Apache Airavata: Design and Directions of a Science Gateway Framework

Marlon E. Pierce
Indiana University &
Apache Software Foundation
Bloomington, IN 47408 USA
marpierc@iu.edu

Suresh Marru
Indiana University &
Apache Software Foundation
Bloomington, IN 47408 USA
smarru@iu.edu

Lahiru Gunathilake
Indiana University &
Apache Software Foundation
Bloomington, IN 47408 USA

Thejaka Amila Kanewala
Indiana University &
Apache Software Foundation
Bloomington, IN 47408 USA

Raminder Singh
Indiana University &
Apache Software Foundation
Bloomington, IN 47408 USA

Saminda Wijeratne
Indiana University &
Apache Software Foundation
Bloomington, IN 47408 USA

Chathuri Wimalasena
Indiana University &
Apache Software Foundation
Bloomington, IN 47408 USA

Chathura Herath
Knight Capital Group &
Apache Software Foundation
New York, NY 10174 USA

Eran Chinthaka
Work Day Inc. &
Apache Software Foundation
San Francisco, CA, USA

Chris Mattmann
NASA JPL &
Apache Software Foundation
Pasadena, CA ,USA

Aleksander Slominski
IBM Research &
Apache Software Foundation
New York, NY, USA

Patanachai Tangchaisin
Wize Commerce &
Apache Software Foundation
San Mateo, CA USA

***Abstract*— This paper provides an overview of the Apache Airavata software system for science gateways. Gateways use Airavata to manage application and workflow executions on a range of backend resources (grids, computing clouds, and local clusters). Airavata's design goal is to provide component abstractions for major tasks required to provide gateway application management. Components are not directly accessed but are instead exposed through a client Application Programming Interface. This design allows gateway developers to take full advantage of Airavata's capabilities, and Airavata developers (including those interested in middleware research) to modify Airavata's implementations and behavior. This is particularly important as Airavata evolves to become a scalable, elastic "platform as a service" for science gateways. We illustrate the capabilities of Airavata through the discussion of usage vignettes. As an Apache Software Foundation project, Airavata's open community governance model is as important as its software base. We discuss how this works within Airavata and how it may be applicable to other distributed computing infrastructure and cyberinfrastructure efforts.**

Keywords—science gateways; distributed computing infrastructure; cyberinfrastructure; open source software

I. Introduction

This paper provides an overview of the Apache Airavata project, which provides science gateways with an abstraction layer for managing the execution and provenance of both single applications and workflows over many types of computational resources. Our purpose in this paper is to review the design of the software, discuss recent developments and innovations, and indicate where we expect future developments and research to take place within the Airavata framework. In particular, we want to push the project's design so that it can serve as the basis for a multi-tenanted service that can act as a "platform as a service" that gateways can use for outsourcing their task execution requirements. This paper presents for the first time an overview of the API Server and the Orchestrator and their relationships with other components. We also clarify changing roles of previous component. We provide an overview of how the project is managed using open governance mechanisms and how its collaborations with stakeholder gateways drives the project's evolution.

II. Airavata Software Design

A. Airavata, Gateways, and Cyberinfrastructure

Science gateways provide user interfaces and supporting services to scientific communities. As such, gateways have two important properties that contribute to Airavata's overall design. First, gateways use ad-hoc resource collections and act (in effect) as a federating over-layer over many different types of resources. A gateway may include access to multiple grids, to campus clusters, and to commercial clouds. The resources may not themselves be federated through standard-based Grid middleware in any way; the campus cluster, for example, may have no middleware at all other than SSH. Second, there are many ways to build successful user-facing services. Web technologies evolve rapidly and mobile devices are as important as traditional laptops and desktops for interacting with the Web. Below this level, however, science gateways need many common services such as application management and system metadata management.

Airavata's overall design goal is to address both these problems. It strives to be a general-purpose collection of services that can work with a wide range of programming languages and development frameworks on one end and with a wide range of resources and associated access mechanisms on the other.

B. System Components

Airavata has evolved from initial work in the Linked Environments for Scientific Discovery (LEAD) Science Gateway [1] and the generalization of the LEAD components

to other scientific disciplines through the Open Gateway Computing Environments projects [2]. The original architecture of the Airavata followed the "everything is a service" principle. This architecture has well served the needs of the complex system and played a key role in applying the software to multiple science gateways. In our earlier conceptions of Airavata, this approach facilitated the framework to not just be used as a turnkey solution, but allowed us to offer individual components of Airavata as standalone services.

