Designing Service-Oriented Tools for HPC Account Management and Reporting

Adam G. Carlyle, Robert D. French, William A. Renaud
National Center for Computational Sciences (NCCS)
Oak Ridge National Laboratory (ORNL)
Oak Ridge, Tennessee, USA
{carlyleag, frenchrd, brenaud}@ornl.gov

***Abstract*—The User Assistance Group at the National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory (ORNL) maintains detailed records and auxiliary data for thousands of HPC user accounts every year. These data are used across every part of the center in system administration scripts, written reports, and end-user communications.**

Record storage tools in use today evolved in an ad-hoc fashion as the center's needs changed; now they are at risk of becoming inflexible and unmaintainable. They also exhibit some scalability issues both with respect to computational time, and—perhaps more importantly—with respect to staff effort and to the triage of new development tasks.

The solutions needed to address these issues must be strongly *service-oriented*. At the center of the service-oriented approach lies the concept of isolation. Isolated services/applications contain their own codebase and business logic, use their own data store, and have their own support teams. The underlying data is accepted and served by the application through a well-defined, versioned API and associated language-specific API client libraries. Incoming data is validated at the API level. Direct writes to the data store from outside of the service are eliminated. Code maintainability is improved since support teams can make changes to the internals of any given service without affecting the service API, and therefore, without affecting other people or applications that interact with it.

This paper details recent efforts at NCCS to redesign the center's two primary record-management solutions into service-oriented applications capable of meeting these future challenges of scalability and maintainability.

***Keywords*-service-oriented; SOA; account management; RATS; ORNL; NOAA;**

I. Introduction

The User Assistance Group at the National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory (ORNL) maintains detailed records for thousands of HPC user accounts every year. Data pertaining to users, projects, allocation levels, completed jobs, etc., are maintained for use in system administration scripts and for reporting purposes. In addition, data pertaining to the state of HPC systems, the state of individual jobs, center announcements, software availability, etc., are stored for reporting purposes as well as for wide dissemination to the center's user base via the web and a host of other communication channels. These data are the cornerstone of the center's day-to-day operations.

There are two primary applications we use to manage these data. The NCCS's *Resource Allocation Tracking System* (RATS) is the primary record-management application for the center. It began as a simple database and has evolved into a complex application as the center has grown. It stores hundreds of millions of business-sensitive records on user accounts, projects, and HPC jobs that the center needs for day-to-day operations. *StatusCast* is a recently developed service-oriented application that will be used to store data that must be disseminated widely and openly to the center's user base.

RATS evolved in an ad-hoc fashion as the center's needs changed; now it is at risk of becoming inflexible and unmaintainable. Getting the right data in the right format typically requires considerable person-hours, particularly for one-off report requests. Getting the right data *quickly* and *efficiently* is considerably more challenging. It also exhibits some scalability issues both with respect to computational effort, and—perhaps more importantly—with respect to staff effort and to the triage of new development tasks.

The demands placed on our center mandate record-management tools capable of servicing many hundreds of different types of record requests from different hosts and/or individuals. Stored records must be searchable and retrievable using both automated techniques and interactive ones. The resulting datasets must be easily parsable by a number of different high-level languages, yet also easily adapted for inclusion into graphs and tables for written reports. Although challenging to develop, data storage and retrieval solutions that can deliver such varied datasets quickly and easily have the potential to save the center many thousands of person-hours in effort annually. Any such solutions would need to be strongly *service-oriented.*

The remainder of this paper details the recent efforts at NCCS to redesign RATS into a modern service capable of meeting future challenges, the ground-up design of *StatusCast*, the first service within our new SOA framework, and the creation of a flexible reporting service.

II. Tenets of Service-Orientation

Service-orientation is a set of guiding engineering principles for the design of interconnected computer software components, called *services*. Services are standalone applications designed to handle a small subset of related operational needs for the organization. Service-oriented applications, in turn, comprise many individual services with the end goal of automating the business logic of an organization. Applications designed in this way are said to have adopted a service-oriented architecture (SOA).

