Design and Evaluation of Opal2: A Toolkit for Scientific Software as a Service

Sriram Krishnan, Luca Clementi, Jingyuan Ren, Philip Papadopoulos and Wilfred Li
San Diego Supercomputer Center
University of California, San Diego
La Jolla, CA 92093, USA
Email: {sriram, clem, jren, phil, wilfred}@sdsc.edu

***Abstract*—Grid computing provides mechanisms for making large-scale computing environments available to the masses. In recent times, with the advent of Cloud computing, the concepts of Software as a Service (SaaS), where vendors provide key software products as services over the internet that can be accessed by users to perform complex tasks, and Service as Software (SaS), where customizable and repeatable services are packaged as software products that dynamically meet the demands of individual users, have become increasingly popular. Both SaaS and SaS models are highly applicable to scientific software and users alike. Opal2 is a toolkit for wrapping scientific applications as Web services on Grid and cloud computing resources. It provides a mechanism for scientific application developers to expose the functionality of their codes via simple Web service APIs, abstracting out the details of the back-end infrastructure. Services may be combined via customized workflows for specific research areas and distributed as virtual machine images. In this paper, we describe the overall philosophy and architecture of the Opal2 framework, including its new plug-in architecture and data handling capabilities. We analyze its performance in typical cluster and Grid settings, and in a cloud computing environment within virtual machines, using Amazon's Elastic Computing Cloud (EC2).**

***Keywords*-Clouds; Service Oriented Architectures; Grid Computing;**

I. Introduction

In the mid 1990s the Globus team introduced the concept of the computational Grid [1] as way to make large-scale computing available to the masses. While there are many positive outcomes of this research and indeed most computationally-driven scientists assume richly-networked infrastructures, there are key issues like reliability, ease-of-use, access to data, and the lack of understanding of the software environment on remote resources that make this original view of the Grid problematic for many users. It is simply too complicated for an individual code to be spread across resources because the different execution environments are frequently uncertain, and overly heterogeneous. Any code has to be ported and tested on every Grid endpoint for the ensemble to work. However, many elements of Grid infrastructure (GSI authentication [2], striped file transfer across networks [3], Virtual organization [4] management, and service-oriented architectures) are now used routinely to enable collaborations and support authenticated access to remote services.

Cloud computing, which is a successor to the Grid, has two main branches from the system/programming viewpoint: (1) a set of vendor-defined scalable key services that may be directly consumed by end users, or composed by high level service providers to build more customized solutions (the Google model), and (2) access to a scalable virtual machine hosting environment where the system configuration (operating system level and up) are defined by the user (the Amazon EC2 Model [5]). Each addresses a significant subset of the early Grid paradigm but does not solve the entire problem. However, both approaches make it possible to have consistency of software on remote resources and reduce porting costs. Google accomplishes this through a fixed programming paradigm on production-level services that they define. Amazon enables the user to define the entire system environment, and deploy it on as many identical instances as the pocketbook allows.

Service-oriented computing represent a significant step forward in providing a real architecture for writing Grid-based programs ([6], [7]). In the commercial (and academic) space, Web services are now standard practice in making very complex systems (like Google Maps or Search) callable through a standard application programming interface (API). Legacy codes may be frequently wrapped as Web services to provide remote access to one-of-a-kind application services that are available across the Internet. The Web service APIs may also be viewed as a composition enabler for building workflows or value added services. Service-based computing forms an important capability, providing applications with the desired dynamic scaling inside a cluster or a data center, to enable service based composition to perform complex tasks.

Technologies like Xen [8] and VMWare enable multiple virtual machines to co-exist on a single physical multi-core machine and present the illusion that there are multiple physical machines. The commercial offerings from Amazon (EC2) allow users to rent virtual machine space for cents/hour. Not only do their virtual machines look and feel like real hardware, but their contents can be completely defined by the user. The revolution in virtualization is increasing the availability of hardware while dramatically reducing its cost for occasional use. It points to a radical shift in the way infrastructure is built and where it is physically

located (local, remote, virtual and combinations).

We strongly believe in paradigms of Software as a Service (SaaS - the Google model) and Service as Software (SaS - the Amazon model) for scientific computing. Firstly, we believe that that SaaS paradigm for scientific software enables us to (1) provide higher-level scientific services, focusing on improving scientific pipelines and workflows, rather than lower-level infrastructure services, and (2) build repeatable solutions for scientific Grid software that can be leveraged by multiple clients. Secondly, we believe that the SaS paradigm enables us to package our scientific software tools as deployable units (in our case, virtual machine images and Rocks rolls [9]) which can be customized and used by a far broader community of scientific software developers and service providers.

