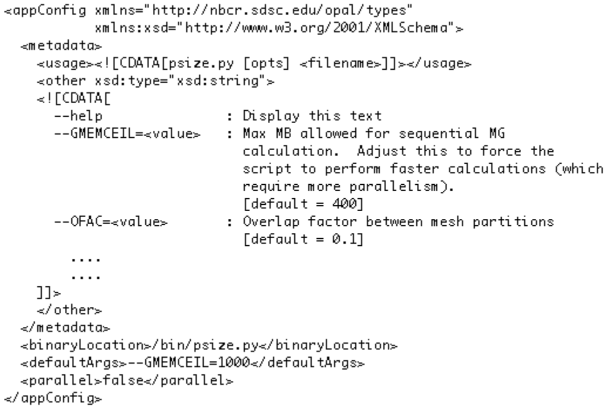
```
# parallel parameters
num.procs=16
mpi.run=/usr/mpich/bin/mpirun

# database information
database.use=false
database.url=jdbc:postgresql://localhost/app_db
database.user=<app_user>
database.passwd=<app_passwd>

# globus information
globus.use=true
globus.gatekeeper=my.host.net:2119/jobmanager-sge
globus.service_cert=/etc/certs/opal_service.cert.pem
globus.service_privkey=/etc/certs/opal_service.key
```

Fig. 4. Opal Container properties

Figure 4 shows an example of the container properties. The number of available processors is specified by the property `num.procs`, and the location of MPI for parallel execution is specified by the property `mpi.run`. If the `database.use` property is set to `true`, the `database.url`, `database.user`, and `database.passwd` properties are used to persist Web service state into a database (assuming the tables are appropriately set up). Similarly, if the `globus.use` property is set to `true`, the `globus.gatekeeper`, `globus.service_cert`, and `globus.service_privkey` are used to submit jobs to an appropriate scheduler using Globus GRAM. If Globus is not used, all jobs are run using simple local process forks.

B. Application Configuration

Every application is described by an application configuration file. This describes the location of the binaries, if it is a parallel application or not, default arguments, and application metadata. Application metadata includes the usage and other optional information specified by the application provider. This can include helpful messages for understanding the various parameters, and inputs and outputs. Currently, this is meant for human consumption - there are no programmatic tools that can interpret the metadata information. However, as part of our future work described in Section VI, we plan on adding presentation logic for use in user interfaces, and data format descriptions for use in workflow tools (for automatic translations between various formats). A sample application configuration for the *Psize* application is shown in Figure 5.



```
<appConfig xmlns="http://nbcr.sdsc.edu/opal/types"
           xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <metadata>
    <usage><![CDATA[psize.py [opts] <filename>]]></usage>
    <other xsd:type="xsd:string">
    <![CDATA[
      --help                : Display this text
      --GMEMCEIL=<value>    : Max MB allowed for sequential MG
                              calculation.  Adjust this to force the
                              script to perform faster calculations (which
                              require more parallelism).
                              [default = 400]
      --OFAC=<value>        : Overlap factor between mesh partitions
                              [default = 0.1]
        ....
        ....
    ]]>
    </other>
  </metadata>
  <binaryLocation>/bin/psize.py</binaryLocation>
  <defaultArgs>--GMEMCEIL=1000</defaultArgs>
  <parallel>false</parallel>
</appConfig>
```

Fig. 5. Opal Application Configuration

C. Service Deployment

Apache Axis uses a Web Services Deployment Descriptor (WSDD) to deploy a Web services. The information in a descriptor includes the name of the Web service, the class implementing the Web service, a list of type mappings, etc. The Axis toolkit uses the information inside the WSDD to appropriately configure and deploy a Web service, and ensures that remote invocations are routed to appropriate implementation of the Web services.

The Opal toolkit provides a template WSDD for deployment purposes. The service provider needs to make two changes to the template WSDD. The name of the service needs to be changed to a unique name that reflects the scientific application being wrapped. This name will be appended to the base URL for the Axis Web Application (*webapp*) to create a unique URL that can be used for accessing the service. Also, the parameter *appConfig* needs to be modified to point to the application configuration for the scientific application. The service can then be deployed easily using an Apache Ant *target*. When the service is invoked by a client for the first time, it will read the configuration file referred to by the appConfig, and configure itself to run the appropriate scientific application, using the static properties specified for the container. In the future, deployment of Opal services will be performed using a GUI; this will shield the user from having the modify the WSDD by hand.

