

SPECIAL ISSUE PAPER

The GenApp framework integrated with Airavata for managed compute resource submissions

Emre H. Brookes^{1,*†}, Nadeem Anjum², Joseph E. Curtis³, Suresh Marru⁴,
Raminder Singh⁴ and Marlon Pierce⁴

¹*Department of Biochemistry, University of Texas Health Science Center, San Antonio TX, USA*

²*Department of Computer Science and Engineering, IIT Kharagpur, Kharagpur 721302, India*

³*NIST Center for Neutron Research, National Institute of Standards and Technology, Gaithersburg 20899, Maryland*

⁴*Pervasive Technology Institute, Indiana University, Bloomington IN, USA*

SUMMARY

A new framework (GenApp) for rapid generation of scientific applications running on a variety of systems including science gateways has recently been developed. This framework currently builds a GUI and/or web-based user interface for a variety of target environments on a collection of executable modules. The method for execution of modules has limited framework restrictions: primarily the requirement of wrapping the application to accept input and output formatted in JavaScript Object Notation (JSON). Initial implementation supports direct execution on a user's workstation, a web server, or a compute resource accessible from the web server. After a successful initial workshop utilizing the framework to create a web-based user interface wrapping a scientific software suite, it was discovered that long-running jobs would sometimes fail, because of the loss of a Transmission Control Protocol (TCP) connection. This precipitated an improvement to the execution method with the bonus of easily allowing multiple web clients to attach to the running job. To support a diversity of queue managed compute resources, a Google 'Summer of Code' project was completed to integrate the Apache Airavata middleware as an additional execution model within the GenApp framework. New features of file management, job management with progress, and message box support are described. *Concurrency and Computation: Practice and Experience*, 2015. © 2015 Wiley Periodicals, Inc.

Received 23 January 2015; Revised 23 March 2015; Accepted 24 March 2015

KEY WORDS: CASE tools; Science Gateway; middleware; design; human factors; languages

1. INTRODUCTION

The GenApp framework [1] is a product of a joint UK Engineering and Physical Sciences Research Council and USA National Science Foundation grant entitled 'CCP-SAS – Collaborative Computational Project for advanced analysis of structural data in chemical biology and soft condensed matter' and is an SI2-CHE cyberinfrastructure project addressing Grand Challenges in the Chemical Sciences. The CCP's initial software elements are primarily small-angle scattering (SAS) software simulation and analysis tools developed by multiple independent laboratories. SAS can be performed using individual lab X-ray sources but is more frequently performed at high-energy synchrotrons and/or neutron sources. A collimated beam of X-rays or neutrons with a fixed

*Correspondence to: Emre H. Brookes, Department of Biochemistry, University of Texas Health Science Center, San Antonio, TX, USA.

†E-mail: emre@biochem.uthscsa.edu

wavelength are scattered by the sample and are recorded on a two-dimensional detector. For solution studies, where particles in solution are typically randomly oriented, the two-dimensional data are radially integrated to produce a one-dimensional scattering curve that contains structural information. Some computations can be quite trivial such as producing various transformed plot views, and others can be computationally expensive such as rigid body modeling and expansion of conformational space utilizing various molecular dynamic and Monte Carlo methods and their subsequent scattering simulation and screening. The grant includes aims of developing new and enhanced SAS analysis methods as well as the development and implementation of the cyberinfrastructure bringing together three preexisting software packages that analyze SAS data [2–5]. Considerations during the design phase of GenApp were based upon observations and discussions with developers of existing independently produced scientific software. Common issues include ease of deployment in an ever-evolving software landscape, support for legacy codes, and the fact that science research groups cannot typically afford a dedicated software team. The design employed was to divorce the computational modules from the user interface and define their inputs and outputs on an easily extensible set of data types. We define applications as seen by the user on a set of preexisting and user-supplied modules. Separately, we define an easily extensible set of user interfaces and execution models known as target languages. Finally, the stage is set to generate working instances (Figure 1).