In this paper, we present a realistic architecture for scientific Software as a Service and Services as Software using the Opal2 toolkit. We describe the design and implementation of Opal2, and analyze its performance when it is used in a typical cluster and national Grid setting, and when it is used in a cloud computing environment (using EC2). The rest of the paper is organized as follows. In Section II, we provide the background and design goals for Opal2. In Section III, we discuss the architecture of the Opal2 framework, and present implementation details. In Section IV, we evaluate the Opal2 framework for its performance, and present a typical usage scenario. We discuss our related work in Section V and conclusions in Section VI.

II. Background and Design Goals

With the introduction of the Web Service Resource Framework (WSRF) [10], Web services gained widespread acceptance in Grid computing. Globus [11], which is the one of the most popular middleware toolkits for Grid computing, provided WSRF-based interfaces to common Grid services. For instance, scientific jobs could be launched on Grid resources using the WSRF-based Grid Resource Allocation Management (GRAM) API [12].

Opal is not yet another API for resource allocation and management. Instead it leverages APIs such as GRAM at the back-end, to expose scientific applications as first class Web services themselves, accessible via simple application-oriented interfaces. We believe that a middleware-oriented API such as GRAM is suitable for use only by Grid systems developers - scientific application developers would rather focus on the algorithms and appropriate scientific interfaces to present to their end-users. Enabling access to scientific applications and tools using a service-oriented approach allows the developers of scientic tools to focus on the domain science, and delegate the management of the complex back-end resources to others who are more proficient in Grid middleware.

The Opal toolkit (described in detail in [6]) provides a mechanism to deploy scientific applications as Web services without having to write a single line of code. Once deployed, scientific applications are available for access through a simple Web service API for launching jobs, monitoring status, retrieving status, etc (via a standard WSDL). The API abstracts out the complexity involved in submission of computational jobs to Grid resources, since the client simply submits command-line arguments and input files during job launch. The toolkit provides management of user data, including the creation of working directories, input and output data staging, and persistent storage for job information and metadata. GSI-based security is optionally supported, for authentication and authorization. Opal also provides an optional XML-based specification for command-line arguments, which is used to generate automatic Web forms for invocation via the "dashboard", which also provides usage statistics.

Opal is a SourceForge project, and has been used for providing production services and workflows at the National Biomedical Computation Resource [13] and other projects ([14], [15], [16]). While we view Opal as a very successful technology innovation, our prior work represents only a first step and several improvements could be made. Based upon experiences from production use and user feedback, the following areas have been targeted for improvements in Opal2.

Performance: The Opal toolkit was built on top of version 1.2 of Apache Axis, and input files were transferred to the Opal server using a Base64 encoded binary representation within a SOAP message. We discovered that it was only realistic to transfer input files of the order of tens of megabytes using this mechanism. Furthermore, Opal didn't provide a mechanism to do third-party transfers – all input files had to transferred by the client to an Opal server since it didn't have the capability of fetching input files from a remote URL. The ability to do third-party transfers is especially useful in a workflow environment, where data should ideally be transferred from one node of the workflow directly to another, rather than being re-routed via the client.

Packaging: Even though the Opal toolkit is fairly easy to deploy, it was highly desirable to be able to package Opal and the application suite that it wraps as a set of Rocks Rolls or virtual machine images. This would enable easy deployment on Rocks clusters and virtual machines.

Configurability: Opal was quite configurable in terms of the schedulers it supports at the back-end and the database it uses for state management. However, once these parameters were set, it was static for the entire Opal installation. It was imperative to provide a mechanism to override static properties of the system on a per-application basis, so that different types of applications could be installed on the same Opal server.

State management: Opal used JDBC to optionally configure a database to persist the job state and metadata. If the database wasn't configured, all state was stored in hash

tables in memory. Because of this, all historic data would be lost upon restarts of the Opal server. Furthermore, the Opal dashboard was incapable of using the in-memory job state to display usage statistics. Hence, it was desirable to provide a mechanism to persist server state out of the box, without having to install a real database system at the back-end, and make it available to the dashboard.

Code Modularity: The Opal toolkit was an example of prototypical software that was put into production use very early due to its initial success. A re-design of software was essential to keep the code base more manageable, so that new features could be easily added.

We believe that we have addressed most of the above requirements with Opal2. Furthermore, we feel that the software architecture is flexible enough to enable us to accommodate any further improvements in the future.