However, as we gained experience from integrating and operating gateways through the full life cycle of development, integration and operation, we learned that the community support required by a "bag of services" approach does not scale for reasons elaborated in Section III. We need a way to scalably support both coherent development within an open source project and coherent usage by downstream clients. The former (component developers) need ways to experiment with different implementations without disturbing the architecture as a whole. The latter (gateway developers) are downstream users who typically do not need to do more than configure some Airavata options. Thus we have devoted significant effort to unifying the way gateways interact with Airavata into a single programming interface that mediates the interactions with Airavata's internal components, which are no longer directly exposed to client users. Airavata's internal components themselves are sharply defined through programming interfaces and interact through overall integration patterns. This is to support open development within the software system itself.

The latest generation of Airavata architecture packages the components based on functional areas, uniformity of access, and reliability needs into the following major components. This builds on and modifies the system described in [3]. We describe here only the separation of concerns and their associated components at the most abstract level. See Figure 1 for a summary. Each component is or will be the subject of more detailed descriptions in separate publications. All components are implemented in Java.

The **API Server** is the public face of Airavata and is based on Apache Thrift, which gives Airavata a strongly typed, programming language independent way of defining its interfaces. From the API definitions, we generate client packages in Java and PHP and are experimenting with other language bindings. Client gateways access Airavata through the API Server through a secure channel (SSL sockets or HTTPS). The API Server implements the Thrift-defined API. The API Server maps the client request into one or more calls to internal components, described next.

The **Airavata Registry** is the repository for all gateway metadata. This includes descriptions of applications and hosts, workflow templates, and workflow instances. The Registry does not manage actual data sets (such as files) and does not manage data movement. The current registry implementation is based on OpenJPA over a relational database backend, with both Derby and MySQL supported. Going forward, we identify several challenges with the Registry. First, its design is currently overloaded with too many distinct capabilities (application descriptions, workflows, user data, etc) that can be decoupled. We are designing multiple modules to efficiently demarcate application and workflow catalog, metadata catalog for recording application and workflow executions, monitoring database for archiving real-time information streams, and provenance data catalog for capturing the trace of executions. Second, the current implementation cannot directly store and retrieve Thrift-generated Java objects, creating an implementation friction at the Object-Relational Mapping level. We are investigating solutions to this problem, as has been discussed on the Airavata architecture mailing list.

The **Orchestrator** provides an abstract scheduling layer for individual and workflow submissions, which it does by managing interactions with the Application Factory and Workflow Interpreter. This component is also responsible, as a client to the Registry, for persistently storing all user requests. In the event of failure, the Orchestrator handles recovery of the system using Registry information. The Orchestrator is a new Airavata component, summarized here for the first time. The Orchestrator was introduced for the following reasons. First, Airavata needed a simpler way to provide an API to gateways who only wish to run single applications rather than workflows. In previous versions, Airavata required a definition of a workflow (stored in the Registry) for even the simplest applications. This made the definition of the API cumbersome. Second, we introduced the Orchestrator to consolidate functionality such as resource selection and integration with information services provided by different Grids. Thirdly, the Orchestrator's responsibility is to explicitly contain the per-submission decision-making making capabilities within Airavata for handling fault-tolerance strategies such as resubmission of failed jobs and recovery of failed services.

The **Workflow Interpreter** handles submissions that involve more than a single application. The Interpreter can submit and manage pipelines and graphs, but we also have examined its use in managing jobs that require iterative loops, conditionals, and human-in-the-loop executions [4][5].

The **Application Factory** (also known as GFAC) [6] manages the submission and monitoring of individual applications on remote resources. GFAC is designed around a Handler-Provider model that allows it to work with different grid and cloud middleware. In summary, providers are clients to different middleware and handlers supplement the provider submission with stage-in, stage-out, and related requests. Handlers also can be used to further customize submissions to specific resources that are not sufficiently abstracted by the generic provider. The Orchestrator directly interacts with GFAC for managing individual jobs; for workflows, the Interpreter invokes GFAC for each component step.

There has been substantial work on Grid job submission and Grid federation [16][17][18][19]. We view these as important foundations over which GFAC operates to accomplish its primary purpose of managing application submissions. As stated in Section IIA and illustrated in Section III, gateways may need to bridge over multiple cyberinfrastructure installations that are not otherwise federated. GFAC along with the Credential Store (below) provide this over-layer of application management. Current

GFAC plugins available in the source code include GRAM, BES/JSDL, SSH, LOCAL, HADOOP, GSISSH, and EC2.