Apache Airavata [6] is open source, open-community distributed system software framework for supporting the metadata and application execution requirements of both science gateways and scientific workflows. Airavata consists of the following internal components. The Registry is used to store, access, and manage descriptions of applications, computing resources, and computational experiments. The Orchestrator is used to schedule executions of both single applications and workflows. Generic FACTory (GFAC) is the component that interactions with external resources and services needed to execute a specific task. The Workflow Interpreter manages steps in experiments that consist of multiple tasks. The Messenger provides publish–subscribe messaging for both internal components and external consumers. Airavata has an Apache Thrift-defined API, which it exposes through the API Service component. The API uses a rich data model, which is enveloped in an Experiment object. An experiment can be a single-application execution or a composite application with a sequence defined as a workflow. These components and their interactions are more fully described in [7]. Airavata is also serving as one of the bases of a hosted Science Gateway Platform as a service, currently under development. Airavata has supported the

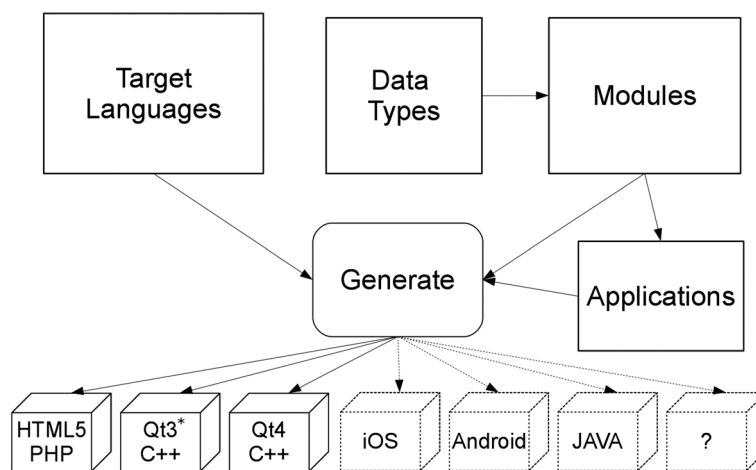


Figure 1. Generating application instances. The generator reads application definitions, module definitions, and chosen target language information to assemble the application instances. Currently generated target languages include HTML5/PHP, Qt3, and Qt4. Qt5 is being tested and is planned to be used to generate iOS and Android apps. *Certain commercial equipment, instruments, materials, suppliers, or software are identified in this paper to foster understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

execution management of the UltraScan and UltraScan-SOMO [4] gateways, which are precursors of the current work.

2. FRAMEWORK

The GenApp framework builds complete applications from a collection of executable modules. We define four roles, the framework (tool) developer who works with the framework generation code, the module wrapper who wraps executable modules to work within the framework, the application developer who organizes wrapped modules into an application, and finally the end user who utilizes the generated application. All framework definition and control files are formatted in JSON [8]. There are two primary restrictions. The first is that the modules must be modified to accept and produce specifically formatted JSON inputs and outputs. The second is that the input and output fields must be valid field types (e.g., integer, float, text, vectors, plot data, and atomic structures). Note that the set of valid field types are extensible, but under the scope of the framework developer, not the typical module wrapper or application developer. The module inputs, outputs, executable, and ancillary information (such as help text and compute resource targeting) are placed in a module-specific definition file. Selected wrapped modules are organized for the user interface and put into the menu definition file. General directives such as target languages are placed in the directives definition file. Once these definition files have been properly created, the GenApp executable is run to produce a complete application. The GenApp executable is a single command-line program requiring no arguments that is run by the application developer in the proper directory and produces applications based upon the definition files. Complete applications are produced currently for GUI-based Qt3/C++ and Qt4/C++ and web-based HTML5/PHP. The code produced is placed in individual subdirectories specific for each target language. The HTML5/PHP target currently requires an appropriate server setup, and the Qt-based generation requires compiling, although the design of the framework allows extensions to include these steps if deemed helpful. A screenshot of a produced example HTML5/PHP user interface is shown in Figure 2. Further details about the framework can be found in [1].

3. EXECUTION

Prior to this work, code-managing execution in GenApp was generated for direct execution on either the client computer (for GUI-based target languages) or directly on a web server (for web-based target languages) or via Secure SHell (SSH) from the web server. To provide support for queuing cluster and High Performance Computing (HPC) compute resources, Apache Airavata was chosen to act as middleware. In this section, we will describe the execution model of GenApp and how it was modified to integrate with Airavata.

3.1. GenApp

User-supplied executable modules are wrapped with JSON definitions of their input and output. Typically, a driver script is written to wrap execution modules so that minimal or no change to the underlying scientific execution module is required. The driver script for each module must accept standard input in JSON and provide standard output as JSON.

The execution models for the modules, prior to Apache Airavata integration, proceed as follows and as shown in Figure 3. The Qt/C++ GUI target language generates a GUI application where each module is executed directly on the user's workstation. The HTML5/PHP target language executes the module through an Asynchronous Javascript And Xml (AJAX) call to a PHP module that directs execution of the module.