Service-orientation gained popularity in the mid-2000s as major tech companies[1][2] adopted service-orientation and its promise of increased return-on-investment and operational agility. A number of web service protocols and best-practices emerged alongside the paradigm, and many such as REST[3], SOAP[4], and JSON-RPC[5] remain popular in service-oriented applications today.

Don Box's work on Microsoft's Indigo project was among the first industry projects to feature a well-defined functional implementation of a SOA.[6] Project Indigo later became Windows Communication Foundation; a set of APIs in the .NET Framework for building service-oriented applications. His writeup on project Indigo lays out the following four tenets of service-oriented development:

1) Service boundaries are explicit
2) Services are autonomous
3) Services share schema and contract, not class
4) Service compatibility is determined based on policy

Each of these tenets has implications for us as we wrestle with the business operations of our HPC center.

A. Service Boundaries are Explicit

SOAs are based on a model of isolated services communicating via explicit message passing (a paradigm with which we in the HPC community should be very comfortable). Each service aims to keep its "surface area" as small as possible by carefully controlling the communication channels it exposes between itself and other services.

This concept of isolation results in the most immediate benefits to our center compared with our historical approaches to information management. Further development of our current information management tools is most primarily constrained by the lack of boundaries between (what could be) individual services.

B. Services are Autonomous

Individual services within a service-oriented application are developed and deployed as self-sustaining applications. They have individual data stores, individual support infrastructure, and often, individual developers and support teams. Services expect that their eventual consumers can and will fail, sometimes silently. They also assume all incoming message data may be malformed or transmitted for malicious purposes.

C. Services Share Schema and Contract, Not Class

The terms *schema* and *contract* in SOA refer to the rules around how data will be transmitted to the service and in what form. In practice, the contract is often enforced through authentication mechanisms built into the service (e.g. API tokens) and the schema through input validations and made public through a published application programming interface (API). In contrast to object-oriented design which relies on common data structures, service-oriented design relies on consistent APIs for each service. Services expect that their consumers will be reliant on the service for long periods of time and across many different hosts and/or physical locations. This assumption mandates a relatively stable API over time. Underlying changes to the internal logic of the service are managed internally so as to not propagate up to the API level. When an underlying change to the service cannot be accommodated by the current API, the API is often versioned, and the new service remains backwards-compatible with the old API version for some amount of time during which consuming services can be updated.

D. *Service Compatibility is Determined by Policy*

In addition to schema and contract enforcement, SOA imposes the requirement of a *policy* upon individual services and the application as a whole. Each service makes available a machine-readable, explicit policy statement that lists capabilities of the service and requirements to use it. An SOA application may also impose a policy for the minimum level and type of testing each service should implement to assure reliability. Often an SOA registry, itself a service, provides a catalog of information about the available services in the SOA implementation.

This particular tenet is of the least concern to our center at the moment, due to the very small number of services needed to carry out our design goals. We see it becoming more important in the future as our service catalog grows.

III. Example of a Non-service-oriented Application

RATS, in its current form, is a database-backed web application. NCCS staff authenticates to the application through a standard web browser, manipulates data, and logs out when finished. It has no well-defined API, and as such, programmatic input and output of data to/from the application are made directly through SQL statements running via an array of scripts that authenticate directly to the underlying RATS database. No real effort has been made to limit connectivity or channel it though any standardized SOA service contract nor SOA service schema. It is, by definition, not currently service-oriented.

The limitations of the current application give rise to a number of inefficiencies, all which have contributed to the desire to redesign the application:

- The application cannot enforce a complex service schema. That is, except for foreign key constraints and non-NULL validations at the database level, it cannot prevent a script from writing malformed or incomplete records or sets of records, nor can it issue useful error messages in the event of malformed input.
- The application cannot adequately log input/output. Any script with valid read/write database user credentials can modify the database with little record of exactly who (person) made the request, and why.
- The application is difficult to shut down for maintenance. Since there is no single point of access for the application, there is no definitive communication channel for all of the various scripts and script maintainers who have a vested interest in the application's availability. Currently, one can only take the application offline and hope that the array of scripts trying to access the application fail gracefully in its absence, and that all staff stakeholders know about the outage.
- The application cannot selectively expose datasets. Except for defining hundreds of database users with very specific grants which can run specific queries (cumbersome), the application cannot easily serve selected subsets of data while restricting others.