The **Messaging System** [7] is a topic-based publish-subscribe system that is used to send system messages to multiple, interested components. It can also deliver messages to external clients that choose to be notified. This component incorporates an Airavata-wide information model that encompasses all system actions as well as used for provenance tracking.

The **Credential Store** [8] manages user credentials needed by a gateway to securely interact with remote distributed computing infrastructure. In particular, the Credential Store can be used to manage different credentials associated with different cyberinfrastructure systems.

Within these components, there are multiple areas for research and improvements. Many individual components have been re-implemented over time and continue to evolve. Furthermore, the specific sequence of interactions between components may also be rewired as we improve fault tolerance, redesign the system to allow it to be more elastic for deployments on clouds, and so on. In order to allow this evolution, and to move towards a "DevOps" approach when dealing with running Airavata instances, we have defined Component Programming Interfaces (CPIs) for each of the above components. These are currently Java interfaces, but we plan to translate these into Apache Thrift interface definitions in order to allow the components to be decoupled into separately running JVMs on different servers, while retaining the option of running everything in a single JVM. The future promise of Thrift-based CPIs is that they may be used to support non-Java component implementations. We note that CPIs are not exposed to end user gateways but are instead supersets of the API, also defined in Thrift's IDL.

The CPI approach allows us to dramatically change component implementations while not disturbing the rest of the system. For example, the Registry has undergone significant changes within the Airavata project, and we expect further evolution: its current implementation uses a relational database with an OpenJPA-based object-relational mapping layer, but we would like to experiment with triple stores, object stores, and NoSQL data stores such as Apache Cassandra. The messaging system also is a good candidate for upgrading. While it has been very stable and its codebase is the least touched of all the Airavata components, it is based on Web service notification and eventing standards that are not widely adopted. High quality open source messaging systems such as Apache Qpid and Apache Kafka are available, so Airavata should not need to maintain its own implementation, although it still needs its CPI and internal information model.

In addition to allowing Airavata to gracefully evolve, the CPI approach allows Airavata to support multiple implementations of a component that are needed for different uses and deployment scenarios. The Orchestrator is an important example. In a simple deployment scenario (such as basic testing), the Orchestrator needs only to communicate with a single Application Factory component instance, but in more complicated scenarios requiring load balancing and failover, we need to replace the "basic" Orchestrator CPI implementation with a "multi-threaded" implementation. These are standard object-oriented concepts extended to distributed systems.

C. Client and Component Interactions

The CPI provides a required internal layer of abstraction that allows components to change or have alternative implementations. The CPI is also useful because we continue to evolve the interactions between the components to improve Airavata's overall fault tolerance and elasticity without breaking the client API.

Figure 1 summarizes the current path through the components components for job submission and monitoring. Airavata supports two major roles from the calling gateway: end user scientists and gateway administrators. The steps for a gateway administrator using Airavata are the following. *First*, the administrator registers the Gateway with Airavata system initializing the required security credentials and registry workspace. *Second*, the administrator registers scientific applications, workflows, and computational hosts with Airavata. When combined, these provide all the information necessary to execute the application on the remote resource. This typically involves deployment and testing of applications on the target resources as well. *Third*, the administrator configures GFAC with appropriate handlers and providers for the target resources. Airavata comes with many preconfigured handlers and providers for different Grids, clouds, and non-Grid resources (as listed earlier), but gateway operators need to make specific configuration choices. In some instances, the gateway operator may need to write new providers or handlers. Fourth, the administrator manages community security credentials using the credential store. Finally, the administrator configures third-party User Identity and Data Management systems so Airavata could as an optional step could directly deposit data as opposed to fetched on request.

In its current form, Airavata does not directly manage users or groups. Instead, it establishes a trust relationship with the gateway and trusts the gateway's authorization and authentication decisions. A fuller discussion of Airavata's security model is given in [8].

End user scientists interact with Airavata through their gateway to create online experiments, submit and monitor jobs, and analyze or download outputs. The steps for a scientist using a gateway to perform a computational experiment are the following.