As part of the CCP-SAS project, a dedicated compute resource was installed at the University of Tennessee Knoxville to host HTML5/PHP targets for SAS software, initially the SASSIE software [3]. The server is a Dell cluster running Rocks [9] with two 64-core compute nodes, an eight-NVIDIA K20m GPU enclosure, and a 12-core head node. The head node is running a virtual

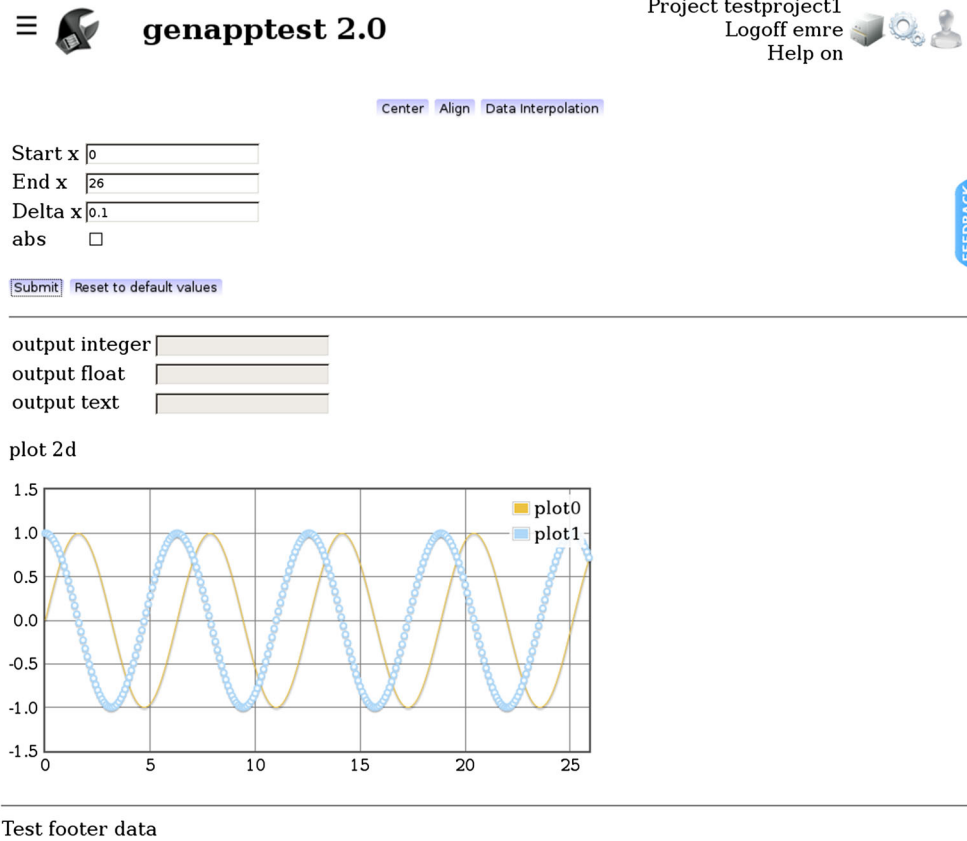


Figure 2. An example GenApp HTML5/PHP-generated user interface for one module. The top row contains the menu controls, the application name, the current project, a file manager, the job manager, and user management icons.

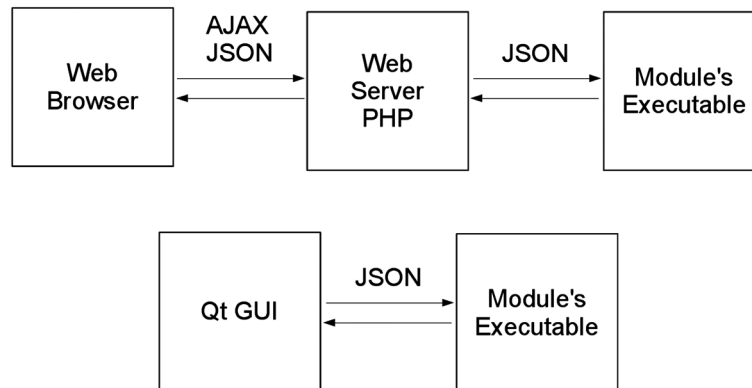


Figure 3. Execution models for target languages HTML5/PHP (top) and Qt/C++ (bottom). The JSON input to and received from the executable will be generated over the same set of fields described in the module definition file, so the executable can remain unchanged in both cases.