III. Architecture and Implementation

Figure 1 describes the overall services-oriented architectural goals based on the Opal2 framework. Opal2 services can be packaged as virtual machine images and/or Rocks rolls [9]. Using virtual machine images, Opal2 can be easily installed on a user's desktop machine to leverage the powerful multi-core machines that exist today. Opal2 services can be accessed via a multitude of Web service clients, including workflow tools such as Kepler [16], problem solving environments such as Vision [17], and Web portals.

Rocks is an open-source Linux cluster distribution that enables end users to easily build computational clusters and Grid endpoints. With the use of Rocks rolls, Opal2 can be installed on cluster and Grid resources in a traditional supercomputer lab setting. The use of virtualization also enables the deployment of the software stack on cloud computing resources, such as the ones provided by the Amazon Elastic Computing Cloud (EC2). Finally, with the use of meta-schedulers such as CSF4 [18] and Condor [19], jobs can be scheduled across multiple Grid and cluster resources in a unified fashion.

Like earlier versions of Opal, Opal2 is still based on Apache Axis 1.2, and can be deployed within a Apache Tomcat container. Even though newer versions of the Axis toolkit are available (in particular, Axis2), the Grid community (e.g., the Globus toolkit) still uses older versions of the software. Since we believe in commoditization of software, we decided to use the same version of Axis, but work on performance improvements within the given restrictions. We discovered that we could significantly improve the performance of Opal2 with the addition of the following features:

Third-party transfers: In addition to the support for Base64 encoded inputs as the default for backwards compatibility, Opal2 supports transfer of input files from remote URLs. This is especially helpful in a scientific workflow environment, where the results from one step do not have

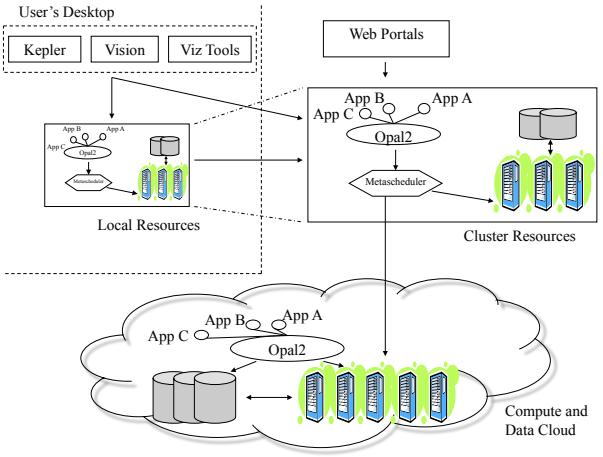

Figure 1. The Opal2 Architecture

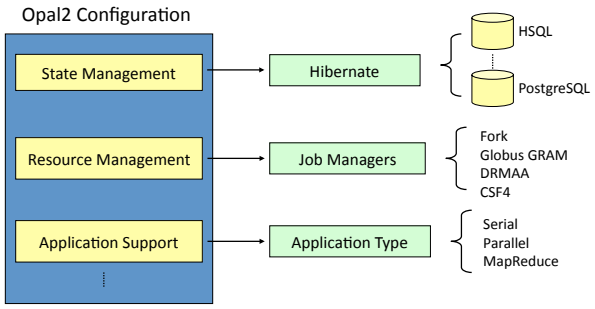

Figure 2. The Opal2 Plug-in Model

to be staged back from a remote execution resource to the client, and then back to another execution resource. Now the client only has to send the second service the location of the outputs from the first service. Currently, we provide support for HTTP and HTTPS URLs. The support of GridFTP [3] URLs is planned.

MIME attachments: Opal2 provides a way to stage input files using multi-part MIME attachments via SOAP [20]. This significantly improves the performance for a number of reasons. Firstly, data from input files need not be encoded into Base64 binary format, which causes additional overhead. Embedding this Base64 encoded binary file inside the input SOAP body increases the parsing time and memory footprint of the Apache Axis server, as demonstrated in Section IV. Using MIME attachments results in significantly improved performance results.

Other new features of Opal2 include availability of statistics like activation and execution times through the Web services API, and an Atom-based registry for services.