D. WSDL API

As mentioned in Section II, automatically generating both source code and WSDL's can be very specific to the SOAP toolkits being used. With an eye on simplicity and configurability, we use a static WSDL for every service that is deployed using Opal. Legacy scientific applications expose only a command line interface to users, and can only be run with command line arguments. Since they do not expose specific operations that can be run with different parameters (unlike Web services), it suffices to simply expose one single method to execute them - *launchJob*. The input parameter to this operation is an XML data structure containing the argument list as a string (which exactly corresponds to the arguments of the scientific application), and a list of data structures encapsulating the input files. The data structure for an input file consists of a *name*, and the *contents* which is a Base64 encoded binary representation of the file contents. The launchJob operation returns a *jobId* that can be used to query for job status, and eventually retrieve job outputs using the *getOutputs* operation. The getOutputs operation returns a list of URLs from where the output files for the scientific applications can be retrieved. Furthermore, Opal services also expose a *getMetadata* operation that can be used to retrieve information about the scientific application such as the usage, input and output information, etc. This information is retrieved directly from the application configuration described above. Furthermore, jobs can be killed at any time via the *destroy* operation using the jobID.

Since Opal services use a static WSDL, every scientific application deployed as an Opal service has the same WSDL. Two Opal services can be differentiated by their unique URLs, and their associated metadata. This metadata can be published in registries, so that Opal-based scientific applications can be discovered and used dynamically by various clients.

V. Use Cases

So far, we have seen how easy it is to expose a scientific application as a Web service using in the Opal toolkit. In this section, we will discuss a few scenarios where Opal-based services have been composed into meaningful scientific workflows, and accessed via a variety of clients.

A. Bio-informatics Workflows using Opal and Kepler

The Opal toolkit has been used to wrap several bio-informatics applications as Web services. Two applications of interest are MEME [4] and MAST [3]. MEME enables a user to discover motifs (highly conserved regions) in groups of related DNA or protein sequences. MAST enables a user to search sequence databases using motifs. These applications are exposed very easily as Web services by creating appropriate application configuration files. The Web services container is configured to submit jobs to a back-end computational cluster using the Sun Grid Engine by setting the appropriate container property.

The Kepler workflow framework has been used to create a scientific pipeline consisting of the Opal-based MEME and MAST Web services, as shown in Figure 6. The workflow has been created with the help of mostly standard Kepler *actors*; however, a couple of actors needed to be written for input generation for the MEME and the MAST services. The MEME input generator first accepts user inputs, generates a SOAP message, and then makes an invocation on the MEME service to launch a MEME job. The service returns an XML message containing a *jobID*, which is extracted using XPath/XSLT. Using the jobID, the client queries for job status and blocks until the job is complete. Once it is done, the service returns a URL where the results can be viewed and downloaded. The MEME output is then downloaded and sent to the MAST service inside a SOAP message as a Base64 encoded binary string, along with the other MAST parameters, by the MAST input actor. The MAST service also returns a jobID, and the client again queries for the job status and blocks until the job is complete. Once MAST is done, a URL is returned that can be used to retrieve the MAST results. Thus, with the use of a Web services wrapper toolkit like Opal, and a workflow toolkit like Kepler, we are able to easily compose two scientific applications running on complex back-end computational infrastructure. It is worthwhile to note that the Kepler workflow toolkit is completely oblivious to several back-end details - how the applications are deployed, how the jobs are submitted on the Grid, etc. These are exactly the details that we strived to abstract out.