First, the end user interacts with a science gateway through a a Web browser to create and submit an online experiment. *Second*, the gateway collects this information and passes it to the Apache Airavata server through the API Server. Note the API Server is not a traditional Web server. Interactions between the gateway client and the API Server use TCP/IP on a configurable port. *Third*, the gateway creates an experiment through the API Server, which invokes the Registry's CPI to create a new record. Airavata returns a unique handle (generated by the Registry) that can be used to access the experiment. *Fourth*, the gateway user uploads any input data needed by the experiment, specifies input parameters, optionally selects the resources to be used, and submits the

experiment. Within Airavata, these are handled by the Orchestrator, which the gateway accesses through the API server. The Orchestrator pulls together all information and files needed to run the job or jobs on a selected resource and passes this information to GFAC (in the case of single jobs) or to the Workflow Interpreter (in the case of workflows).

Fifth, within Airavata, GFAC is responsible for staging input input files and for preparing and launching the jobs on the remote resource. GFAC providers are clients to various Grid and cloud submission mechanisms. *Sixth,* the gateway user can monitor the progress of the job through Airavata's monitoring API. The API also provides access to intermediate data. Within Airavata, the Job Monitor provides this capability. The Job Monitor is decoupled from the submission mechanisms in GFAC, allowing us to have multiple monitoring mechanisms. For example, on XSEDE, we may monitor jobs with both pull approaches ("qstat" or similar Grid mechanisms) and push messaging [9]. *Seventh,* if the job completes successfully, the gateway user can download outputs from the remote resource via Airavata. GFAC is responsible for staging files off the machine and back to the gateway. This is accomplished through various mechanisms that the gateway may select: by using GlobusOnline or direct GridFTP clients, by using SCP, and by using HTTP. *Eighth,* in cases of job failures, Airavata will also preserve intermediate outputs and standard error files.

Airavata's component interaction design is the outcome of an ongoing effort within the Airavata project to provide greater fault tolerance by centralizing system state within the Registry. The challenge for Airavata state management is that it must act as a bridge between the client (left side of Figure 1) and the computing resources (right side of Figure 1), both of which are outside of Airavata's direct control. In particular, a job submitted to a resource (an XSEDE supercomputer, for example) has a state (queued, executing, completed or failed) that evolves independently of Airavata, is only accessible through monitoring mechanisms, and may last for minutes, hours, or days. This individual job may be part of a workflow as well, introducing another layer of statefulness. Because of this, we have introduced a three level state model in our Thrift data model for managing applications. A full description is out of scope for the current paper, but we provide a short summary. "Job" state is a mapping to specific job submission states on a target machine. "Task" state is associated with the Job state and any "handler" states surrounding the Job; a Job can succeed but its Task can fail. Lastly, Experiments can contain multiple Tasks and have their own state. The full state model is described explicitly in Airavata's Thrift file definitions available through the project's public code repository.

While centralization of state management in the Registry is an important recent effort, it is not desirable to think of all Airavata components as completely stateless. For example, the GFAC component may need to do several preprocessing steps to complete job submission, such as stage in a file or provision a virtual machine. Deciding when to recover a partial GFAC submission, for example, is still an open problem.

III. Science Gateway Case Studies

We now summarize two gateway collaborations and illustrate how they have contributed to Airavata's design. We use these to illustrate the balance we must make between custom integration on the one hand and scalable support on the other. For a list of collaborating projects, see http://airavata.apache.org/community/projects-using.html.

A. The UltraScan Science Gateway

The UltraScan Science Gateway [10] is an XSEDE science gateway that enables experimental biophysicists to use high performance computing resources to perform data analysis on analytical ultracentrifugation experiments as well as manage their data sets through a laboratory information system. UltraScan is also being extended to support modeling and simulation for related experimental techniques like small angle scattering. UltraScan's use case can be summarized as follows. *First*, users select samples and create input packages from the UltraScan database using desktop tools. *Second,* users stage their inputs, run their analysis on supercomputers, and stage out their results using a Web-based gateway. Analysis can take place on the XSEDE, on campus resources provided by the UltraScan principal investigator, or on the Juropa supercomputer resource. *Third,* While running, the application generates status updates via UDP messages. *Fourth,* when the job completes, data is staged off the resource.