The mantra of Opal2 is configurability. Almost every feature of Opal2 is configurable on a per-application basis. Figure 2 shows the "plug-in" architecture of the Opal2

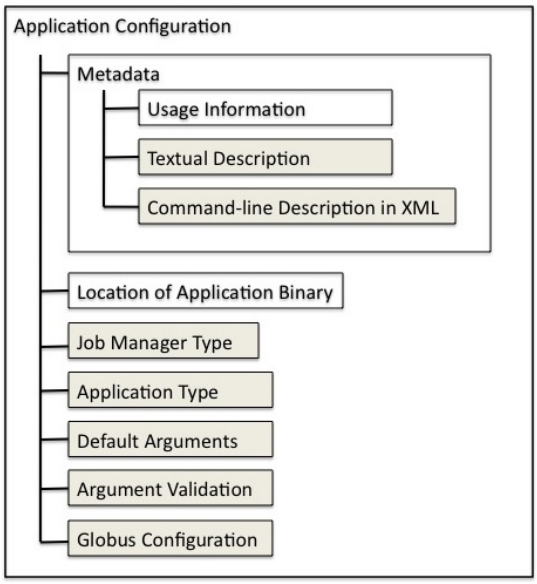

Figure 3. Opal2 Application Configuration

implementation. With the help of a set of static properties for the container, Opal2 pre-configures a set of environment variables for the system. In particular, system properties can be configured for the following entities:

State Management: State management for the Opal2 services is provided with the help of the Hibernate toolkit [21]. Hibernate provides a powerful, high performance object/relational persistence and query service. Through configuration files, Hibernate can be connected to in-memory databases, or other typical production databases (like PostgreSQL, IBM DB2, MySQL, etc). Even if Opal2 is not configured with a production database, by default it uses HSQL, which is a 100% Java database that can persist its state to a file system. Because of this feature, persistent storage of usage statistics is still available for use by the dashboard out of the box.

Resource Management: Resource management is provided by a set of job managers. Job managers currently provided include Fork (simple process exec), Globus GRAM [12] (both to a local cluster and remote Grid resources), and the Distributed Resource Management Application API (DRMAA) [22]. A job manager based on CSF4 [18] for meta-scheduling jobs across distributed resources is currently under alpha-testing. In the future, job managers for other systems like Apache Hadoop [23] may be targeted.

Application Support: The current version of Opal2 supports applications that are serial or parallel (SPMD, based on MPI). We are working on adding support for applications that require array job submission. In the future, we plan on extending this to support other types of applications such as parameter sweeps and MapReduce [24].

All the static properties of the Opal2 framework can be reconfigured on a per-application basis using the application configuration. Figure 3 shows the elements of a typical application configuration. Boxes shown in gray are optional, while clear boxes are required. The application metadata consists of usage information, which describes the command-line usage of the scientific application that is wrapped. The textual description provides more details about the application, and is optional. Both of the above elements are meant for human consumption. The command-line description of arguments in XML is optional, and is used for validation of arguments and automatic interface generation. More details about the schema for the command-line arguments, and automatic interface generation is available elsewhere [25]. The Opal2 server configures a new service for every application using this description, and can launch an executable using the location of the application binary provided. The application type (serial or parallel) can also be specified within the configuration, and the job manager type and Globus configuration can be customized per application. Optionally, command-line arguments can be validated using the XML description, by setting the appropriate flag.

The code-base of Opal2 has been redesigned to be highly modular. In particular, the job managers are implemented using the "factory" pattern. The factory pattern is an object-oriented design pattern which relies on a library that encapsulates the creation of objects at run-time. The "OpalJobManagerFactory" encapsulates the creation of job managers based on the static container properties and application-specific configuration. The Opal2 service implementation simply uses a generic "OpalJobManager" interface to launch and manage jobs, without having to deal with the intricacies of the individual job managers. This technique has helped us improve the manageability of the code, especially for adding new job managers, with minimal changes the service implementation. Similarly, the use of the Hibernate toolkit abstracts out the state management, enabling a choice of various database to be plugged in at the back-end.

IV. Experiences

In the following section, we discuss our experiences with using Opal2 in a traditional cluster environment, which has access to high speed networking, and in a cloud computing environment on Amazon EC2 over commodity internet. We conclude with a realistic use case scenario, running on TeraGrid resources [26].

A. Local Cluster Environment

For the local cluster environment, we use a 4-node cluster with dual CPU 3.06GHz Intel Xeons having 2GB RAM, and Gigabit ethernet connection. The machines are running Rocks version 4.1, with Linux kernel version 2.6.9-22, and the Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_05-b05). We set up an Opal2 server on one of

the nodes within a Tomcat 5.0.30 container, and the client is run on another node. The Opal2 server is set to use the "Fork" job manager to launch jobs on the same node. In all our tests, we only measure job submission time through Opal2, and not the actual execution time for the scientific application. This is because the only overhead caused by Opal2 is the process of job submission via the WSDL API – the actual job execution is independent of Opal, and depends on the scientific application and its input parameters. We run the tests 62 times, and average the results after dropping the two extreme data points.