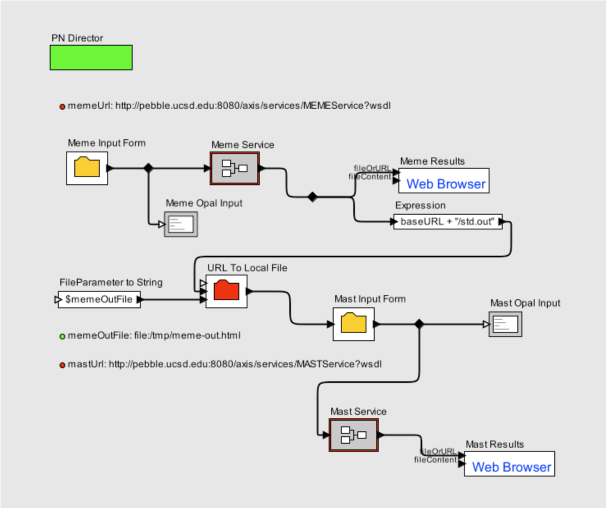


Fig. 6. Kepler-based Meme-Mast workflow

In this example, it so happens that the outputs of MEME can easily be consumed by MAST, without having to perform any data conversions. This makes the workflow very straightforward to implement. In several cases, this is typically not true. In such cases, data conversions between various formats becomes necessary if Opal services are used for every stage of the workflow. However, strongly typed Web services that wrap certain scientific applications can also be written by hand (i.e. without the use of Opal). Strongly typed services aid in the extraction and translation of data types using general purpose workflow tools [34]. A combination of Opal and strong typed services can then be used to accomplish complicated workflows, as described in the following subsection.

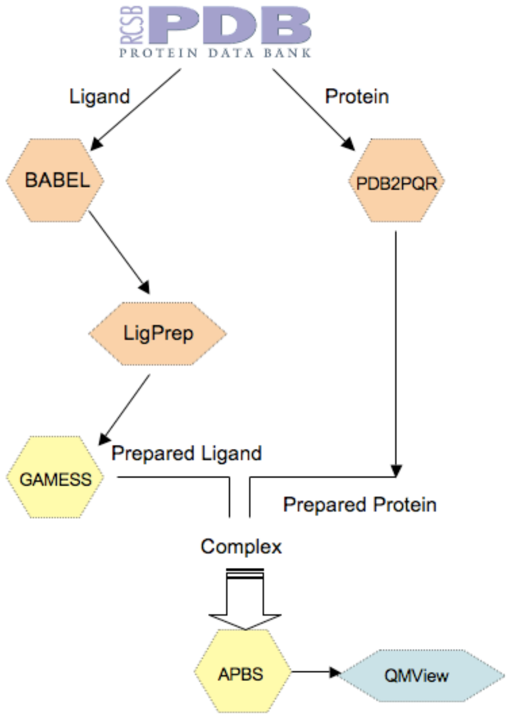


Fig. 7. Ligand-Protein Interaction using Opal and Strongly Typed Services

B. Molecular Science using Opal and Gemstone

Estimating correct three-dimensional atomic structures of complexes between proteins and ligands is an important component of the drug-design process in the pharmaceutical industry. This process usually involves extracting separate geometries for the protein and the ligand of interest from structural databases, and varying the relative orientation between the protein and the ligand until an optimal orientation is found. A highly accurate quantum chemical model is used for the small ligand, employing the computational chemistry package GAMESS [36], with the less accurate electrostatics model for the ligand-protein complex, using the APBS [31] computational package. The associated tool, QMView, is then employed for visualization and analysis.

These computations are wrapped as Web services to fit within the architecture described in Section III. However, for reasons described above, some services have been implemented as strongly typed, while others have been wrapped using Opal. The GAMESS and APBS services are written by hand, because they deal with sophisticated data structures that need to be extracted and exchanged - in particular, the molecule datatype that is used to represent proteins and ligands. The Babel, LigPrep and PDB2PQR applications perform small utility functions in the workflow, and have been wrapped using the Opal toolkit for rapid deployment.

Figure 7 shows the scientific workflow that has been implemented using the above setup. First, an interesting protein and a ligand are downloaded from the Protein Data Bank (PDB) database. Next, the Opal-based PDB2PQR service is used to convert the protein from the PDB to the PQR format. Then, the Opal-based Babel service is used to add Hydrogens to the ligand, which is missing from the PDB format. The LigPrep service, also Opal-based, is then used to generate rotational conformations for the ligand. Next, the strongly typed GAMESS service is used to generate accurate charges for the ligand. The strongly typed APBS service is then used to calculate the binding energy for the resulting complex generated by the protein and ligand. The results from APBS are then visualized using the QMView client.

Note that Gemstone currently does not support automation of workflows, although that is part of the future plans. Currently, the workflow described above is driven interactively by an end-user.