Each of these shortcomings has practical impact on routine tasks we wish to carry out during the day-to-day operation of the center. Take, for example, the last enumerated shortcoming above as it pertains to the all-too-common HPC center task of communicating a project's allocation utilization metrics via multiple channels: on the command line of a computational resource, on an access-controlled website for end-users, and within an access-controlled website for center management.

Of course, utilization metrics change moment-to-moment as a project's jobs complete. Within our current application, the steps required to complete the task above are as follows:

1) Jobs complete and are written out to log files by the computational resource's job scheduler/resource manager.
2) Every Q minutes, cron executes script A which parses the log file and writes records directly to the application database.
3) Every R minutes, cron executes script B which connects directly to the database, aggregates records related to project utilization, and writes the results to a flat file on a staging file system.
4) Every S minutes, cron executes script C which rsyncs data on the staging file system to various

website docroots. The staging filesystem is not mounted on web servers for security safeguards, so the files must be synced over.
5) Data on the staging file system and/or a website docroot is accessed by a command line utiliy or browser request, as appropriate.

IV. Problems With Non-service-oriented Applications

The limitations of the non-service oriented approach to RATS become apparent upon examination of this process:

- The process takes up to ($Q + R + S$) minutes to complete.
- The process has many points of failure (multiple cron jobs, multiple scripts, multiple filesystems)
- The scripts must each separately enforce the proper application data schema.
- The process is difficult to document and understand since it lacks well-defined boundaries for what components it contains.

Figure 1 shows the components involved in data management for the process, and are typical for a non-service-oriented application.

The components involved are managed by many individuals in the center, making coordination difficult and development tasks prone to error due to unknown dependencies across components. Applications in this state are difficult to develop, and lacking significant staff effort, begin to erode as the supporting software packages (database technologies, server software stacks) evolve and move forward. Required changes to the application are also difficult to test, since there is no real constraint on how the data is being consumed and used.

V. A Service-oriented Redesign

With these limitations in mind, we in NCCS User Assistance set out to redesign a separate, smaller application to see if we could increase operational efficiency using a SOA approach.

The NCCS's Auto-Downtime System has been a primary source of system status information for several years. This system, previously described as implemented within the National Climate-Computing Research Center (NCRC) project[7], consists of a suite of Perl scripts that parse Nagios logs to determine the status of a computational resource. This status is then written to files that are parsed and displayed on user-facing websites in a process very similar to that described previously in our example use case. Basic checks are put in place to minimize false positives. In addition to the functionally previously described, the system provides active notifications to users via email and Twitter[8].

The existing status system, while helpful to users, is not without several shortcomings. It provides only current status; however, since it uses a database for back-end storage it could potentially provide prior status change information. Unfortunately, any such changes would have to be generated by someone knowledgeable of the database schema as having a script or program directly access the database is the only way to pull information. The existing database is also lacking in the level of detail it can provide about previous outages and does not provide the capability for entering planned outages. It shows the existence of an outage, not the reason for one. It too exhibits many of the hallmarks and limitations of non-service-oriented design patterns.

A. *StatusCast*

A new service-oriented initiative, called *StatusCast*, aims to overcome these shortcomings. Similar to the existing system, StatusCast will rely on a back-end database; a redesigned version of the database used by the Auto-Downtime System. In addition to providing similar features (storage of each down/up state encountered for a system, logging the notifications about those state changes, etc) it also provides the capability for storing detailed entries for each downtime. Additionally, it provides for entry of system announcements and for the storage of other details of a system's status, such as job load.