We noticed that the Java heap size had no noticeable impact on the performance when MIME attachments were being used, since the attached file is streamed directly to disk without storing it in memory. On the other hand, if the input files are being transferred using the Base64-encoded binary format, the entire input file is stored in the heap, causing out of memory errors depending on the heap size. However, during heavy concurrent loads, heap size does impact the performance even if MIME attachments are used. Since we were mostly interested in testing the performance of Opal with MIME attachments, we ran our experiments with a constant heap size of 512MB and the "-server" flag active, which activates the usage of the Java HotSpot Server VM, which is optimized for maximum program execution speed for applications running in a server environment.

To compare the performance of Opal2 job submissions with regular FTP transfers, we used the Globus 4.0.1 implementation of the FTP server (GridFTP [3]), which supports vanilla FTP, as well as the ones with GSI authentication, multiple streaming, and parallel data transfers. On the client side, we used LFTP 3.0.6 [27] and the "globus-url-copy" client provided by Globus.

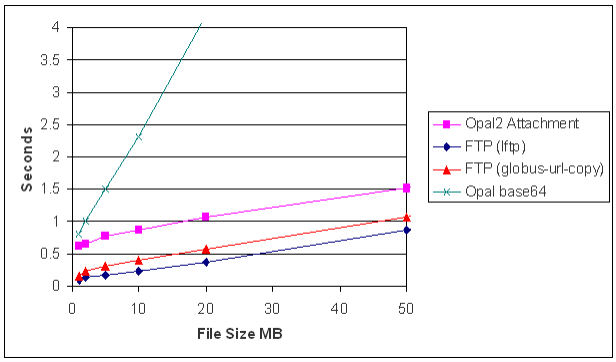

Figure 4. Execution times for small files

Figures 4 and 5 show the results we obtain with different input file sizes. LFTP performs slightly better than globus-url-copy because it is optimized for batch execution, and has a lower bootstrap time. As seen in the figures, Opal2 has a fixed overhead due to SOAP, the creation of the working directory, status updates in the database, and the execution

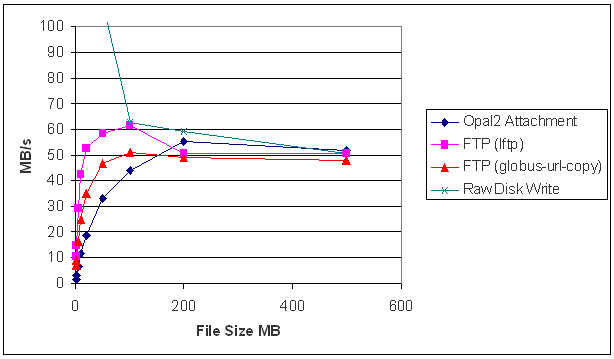

Figure 5. Data throughput comparison

of the application on the server. This overhead is observed to be around 0.3s for a submission without input files in our testing environment. When input files are small, the fixed overhead is the main component of the overall response time, lowering the observed throughput. When the input files are large than 100MB, the Opal2 throughput is comparable to FTP, because the fixed overhead becomes negligible as compared to the overall response time. It can also be seen that the performance of MIME attachments is significantly better than the Base64 encoded binary format.

In Figure 5, we have also plotted the throughput for disk writes. The throughput for small files is comparable to that of RAM because of OS caching; however, when the files are larger than 50MB, the effect of caching is almost negligible. The Opal2 performance with MIME attachments is practically the same as the disk throughput as the file size grows. Thus, the improvement of Opal2 over earlier versions is two-fold. Firstly, there is no upper-limit on file size that was imposed by the use of the Base64 encoded binary format (due to the fact that it was stored in memory). Secondly, the performance of MIME attachments is significantly better than the Base64 encoded binary format used by prior versions.

Finally, we perform some tests with security enabled, as shown in Figure 6. We compare the performance of Opal2 with MIME attachments over HTTPS, GridFTP with and without data encryption, and SCP, which uses OpenSSH 3.9p1 and encryption routines from OpenSSL 0.9.7a. We observe that the performance of unencrypted GridFTP is closest to disk throughput for large files, while that of the rest (which use encryption) is much lower. This is because, the main bottleneck happens to be the CPU-intensive encryption and digest algorithms, that are provided by the cipher suite. Note that we have not attempted to change the cipher suites for these tests - changing the encryption routines could possibly improve the overall performance.