VI. Future Work

Opal is just the first step in Grid-enabling scientific applications. Accessing Opal services via application-specific interfaces that encapsulate the underlying computer science APIs still involves some work. We are currently working on trying to expose the presentation logic for Opal services remotely, along with the business logic to translate from the user inputs to the underlying SOAP messages. Once this is defined, custom clients for every application will not have to be written. Instead, generic clients capable of dynamically downloading the user interfaces can be designed, and used for all applications. This is similar to the Gemstone model described in Section III, but is generic enough to be accessible by interfaces that are not necessarily built on top of the Mozilla platform.

One disadvantage of using Opal is that the scientific applications still use their legacy input and output formats. Even in the same community, people define their own custom formats for representing similar entities, e.g. in the molecular science community, there exists several definitions for the representation of a protein or a molecule - PDB, PQR (used by APBS), CML [35], etc. If every application uses its own data formats that are flat-file based (in other words, not strongly structured or typed), it is very difficult for third-party tools to interpret the data without human intervention. Furthermore, composition of workflows becomes extremely complicated because it involves custom data translations from one format to another - these have been known to be very error prone and not scalable, since there is a need to write routines to convert from one format to another. We plan to use Data Format Definition Languages (DFDL) [14] to describe application-specific formats as abstract data structures, and develop generic tools operating on these abstract structures that perform translations from one format to another. These can be easily integrated inside workflow tools, thus enabling the creation of complex scientific pipelines.

From the technical perspective, there are a few performance and engineering improvements that can be made. In the future, we plan to enable access to the application output data using standard high-performance Grid file transfer tools such as GridFTP [17]. Furthermore, the stateful Web services approach of Opal services lends itself very well to be adapted into the WSRF world. The Web services state can be represented easily as a WS-Resource, and can be accessed by standard WSRF mechanisms. Clients can then be notified of state changes using asynchronous one-way messages provided by WS-Notification [15]. Lifetime management of job inputs and outputs can also be performed as specified within the WSRF model. Opal services can then be accessed by standard WSRF clients, and incorporated very easily into Grid toolkits.

VII. Conclusions

In this paper, we presented Opal, a toolkit for wrapping scientific applications as Web services in a matter of a few hours. Once the scientific applications are deployed as Opal services, multiple users can concurrently access these applications, via a multitude of user-interfaces. Opal itself can be configured to submit jobs to schedulers on Grid resources using the Globus toolkit, and save its state into a database, if need be. Furthermore, it can be set up to use GSI-based authentication and authorization mechanisms for secure application access. We described the technical details of the Opal implementation, and demonstrated access to Opal services via the Mozilla Firefox-based Gemstone framework, and the Kepler workflow toolkit.

For more information about our work, including software releases, readers are strongly encouraged to visit our Web-site: http://nbcr.net/services.

VIII. Acknowledgments

We thank the NIH for supporting NBCR through the National Center for Research Resources program grant P41RR08605, and the NSF for supporting Gemstone through the Middleware Grant SCI-0438430. We also thank Jerry Greenberg, the members of the Kim Baldridge Research Group at the University of Zurich, Robert Konecny, and Michel Sanner for their invaluable feedback on using Opal for Grid-enabling their scientific applications, Kurt Mueller for his work on GAMA and portlets for Gridsphere-based access to Opal services, Ilkay Altintas, Efrat Jaeger, and Nandita Mangal for their help on using Kepler, and Chris Misleh for using Opal to expose the MEME and MAST applications as Web services, and providing general infrastructure support.

References

[1] GEON: The Geosciences Network. http://www.geongrid.org/.

[2] GridLab: A Grid Application Toolkit and Testbed. http://gridlab.org/.

[3] MAST – Motif Alignment and Search Tool. http://meme.sdsc.edu/meme/mast-intro.html.

[4] MEME – Multiple EM for Motif Elicitation. http://meme.sdsc.edu/meme/meme-intro.html.

[5] Soaplab - Analysis Web Service. http://www.ebi.ac.uk/soaplab/.

[6] The Gemstone Project. http://grid-devel.sdsc.edu/gemstone/.

[7] The National Biomedical Computation Resource (NBCR). http://nbcr.net.

[8] The Python Molecular Viewer (PMV). http://www.scripps.edu/ sanner/python/pmv/.