SOA has been adopted from the ground up with StatusCast. StatusCast is, at its core, a service API through which utilities and people interact with the back-end database. The automated status script, system administrators entering details about an upcoming (or previous) downtime, staff members entering announcements, and utilities storing other pieces of status information will utilize the StatusCast API to add (or edit) data in the database. Furthermore, consumers of that data (websites and myriad other potential utilities) are provided with a lightweight, easy-to-use mechanism by which they can access data.

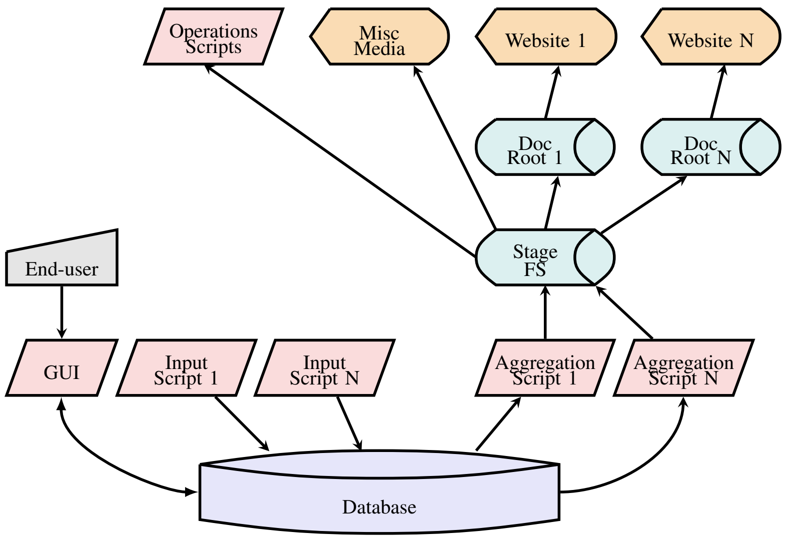

Figure 1. Data management process for a typical non-service-oriented application. Note the reliance on multiple staging areas on physical disk.

StatusCast uses hypertext transfer protocol for data transmission within the architectural constraints laid out by representational state transfer (REST). The service API backend is implemented in Ruby with the popular Ruby on Rails framework. It exposes a few hundred URLs for interacting with the service and the RESTful resources it tracks. StatusCast also provides a series of language-specific service libraries to ease adoption of the service by NCCS staff. Libraries provide a largely consistent development interface by centering library functions around the standard actions on persisted storage (search, create, read, update, delete). It provides a GUI for end-users based around the same consistent end-user experience, and enforces access controls via two-factor authentication for the GUI and via API token for the API.

Through this approach, we gain many advantages over the previous design:

- All requests, both through the GUI and API, are validated for service schema compatibility at the application layer. Validations can be, and are, very complex. Malformed requests can trigger useful error messages for the end-user.
- All requests, both through the GUI and API, are adequately logged, and in a completely customizable format, easing integration with data indexing engines like Splunk.
- The application can be shut down for maintenance easily. When the database is down, the GUI/API layer can say so. When the GUI/API layer is down, the wrapper libraries can say so.
- The application can selectively expose datasets in any form. Only the internal functions exposed through URLs can be accessed at all from outside the service, and this is completely customizable.

Recall our earlier use case—communicating a project's allocation utilization metrics via multiple channels. Within a service-oriented design such as employed by StatusCast, the steps required to complete the task could be as follows:

1) The resource manager/job scheduler epilogue calls a python script that uses the service's Python wrapper library to write a job resource record to the service.
2) Websites and command line utilities use PHP, Javascript, and Python wrapper libraries as appropriate to access utilization data within the service, on-demand.

Figure 2 shows the components involved in data management for the process, as typical for a service within a SOA.