Thus UltraScan provides a driving use case for running single applications for multiple users on multiple, uncoordinated resources. Even the simple "happy path" above has many pitfalls resulting from heterogeneity of grid resources and middleware. UltraScan requirements have had significant impact on the design of Airvata's Orchestrator and GFAC components. UltraScan has also influenced our API Server. In our initial integration with UltraScan, we focused exclusively on exposing the GFAC component. Other components (particularly the Registry) were still needed internally but not exposed to the gateway developers. This led to a very customized integration between UltraScan and Airavata that included several supplemental services outside the core of Airavata, some of which duplicated Registry services. This custom integration, while useful in a one-off case, segregates gateway-Airavata integration into code extensions that are outside the main framework and well understood only by a subset of Airavata developers. This results in un-scalable support effort for readily apparent reasons. By switching to a common API and removing supplemental services, any Airavata developer can diagnose problems with UltraScan operations.

B. The ParamChem Science Gateway

The ParamChem Science Gateway [5] uses Apache Airavata to address the computational intensive needs of empirical force field parameter optimization for chemical systems. ParamChem use case of Apache Airavata workflow capabilities can be summarized as follows. First, an expert user constructs workflows to models energy functions or Hamiltonians and registers with Workflow Catalog within the Airavata Registry. The commonly used workflow wraps molecular dynamics applications as workflow tasks to perform

atom typing; generation and optimization of initial guess charges and Lennard-Jones parameter assignment; generation and optimization of target data for charge optimization; generation and optimization of target data for optimization of bond, angle, dihedral and improper dihedral parameters; and generation and optimization of target data for optimization of dihedral parameters about rotatable bonds.

Next, using a workflow template from the Registry, the user first visually constructs the specified molecule on their desktop and interact with with ParamChem CGenFF Service to obtain the initial guess parameters. Next, the user then walks through steps in the ParamChem user interface to launch the Dihedral Optimization process. The Paramberoo GUI bundles an Airavata Client SDK that interacts with the Airavata Server to configure and launch and monitor the optimization workflow. Finally, the user fetches the optimized results through the Airavata Registry and visualizes the results to obtain plots such as the comparison of QM and MM energy.

From this description, it is clear that ParamChem requirements have influenced the Workflow Interpreter's design. However, ParamChem has also influenced our overall design philosophy for client interactions as well: Airavata services need to be accessed through a single API that exposes the full functionality of the system rather than individual pieces. Finally, ParamChem also presented an interesting set of requirements for acting as a desktop rather than Web-based gateway. Proper support for desktop clients using Apache Thrift is an active current effort.

C. *Experiences from Use Cases*

Apache Airavata has been applied to a number of different scientific domains to act as both a scientific workflow engine for small research groups (for example [4]) and as an online service that powers science gateway application management. One of the challenges we find with gateways (as illustrated in the UltraScan case) is scaling our operations and support efforts, not just scaling our software.

Airavata is a complicated collection of software that is still best operated by our group at this stage of its development, suggesting we adopt a service model for dealing with collaborating gateways. Even with this constraint, problems can arise. First, we must understand the gateway's use case in some detail from the beginning. Otherwise, it is likely that we will evolve many workarounds and gateway-specific solutions for common problems. Second, excessive, unnecessary customization for a specific gateway decreases reliability. Customized gateway-Airavata integrations result in too many deployed services that do not pass through the regular Airavata development and release processes. Third, customized interactions between the gateway and Airavata decrease sustainability. Operating a gateway with a very customized interface to Airavata services becomes specialized knowledge of a subset of the development team. This results in only a subset of an operations team being able to troubleshoot a particular problem. Finally, customizations result in unnecessary duplication of effort. We must instead look for generalizations that can be incorporated into Airavata's core code base and expressed through the Airavata API. These also help us integrate with new gateways by articulating common use cases to the gateway developers, encouraging them to develop their gateways following common patterns.

IV. Project Governance and Sustainability

Apache Airavata is a top-level project within the Apache Software Foundation (ASF). We have made 11 releases (0.1 through 0.11) through the Apache Software Foundation mechanisms, which includes proper binary and source packaging along with necessary licensing and notification files. Release 0.12 is in preparation at the time of writing. Our release 1.0 will be determined by community vote when the API is sufficiently stable. Following the 1.0 release, we will adopt semantic versioning (that is, all 1.X.Y releases will have a compatible API).