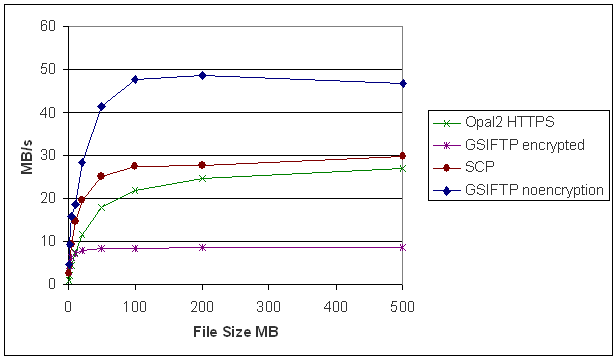

Figure 6. Data throughputs with encryption

B. Cloud Computing Environment

We have created an Amazon Machine Image (AMI) for Opal2 based on Rocks (v5), which wraps several NBCR applications such as PDB2PQR and MEME. Users can install a virtual machine on Amazon EC2 using our AMI, and have their own instance of the NBCR software stack, including the Opal2 Web services wrappers.

To test the feasibility of hosting scientific software on cloud computing resources, we performed a set of tests on Amazon EC2. The Opal2 server was installed on an Amazon "small instance", which provides 1.7GB of RAM with a single EC2 compute unit, where one compute unit provides the equivalent of a 1.0-1.2 GHz 32-bit 2007 Opteron or 2007 Xeon processor. This type of instance is the cheapest and the smallest, and can be used for around $0.10 per hour. The clients were running on the same test cluster as above, at the University of California, San Diego.

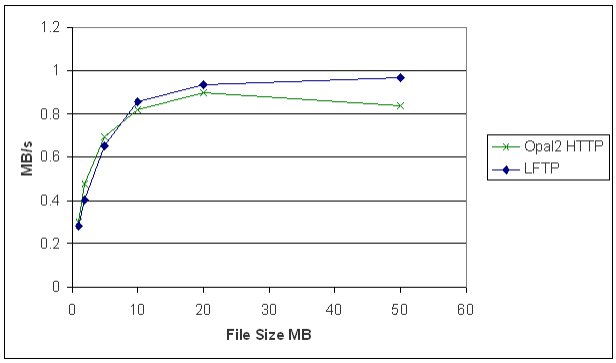

Figure 7. Data throughput comparison on EC2

Figure 7 shows our performance results on our test environment. The disk throughput on the small instance was observed to be close to 45 MB/s. Since this is significantly greater than the observed network bandwidth, the disk throughput is not the bottleneck anymore. Using LFTP, we observed a bandwidth of just under 1 MB/s. This can be attributed to the TCP buffer size, and the fact that the route to the Amazon data centers required about 20 hops. Using 4 concurrent streams, we could achieve an aggregate bandwidth of around 2.8MB/s, and around 4.9 MB/s with 8 data streams (data not shown in the figure). With Opal2 and MIME attachments over plain HTTP, our performance was practically similar to LFTP without multiple data streams.

Our preliminary testing with Opal2 services running over HTTPS (not shown) proved that similar throughput can be expected as the other tests shown in Figure 7. This is because in the context of Amazon EC2, the bottleneck is the network and not the CPU-intensive encryption and digest algorithms. Since the network is the bottleneck, we also concluded that there would be no benefit in testing the Opal2 overhead on the larger EC2 instances, which provide better CPU and disk performance. However, we note that the performance of the scientific applications themselves may be significantly improved on the larger instances.

Internal data transfer performance within EC2 infrastructure has been already discussed in other literature (e.g. [28], [29]), and it is possible to expect close to 60 MB/s of raw TCP throughput between instances running within the same EC2 availability zone. In [29], the authors underline that CPU-bounded serial applications are affected minimally (10-15%) as compared to a typical HPC system; however, tightly coupled MPI applications may degrade by as much as 800%. This implies that not all scientific applications may be ported on to the cloud. Serial or loosely coupled parallel applications (e.g. parameter sweeps) will run without major performance degradation on EC2, while tightly-coupled parallel scientific applications such as NAMD (see Section IV-C), where there is a need for a lot of data exchange between various parallel threads, will perform poorly. However, certain parallel scientific algorithms could be run on cloud resources if they are rewritten using paradigms such as MapReduce [24], which exploits data locality and minimizes data exchange.

C. Scientific Applications on the Grid

NAMD (NAnoscale Molecular Dynamics) [30] is a parallel molecular dynamics code designed for high-performance simulation of large bio-molecular systems. NAMD can scale up to thousands of CPUs, and typically needs several input files containing the description of the molecular structure and the force field that will be used during for the simulation.