VI. Flexibility Through Service Plugins

As discussed in the previous section, StatusCast, in and of itself, is neither the presenter of data to interested parties nor the source of information from monitoring software, system administrators, user support staff, etc. Rather, it serves as a information

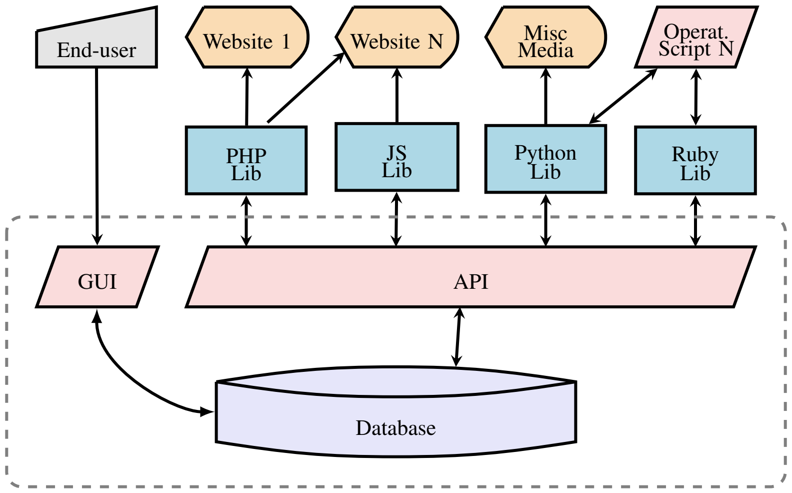

Figure 2. Data management process for a typical service within a SOA. Data to/from the service is posted/requested in real-time via a well-defined API.

broker service by which user-facing and center-facing processes access the service's central data store. We recognize that not every producer process nor consumer process of the StatusCast service will have the need to (nor the resources to) be implemented as a fully SOA-compliant service itself. To add more flexibility to the SOA application as a whole, we define the concept of a *service plugin* to describe any "outside" producers or consumers of the data stored within an application service that are not fully SOA-compliant services themselves. Service plugins, then, are typically small utilities or scripts developed by center staff (and potentially others) that make use of one or more services within the SOA application at large. The StatusCast service provides an API call that can be used to store service plugin parameters for those plugins that require limited amounts of plugin-related data to be persisted (e.g. time of last run).

Prior to recent redesign efforts, staff developers often needed access to information stored within our non-service-oriented applications like RATS or the Auto-Downtime System. Such tools, which we'll call *application plugins*, were difficult to develop at the center in the past. Coordination issues were inevitable when multiple users wished to provide and/or access data within the application. Without sufficiently service-oriented infrastructure in place, those developers needed to know the low-level details of the application's back-end database to effectively use it. Keeping developers up-to-date on current database design (since the schema is subject to change) was a huge administrative challenge for the application maintainers as well as the developers. With no way of isolating modifications to the database layer, even minor changes in the database schema would cause all remotely-developed tools to fail.

Such tools also needed to run on a system with special access the database, and needed one-off authentication credentials with which to access the database. As a result, application plugins were very limited in where they could run due to the access control concerns. As hosts are upgraded and decommissioned, application plugins were shuffled around and subsequently failed due to firewall rules and host-specific database grants. Clearly, this design was suboptimal from both administrative and security perspectives.

Finally, with no centralized service/plugin registry in place to communicate the center's current capabilities, development of application plugins was disparate and completely individualized. This resulted in several scripts, databases, and support systems that all handled similar data. At one point, no less than three different databases all held different pieces of downtime information.

A. Interacting with a Service API via Plugins

With a service-oriented approach StatusCast overcomes these limitations and permits centralization of these data. The hypertext protocol approach permits plugins to run on any system that can access the URL's domain (typically limited to other systems on the center's network). Changes to the database schema need not be exposed at the API layer. This model of operation also isolates the service plugin from database credentials, which is beneficial from both administrative (no need to create and distribute specific credentials to a group of developers) and security perspective.