[9] The Sun Grid Engine (SGE). http://gridengine.sunsource.net/.

[10] The TeraGrid Project. http://www.teragrid.org.

[11] The Vision Programming Environment. http://www.scripps.edu/ sanner/python/vision/.

[12] TOP500 Supercomputer Sites. http://www.top500.org/.

[13] WS/XSUL2: Web and XML Services Utility Library (Version 2). http://www.extreme.indiana.edu/xgws/xsul/index.html.

[14] Data Format Description Language (DFDL), 2005. http://forge.gridforum.org/projects/dfdl-wg/.

[15] Akamai, C.A.I., Fujitsu, Globus, HP, IBM, SAP AG, So nic, and TIBCO. Web Services Notification, June 2004. http://www-106.ibm.com/developerworks/library/specification/ws-notification/.

[16] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell'Agnello, A. Frohner, A. Gianoli, K. Lorentey, and F. Spataro. VOMS: An Authorization System for Virtual Organizations. In *1st European Across Grids Conference, Santiago de Compostela*, 2003.

[17] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster. The Globus Striped GridFTP Framework and Server. In *Super Computing 2005 (SC05)*, 2005.

[18] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludaescher, and S. Mock. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *16th International Conference on Scientific and Statistical Database Management (SSDBM'04)*, 2004.

[19] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic Local Alignment Search Tool. In *J. Mol. Biol.*, volume 215, 1990.

[20] J. Basney, M. Livny, and T. Tannenbaum. High Throughput Computing with Condor. In *HPCU news, Volume 1(2)*, June 1997.

[21] K. Bhatia, S. Chandra, and K. Mueller. GAMA: Grid Account Management Architecture. Technical report, SDSC, UCSD, 2005. SDSC TR-2005-3.

[22] The Mozilla Corporation. XML User Interface Language (XUL). http://www.mozilla.org/projects/xul/.

[23] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *IPPS/SPDP 98, Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[24] A. Bosworth et al. Web Services Addressing, March 2004. http://www-106.ibm.com/developerworks/library/specification/ws-add/.

[25] K. Czajkowski et al. WS-Resource Framework, May 2004. http://www-106.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf.

[26] L. Pearlman et al. A Community Authorization Service for Group Collaboration. In *IEEE 3rd International Workshop on Policies for Distributed Systems and Network*, 2002.

[27] S. Tuecke et al. Internet X.509 Public Key Infrastructure Proxy Certificate Profile, 2003. IETF.

[28] I. Foster and C. Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.

[29] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *Computer 35(6)*, 2002.

[30] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *ACM Conference on Computers and Security*, 1998.

[31] M. Holst and F. Saied. Numerical solution of the nonlinear poisson-boltzmann equation: Developing more robust and efficient methods. In *J. Comput. Chem., 16*, 1995.

[32] C. Kesselman I. Foster. Globus: A Metacomputing Infrastructure Toolkit, 1997.

[33] Gopi Kandaswamy, Liang Fang, Yi Huang, Satoshi Shirasuna, Suresh Marru, and Dennis Gannon. Building Web Services For Scientific Grid Applications. In *IBM Journal of Research and Development*, 2005.

[34] Sriram Krishnan, Kim Baldridge, Jerry Greenberg, Brent Stearn, and Karan Bhatia. An End-to-end Web Services-based Infrastructure for Biomedical Applications. In *6th IEEE/ACM International Workshop on Grid Computing*, 2005.

[35] P. Murray-Rust and H. S. Rzepa. Chemical Markup, XML, and the World Wide Web. 4. CML Schema. In *J. Chem. Inf. Comput. Sci.*, volume 43, 2003.

[36] M.W. Schmidt, K.K. Baldridge, J.A. Boatz, S.T. Elbert, M.S. Gordon, J.J. Jensen, S. Koseki, N. Matsunaga, K.A. Nguyen, S. Su, T.L. Windus, M. Dupuis, and J.A. Montgomery. GAMESS. In *J. Comput. Chem., 14*, 1993.

[37] G. von Laszewski, J. Gawor, S. Krishnan, and K. Jackson. *Grid Computing: Making the Global Infrastructure a Reality*, chapter 25, Commodity Grid Kits - Middleware for Building Grid Computing Environments. Wiley, 2003.