The input files to NAMD are typically several hundred megabytes. Using an earlier version of Opal that relied on Base64 encoded binary inputs is not acceptable, as seen from the performance results in Section IV. However, the availability of MIME attachments and its improved performance enable the use of Opal2 for managing NAMD jobs. Furthermore, a maximum bandwidth of 60 MB/s between the EC2 instances is unacceptable for a high performance parallel code such as NAMD. This necessitates the use of supercomputing facilities with low latency and high

performance Myrinet or Infiniband connectivity, such as the ones provided by the TeraGrid. Hence, we use the 868-node NCSA TeraGrid cluster called Mercury to run the NAMD simulations.

We use the factory pattern and the plug-in architecture described in Section III to implement a new Globus job manager for remote job submissions to the TeraGrid. In this model, the Opal2 server resides on the NBCR cluster at UCSD, while the NAMD jobs are run on remote TeraGrid machines. While this requires an additional hop because of the need to stage files from the Opal2 server and the TeraGrid machine via GridFTP, it is necessary because TeraGrid machines do not allow hosting long running Web services. However, since the NAMD jobs are long-running (order of days), the data transfer time is negligible with respect to the overall execution time.

NAMD and other scientific applications such as PDB2PQR are used in the Relaxed Complex Scheme (RCS) to develop novel inhibitors for infectious diseases [31]. A computer aided drug discovery pipeline based on the RCS method, which uses Opal2-based remote resources for computationally intensive tasks and local Vision-based [17] 3D rendering capabilities for visualization of results, has been developed. Using this setup, undergraduate students having limited knowledge of HPC systems are able to perform experiments using RCS and remote Grid resources.

V. Related Work

There have been several efforts to provide Web service wrappers for scientific applications. In particular, the Generic Factory Service (GFac) [32] is a toolkit developed at Indiana University. The goals of GFac and Opal2 are quite similar. They use different approaches and software stacks to provide similar functionality. GFac provides automatic service generation using an XML-based application description language (called "serviceMap"), using the XSUL SOAP libraries [33] for Web services support. Instead of creating source code for the stubs, GFac 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. XSUL is a high-performance SOAP implementation, but home-grown and non-standard in the Grid community.

SoapLab [34] is another toolkit, that an application description language called ACD to provide automatic Web service wrappers. When we began our work on Opal2, SoapLab was built on a CORBA-based back-end, which is fairly non-standard in the Grid community, thus introducing a further entrance barrier for new adopters. Version 2 of SoapLab released recently does not rely on CORBA, and we plan on evaluating it in the future.

GFac and SoapLab also differ from Opal2 in the use of different auto-generated WSDLs for every deployed application. This results in different stubs on the client side for every application. For greater simplicity, we have adopted a generic WSDL API for all Opal2 services so that the same client stubs may be used to access all Opal2 services. Command-line arguments can still be described using optional XML metadata in Opal2, thus providing additional expressibility and type-validation, if need be.

The Otho toolkit [35] developed at the University of Innsbruck, provides semi-automatic transformation of existing scientific applications deployed on Grid resources into Grid application services. The generated output are service source codes that are either automatically built and packaged into a ready-to-deploy services or taken by developers for manual refinement. However, the software is fairly prototypical, and not available for public download.

VI. Conclusions

In this paper, we presented Opal2 as a toolkit for scientific Software as a Service (SaaS), which enables the wrapping of computational codes as Web services on Grid and cloud computing resources. We also described packaging of the Opal2 toolkit inside virtual machines and as Rocks rolls, thus demonstrating one scenario of the Service as Software (SaS) paradigm. We provided a detailed analysis of the Opal2 architecture and implementation, and analyzed its performance on traditional Grid resources, and cloud computing environments (using Amazon's EC2), and discussed how Opal2 is being used to support complex computational pipelines in drug discovery research. The use of service orientation enables the creation of higher-level scientific services and repeatable solutions for Grid computing that can be used by multiple clients in complex scientific pipelines. The packaging of scientific software as deployable units enables customization and use by a broad community of scientific users and developers.

In the near future, several improvements will be made to the Opal2 framework. In particular, we plan on adding support for more resource providers, e.g. for the use of other meta-schedulers such as Condor [19]. We plan on adding support for other application types, e.g. array and MapReduce [24] jobs. We also plan on a public release of the Rocks Rolls and virtual machine images of the NBCR software stack (including the Opal2 services), and Python and Perl client side libraries. Looking further ahead, we anticipate better support of mature community standards for Web services and the semantic Web. Complex high level services require better semantic annotation to ensure automated service discovery, provenance, safe data typing, and input/output validation. In particular, we are investigating the recent W3C recommendation of Semantic Annotations for WSDL (SAWSDL), which has been applied to bioinformatics semantic Web services [36].

Our work on Opal2 has been supported by the National Institutes of Health (NIH) through a National Center for

Research Resources program grant (P41RR08605).