Plugin developers are encouraged to always use the provided wrapper libraries to access the service API. The hypertext protocol approach has its advantages but has the disadvantage of requiring URL query strings that can quickly become unwieldy and therefore error-prone. The wrapper libraries handle the generation of the API URL in a controlled manner across multiple programming languages. This allows service plugins to retain a friendly, function-oriented and/or object-oriented approach to gathering and updating information.

Some service plugins will be developed by center staff, but the API will certainly accommodate a larger group of developers. These developers will have the tools to readily access only data they need, while requiring minimal knowledge of service back-end. The API can be distributed to a larger corps of developers to permit them to generate service plugins that provide highly customized data if they find that, for example, the available service-provided data formats do not meet their needs.

While the most common view of a plugin is a consumer of data—that is, a utility to read data from the service's data store and display it—it is important to remember that plugins can also appear on the producer side. The scripts that provide system status information, the automated mechanisms that permit system administrators to enter downtimes, and the utilities that support staff will use to enter announcements in an automated fashion are also considered service plugins. These producer plugins interact with the service identically to consumer plugins—they use the service API.

B. Future Direction of Plugin Development

Initial development of service plugins have been carried out solely by divisional staff. Plugins in development for the initial release of StatusCast include an automated status plugin to update and control the recorded state of a computational resource, a system queue utilization plugin to track snapshots of a computational resource's job queue, plugins for command-line creation of downtime and announcement entries, and a plugin that generates a user status page on NCCS websites. As StatusCast matures, its development model can be extended to permit service plugin development by other divisions within the lab. A small amount of coordination may be needed with center staff, but this is anticipated to be minimal. Ultimately, StatusCast will permit center staff to work more efficiently in presenting status information to users of our center.

VII. Flexible Reporting as a Service

The NCCS encounters new reporting needs on an almost daily basis. These reports are requested by varying audiences such as center management, our Resource Utilization Council, our program sponsors at the Department of Energy, our users, and our outreach staff. The nature of business intelligence is that many of the requests we see are similar to prior requests, but distinct enough that attempts to build a generalized software solution are often inadequate or inflexible. As such, the reporting tools that have evolved for use by our center still require many person-hours of manual effort.

The current process follows a general framework which attempts to leverage the similar nature of the reports: First, the report developer will construct a set of SQL queries to retrieve appropriate records from RATS, our central accounting database. The developer usually begins this task by referencing existing report scripts in order to ensure that similar queries are carried out in a uniform manner. For example, when reporting on the utilization of Titan, both job records and downtime records must be aggregated, with the caveat that planned outages are separated from unplanned outages. Expressing this constraint in SQL can be non-trivial, especially when considering edge cases such as jobs that span calendar borders (i.e., should a job be counted in the month when it starts or the month when it finishes?).

Second, the report developer will craft a post-processing script (usually in PHP) to drive the SQL queries and transform their results into a single web page. These scripts are executed daily as cron tasks, and the resulting HTML pages are staged to an appropriate web server which is accessible by the report's audience. Again, the similarity of the reports lends itself to developing new report scripts by the same "copy-and-tweak" method described above, but it is still difficult to generalize these scripts to handle more than a few types of request.

Lastly, the report developer must verify that the figures being produced are accurate, and that they answer the same question which the report audience has asked. This too can be a non-trivial, as the report audience will likely be familiar with neither the schema of our accounting database nor (more importantly) the system processes which supply records to that database. Understanding the context in which the data was created is vital to verifying that report information is dependable.

This process is time consuming, prone to error, and produces reports in only one format. Ideally, report data could be produced in different formats (spreadsheets, slideshows, etc.) to support the varying needs of our audience, and reports should be programmatically queryable.

A. *Unsuitability of Existing Solutions*

Many excellent business intelligence tools exist (notably On-line Analytics Processing or "OLAP"-style solutions), and many of them come with GUI tools for visual report development. After reviewing these tools and their capabilities, we were concerned with the following: Many of these tools are enterprise-class solutions, and anticipate a staff dedicated to data modeling and report development. At the NCCS, reports are developed by the User Assistance group, who are HPC experts and largely not data modeling professionals.