It has been the authors' goal to bring the ASF's open governance principles to academic research software development and management. Open governance goes beyond the usual open source metrics [11] and beyond simply taking advantage of free services (such as GitHub) that support open source projects. Projects with open governance have the usual open source characteristics but also make decisions through public forums with voting by stakeholders. Important characteristics of openly governed projects are the diversity of their stakeholders and the willingness of the project to add new stakeholders via a well-defined process. We believe this is an important complementary effort to the open standards work that have been pursued by the Open Grid Forum and other venues. Openly governed software encourages open, community-wide reference implementations of the standards, allowing the cyberinfrastructure development community to both collaborate (on the reference implementation and common features) and compete (on extensions and advanced capabilities).

Sustainability is a challenge for many academic software projects. Specific strategies (like commercialization) aside, academic software must a) demonstrate relevance to user communities; b) provide a way to transform users into stakeholders who contribute to the project in addition to using it; and c) foster a pipeline of developers and architects at all skill levels who can directly contribute to the software. Airavata's strategy for solving these problems has been to adopt the Apache Software Foundation's "community over code" open project governance approach. The core idea from the point of view of academic software projects is that sustainability comes from a diversity of stakeholders (that is, a community), and a diversity of stakeholders requires a governance model to work.

ASF methods are simple but profound: projects make decisions openly on archived mailing lists; all discussions occur or are summarized on mailing lists or other open venues (like IRC channels and Google Hangouts); and projects have a flat hierarchy. Typical decisions include when to release software, priorities for project features and improvements, and design changes. All decisions are public with two exceptions: if a contributor has earned enough merit to warrant committership (that is, write access to the code base) or not, and if a committer has deserves promotion to the Project

Management Committee (that is, given full voting writes) or not. Voting in the strictest sense can go be straight up or down tallies, but Airavata and other projects typically seek to achieve consensus.

V. Related Work

Science gateways and science gateway frameworks are numerous and have been the subject of several workshops. Gateways are supported by both the XSEDE science gateway program [12] and the EGI through SCI-BUS [13]. Several efforts to provide "as a service" gateways are directly comparable. The HUBzero project [14] provides a turnkey gateway hosting solution that includes extensive collaboration capabilities as well as application execution management; HUBzero is also available as an open source package. The iPlant Agave project [15] provides a REST-based Platform as a Service as well as client building tools for the Life Sciences community. The Globus team has developed a cloud-style model for reliably managing data transfers and for sharing data [16].

Compared to these other efforts, Apache Airavata focuses on using a Thrift-based "RESTless" approach to client development. We are interested in Airavata as a "Platform as a Service" and collaborate with other gateways to provide domain-specific frontends. There are strategic reasons as well as technical reasons for this: successful gateways need champions who interact with their communities. Airavata's goal is to provide very targeted services that help gateways while being as flexible as possible on how gateways can integrate the Airavata clients.

Scientific workflow systems share many commonalities and generally encapsulate the science gateway's usage scenario as an execution pattern, and also map that execution pattern to the atomic functionalities of grids. Airavata's approach is similar to SCI-BUS in this regard. Workflow systems and science gateways, generally treated as separate tools within the cyberinfrastructure software layer, are actually mutually beneficial. Workflow systems by themselves can be too abstract and far removed from the scientific use cases of a specific domain to provide ready accessibility for community members. Successful gateways on the other hand connect direct with scientific users but may not be able to support the complicated execution patterns supported by workflow engines.

VI. Future Directions

Our main challenge in Airavata is to design the software so that it can work as a hosted cloud service that is both elastic and fault tolerant. That is, we must be able to isolate system state (which includes proxies for applications on remote resources) into a single component, the Registry, while keeping other parts as stateless as possible. This means that we can more easily create and destroy other parts of Airavata (such as GFAC) as needed to handle load spikes and recover from failures. It will also be necessary to explore how best to handle per-gateway configurations, as different gateways will want to use different resources in different ways through the same Airavata service. We currently accommodate this with editable configuration files, but this assumes each gateway has full access to its own Airavata instance. This approach will not scale. We anticipate that Apache ZooKeeper will be useful here for tracking running component instances and plan to explore this.

Airavata also must be a platform for distributed computing research for its own sake. Our use of CPIs enables us replace different component implementations. For example, the Orchestrator CPI may include "simple" implementations for quick testing and conservative "production" implementations that are used in deployment as well as "research" implementations that incorporate smarter (or at least more complicated) scheduling algorithms using externally supplied information services. Likewise, the Messenger component, while reliable, is an in-house implementation that predates the availability of high quality messaging systems such as RabbitMQ, Apache QPid, and Apache Kafka.