REFERENCES

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

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

[3] W. Allcock, et al, "The Globus Striped GridFTP Framework and Server," in *Super Computing 2005 (SC05)*, 2005.

[4] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," in *Intl. Journal of Supercomputer Applications*, vol. 15(3), 2001.

[5] "Amazon Elastic Comp. Cloud," http://aws.amazon.com/ec2/.

[6] S. Krishnan, et al, "Opal: Simple Web Services Wrappers for Scientific Applications," in *IEEE Intl. Conf. on Web Services (ICWS)*, 2006.

[7] S. Krishnan, et al, "An End-to-end Web Services-based Infrastructure for Biomedical Applications," in *6th IEEE/ACM International Workshop on Grid Computing*, 2005.

[8] P. Barham, et al, "Xen and the Art of Virtualization," in *Proc. Symposium on Operating Systems Principles*, 2003.

[9] P. Papadopoulos, M. Katz, and G. Bruno, "NPACI Rocks: Tools and Techniques for Easily Deploying Manageable Linux Clusters," in *Concurrency and Computation: Practice and Experience Special Issue*, 2001.

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

[11] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," 1997.

[12] K. Czajkowski, et al, "A Resource Management Architecture for Metacomputing Systems," in *IPPS/SPDP 98, Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

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

[14] "The MEME/MAST System - Motif Discovery and Search," http://meme.sdsc.edu/meme/.

[15] "PDB2PQR: An Automated Pipeline for the Setup, Execution, and Analysis of Poisson-Boltzmann Electrostatics Calculations," http://pdb2pqr.sourceforge.net/.

[16] I. Altintas, et al, "Kepler: An Extensible System for Design and Execution of Scientific Workflows," in *16th International Conference on Scientific and Statistical Database Management (SSDBM'04)*, 2004.

[17] M. Sanner, "A Component-based Software Environment for Visualizing Large Macromolecular Assemblies," *Structure*, vol. 13, pp. 447–462, 2005.

[18] W. Xiaohui, D. Zhaohui, Y. Shutao, H. Chang, and L. Huizhen, "CSF4: A WSRF compliant meta-scheduler," *The 2006 World Congress in Computer Science, Computer Engg, and Applied Computing, GCA*, vol. 6, pp. 61–67, 2006.

[19] J. Basney, M. Livny, and T. Tannenbaum, "High Throughput Computing with Condor," in *HPCU news*, vol. 1(2), 1997.

[20] "SOAP Messages with Attachments," http://www.w3.org/TR/SOAP-attachments.

[21] C. Bauer and G. King, *Hibernate in Action*. Manning Pub., 2004.

[22] "Distributed Resource Management Application API (DRMAA)," http://www.drmaa.org/.

[23] "Apache Hadoop," http://hadoop.apache.org/.

[24] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[25] L. Clementi, et al, "Providing Dynamic Virtualized Access to Grid Resources via the Web 2.0 Paradigm," in *International Workshop on Grid Computing Environments*, 2007.

[26] "The TeraGrid Project," http://www.teragrid.org/.

[27] "LFTP: A Command-line FTP Client," http://lftp.yar.ru/.

[28] D. Nurmi, et al, "Eucalyptus: A Technical Report on an Elastic Utility Computing Architecture Linking Your Programs to Useful Systems," Aug 2008, UCSB Computer Science Technical Report Number 2008-10.

[29] E. Walker, "Benchmarking Amazon EC2 for High-Performance Scientific Computing," *;login: The USENIX Magazine*, vol. 33, no. 5, Oct 2008.

[30] "NAMD: Scalable Molecular Dynamics," http://www.ks.uiuc.edu/Research/namd/.

[31] L.S. Cheng, et al, "Ensemble-based virtual screening reveals potential novel antiviral compounds for avian influenza neuraminidase," *J. Med. Chem.*, vol. 51, pp. 3878–3894, 2008.

[32] G. Kandaswamy, et al, "Building Web Services For Scientific Grid Applications," in *IBM Journal of Research and Development*, 2005.

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

[34] "SoapLab Web Services," http://www.ebi.ac.uk/soaplab/.

[35] J. Hofer and T. Fahringer, "The Otho Toolkit–Synthesizing tailor-made scientific grid application wrapper services," *Multiagent and Grid Systems*, vol. 3, no. 3, pp. 281–298, 2007.

[36] P. M. K. Gordon and C. W. Sensen., "Creating Bioinformatics Semantic Web Services from Existing Web Services: A Real-World Application of SAWSDL," in *Int. Conf. Web Service (ICWS)*, 2008.