Using the GUI tools to allow our stakeholders to develop their own reports was initially promising, but after further consideration is seemed likely that users could easily misinterpret the data schema and unknowingly craft misleading reports.

Further, the reporting needs we face do not require reports to be generated more frequently than every 24 hours. This is an advantage, but as many OLAP tools are designed for "on-line" or "real-time" analytics processing; selecting one of these solutions would involve signifcant overhead in infrastructure which would go unused.

B. *Towards a Service-Oriented Solution*

The middle-ground between a full-stack OLAP solution and a manually maintained set of one-off scripts is an infrastructure that aides the report developer in minimizing effort. Our aim was that all database interaction would be centralized, with common queries broken down into small, self-contained, and reasonably-generalized actions which are provided by the reporting service. Instead of copying code from old scripts, new report scripts could be developed with more confidence by leveraging the queries provided by the reporting service, and service plugin developers can extend the reporting service with new methods as needed.

Using as an example the utilization query described above, the reporting service would provide the capability to answer the reasonably-general question "What fraction of system X was utilized from time Y until time Z?". Report scripts can then be developed which call this method for all appropriate utilization information, thus assuring consistency across reports.

In order for report scripts to consume the output of reporting service API calls, the service schema of this output must be published alongside the method calls themselves, so that report developers can make accurate use of existing queries.

C. *Future Direction*

Moving forward, we see an opportunity to implement robust, flexible reporting as a service within the larger SOA framework of our center. We believe that many simple reports could be represented as linear lists of reporting service queries, together with appropriate input parameters and guidance for displaying output. In many cases, new reports could be developed using simple (likely JSON-formatted) configuration files instead of having to be developed in a scripting language. This would allow for software to programmatically define new reports, and could support the creation of a very limited ad-hoc reporting environment.

VIII. Conclusion

Within User Assistance at NCCS, we have taken the initial steps to move our information management tools to a SOA design, and future projects will be designed from the ground up to fit within the SOA approach.

The potential benefits of service-oriented architectures are becoming widely understood across industries outside of HPC, and the operational efficiencies of the approach can certainly be leveraged within the business logic prevalent in our field. Open-source frameworks are available in all of the popular high-level scripting languages to aid in creation of service APIs and associated service GUIs. Open-source libraries for object-relational mapping make interacting with data stores like relational databases and key-value stores easier than ever before. The technology is relatively mature, having been adopted and expanded by many of the large tech firms over the past five to ten years.

The demand for HPC operational data is ever-changing and ever-growing; we feel that SOA will be an instrumental part of any successful strategy for maintaining control of information management processes in this dynamic environment.

References

[1] (2014) About aws. Amazon Web Services, Inc. [Online]. Available: http://aws.amazon.com/about-aws/

[2] L. Cherbakov, G. Galambos, R. Harishankar, S. Kalyana, and G. Rackham, "Impact of service orientation at the business level," *IBM Systems Journal*, vol. 44, no. 4, pp. 653–668, 2005.

[3] R. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, UC Irvine, 2000. [Online]. Available: http://www.ics.uci.edu/˜fielding/pubs/dissertation/rest_arch_style.htm

[4] (2014) Soap version 1.2 messaging framework. World Wide Web Consortium. [Online]. Available: http://www.w3.org/TR/soap12-part1/

[5] (2014) Json-rpc 2.0 specification. JSON-RPC Working Group. [Online]. Available: http://www.jsonrpc.org/specification

[6] D. Box. (2004) A guide to developing and running connected systems with indigo. Microsoft Corporation. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc164026.aspx

[7] A. Carlyle, R. Miller, D. Leverman, W. Renaud, and D. Maxwell, "Practical support solutions for a workflow-oriented cray environment," in *CUG Conference Proceedings*, Stuttgart, Germany, May 2012.

[8] OLCF. (2014) Communications to users. [Online]. Available: https://www.olcf.ornl.gov/kb_articles/communications-to-users