The Registry is also subject for significant revisions. One long-standing design choice is that we overload the Registry with many different types of data and provide strong relationships between different data. This is a consequence of carrying XML data structures and their tree structures forward from Airavata's predecessors. The explicit use of XML has ended in the current versions (0.12 and higher) of Airavata, and we need to rethink how we can make more modular data models by examining use cases. There is a temptation to jump to NoSQL solutions, but the strong theme on Airavata architecture discussion threads is to carefully think through our requirements first, since our data sizes are not likely to be large.

VII. Conclusions

This paper has surveyed the Apache Airavata project and its software framework. The key contributions of this paper are the description of Airavata's overall design as a general purpose application and workflow management framework for science gateways, an examination of its integration with science gateways through usage scenarios, a discussion of how these case studies drive Airavata's evolution, and the authors' use of Apache Software Foundation open governance mechanisms. The core theme of this paper is the balance between distributed computing research and operational usage. This balance requires thoughtful governance.

This paper presents for the first time two new Airavata components: the API Server and the Orchestrator. More detailed descriptions of these components are the subject of future publications. Here our goal is to show the relationships of these components to the other parts of Airavata. We also summarized significant design changes in the Registry, with are needed both to support the use of Apache Thrift and to centralize Airavata state management. The latter is required to make Airavata more elastic. Finally, we identified future development work for the Registry, GFac, and Messenger components.

Acknowledgment

This work has been funded by NSF awards ACI-1339774, OCI-1216730, and OCI-1127210. We thank Eroma

Abeysinghe for critical reading of the manuscript and the Apache Airavata Program Management Committee for contributions to the software architecture and implementation.

REFERENCES

[1] Droegemeier, Kelvin K., Dennis Gannon, Daniel Reed, Beth Plale, Jay Alameda, Tom Baltzer, Keith Brewster et al. "Service-oriented environments for dynamically interacting with mesoscale weather." *Computing in Science & Engineering* 7, no. 6 (2005): 12-29.

[2] Alameda, Jay, Marcus Christie, Geoffrey Fox, Joe Futrelle, Dennis Gannon, Mihael Hategan, Gopi Kandaswamy et al. "The Open Grid Computing Environments collaboration: portlets and services for science gateways."*Concurrency and Computation: Practice and Experience* 19, no. 6 (2007): 921-942.

[3] Marru, Suresh, Lahiru Gunathilake, Chathura Herath, Patanachai Tangchaisin, Marlon Pierce, Chris Mattmann, Raminder Singh et al. "Apache airavata: a framework for distributed applications and computational workflows." In *Proceedings of the 2011 ACM workshop on Gateway computing environments*, pp. 21-28. ACM, 2011.

[4] Erickson, Brandon, Raminderjeet Singh, August E. Evrard, Matthew R. Becker, Michael T. Busha, Andrey V. Kravtsov, Suresh Marru, Marlon Pierce, and Risa H. Wechsler. "A high throughput workflow environment for cosmological simulations." In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond*, p. 34. ACM, 2012.

[5] Ghosh, Jayeeta, Suresh Marru, Nikhil Singh, Kenno Vanomesslaeghe, Ye Fan, and Sudhakar Pamidighantam. "Molecular parameter optimization gateway (ParamChem): workflow management through TeraGrid ASTA." In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, p. 35. ACM, 2011.

[6] Kandaswamy, Gopi, Liang Fang, Yi Huang, Satoshi Shirasuna, Suresh Marru, and Dennis Gannon. "Building web services for scientific grid applications."*IBM Journal of Research and Development* 50, no. 2.3 (2006): 249-260.

[7] Huang, Yi, Aleksander Slominski, Chathura Herath, and Dennis Gannon. "Ws-messenger: A web services-based messaging system for service-oriented grid computing." In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, vol. 1, pp. 8-pp. IEEE, 2006.

[8] Thejaka Amila Kanewala, Suresh Marru, Jim Basney, and Marlon Pierce, "A Credential Store for Multi-Tenant Science Gateways," International Symposium on Cluster, Cloud and Grid Computing (CCGrid), May 2014, Chicago, IL. http://hdl.handle.net/2022/17379

[9] Hanlon, Matthew, Warren Smith, and Stephen Mock. "Providing resource information to users of a national computing center." In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, p. 43. ACM, 2013.

[10] Demeler, Borries. "UltraScan: a comprehensive data analysis software package for analytical ultracentrifugation experiments." *Modern analytical ultracentrifugation: techniques and methods* (2005): 210-229.

[11] OSSWatch: http://oss-watch.ac.uk/

[12] Wilkins-Diehr, Nancy. "Special issue: Science gateways—Common community interfaces to grid resources." *Concurrency and Computation: Practice and Experience* 19, no. 6 (2007): 743-749.

[13] Kacsuk, Peter, Gabor Terstyanszky, Akos Balasko, Krisztian Karoczkai, and Zoltan Farkas. "Executing Multi-workflow simulations on a mixed grid/cloud infrastructure using the SHIWA and SCI-BUS Technology." *Cloud Computing and Big Data* 23 (2013): 141. See also https://www.sci-bus.eu/.

[14] McLennan, Michael, and Rick Kennell. "HUBzero: a platform for dissemination and collaboration in computational science and engineering." *Computing in Science & Engineering* 12, no. 2 (2010): 48-53.

[15] Agave API Developer Site. Available from: from: http://agaveapi.co

[16] Foster, Ian. "Globus Online: Accelerating and Democratizing Science through Cloud-Based Services." *IEEE Internet Computing* 15, no. 3 (2011).

[17] Goodale, Tom, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Gregor Von Laszewski, Craig Lee, Andre Merzky, Hrabri Rajic, and John Shalf. "SAGA: A Simple API for Grid Applications. High-level application programming on the Grid." *Computational Methods in Science and Technology* 12, no. 1 (2006): 7-20.

[18] Foster, I., A. Grimshaw, P. Lane, W. Lee, M. Morgan, S. Newhouse, S. Pickles, D. Pulsipher, C. Smith, and M. Theimer. "OGSA® Basic Execution Service Version 1.0." (2007).

[19] Morgan, Mark M., and Andrew S. Grimshaw. "Genesis ii-standards based grid computing." In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pp. 611-618. IEEE, 2007.

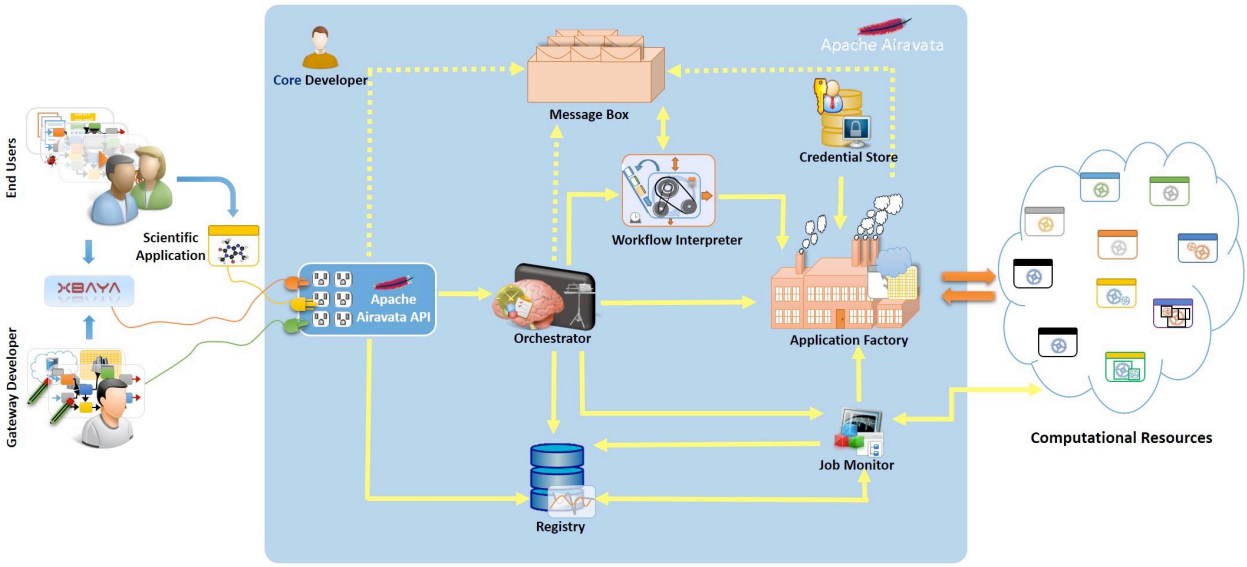

Figure 1 Airavata component interactions. See text for a full description.