machine hosting the web server for the HTML5/PHP server. To utilize the compute nodes for execution, we extended the job submission mechanism to support direct SSH execution of modules. The globally available resources are defined as name/value pairs in a JSON application configuration object where global default resource is also defined (Text 1). Each individual module's definition file can also provide an overriding module-specific resource or resources. The changes to

the GenApp HTML5/PHP target language module execution required an appropriate prefix to the executed command based upon the called module's determined target resource. This basic resource targeting was functional for our first user's workshop given at the American Conference of Neutron Scattering in June 2014 [10]. Of course, such a system has limited resources and could become saturated with user requests. Therefore, the long-term plan is to support queued resources by integrating with Airavata middleware.

The workshop was successful, but subsequently an issue came to light with respect to long-running jobs. The HTML5's AJAX call to PHP was waiting on completion of the job to return the JSON results. This required the TCP connection to remain open. It did not appear during our testing with directly connected web clients, nor for general users' running jobs of less than approximately 1 h, but users reported failures of long jobs that appeared to be the result of a dropped TCP connection during the AJAX call. This precipitated a redesign of the HTML5/PHP execution method. In the updated execution method, the HTML5 initiates an AJAX call, the PHP starts the module execution under a monitor daemon and returns immediately with a simple acknowledgment JSON object. Subsequently, the monitor daemon waits on the job completion and messages the HTML5 client via a pub-sub WebSocket (see Section 4 for messaging details). Once messaged, the HTML5 client produces another AJAX call to retrieve the results. Additionally, the HTML5 client polls for results (currently at 16 s intervals) as a fallback for the case of a lost message. The updated execution model is shown in Figure 4.

An additional benefit of the updated enhanced execution model is that previously executed module results are easily retrieved and multiple web clients can attach to a running module. These features are both implemented in the current version of GenApp. A minor downside is that additional storage is required to hold the results and should be managed and eventually purged by the system administrators.

3.2. Airavata integration

Integration with Apache Airavata middleware provides GenApp an additional execution model. This execution model supports queuing compute resources. The integration was completed as a Google Summer of Code 2014 Project [11], providing GenApp the capability to execute long-running, noninteractive jobs on distributed computing resources, including local clusters, supercomputers,

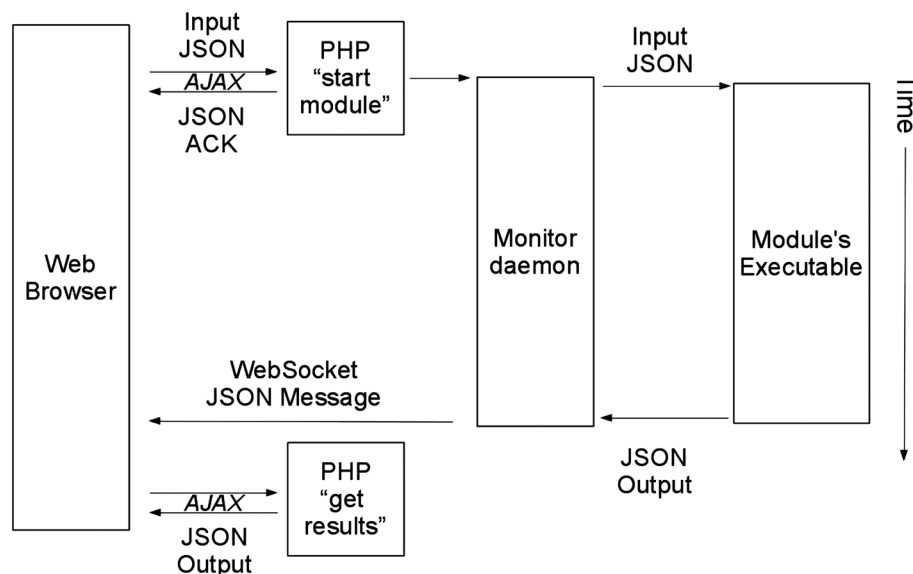


Figure 4. Two-step HTML5/PHP execution. The first AJAX call is quickly acknowledged (ACK). The client web browser makes a second AJAX call to retrieve the results once messaged that the module's execution has completed. Additionally, there is a 16 s interval poll running on the web browser checks for results in the case the WebSocket message is for some reason not received. The web server components coordinate job status via a database (not shown).

national grids, and academic and commercial clouds. The integration has been achieved for both the HTML5/PHP and C++/Qt target languages. The overview of the integration can be seen in Figure 5.

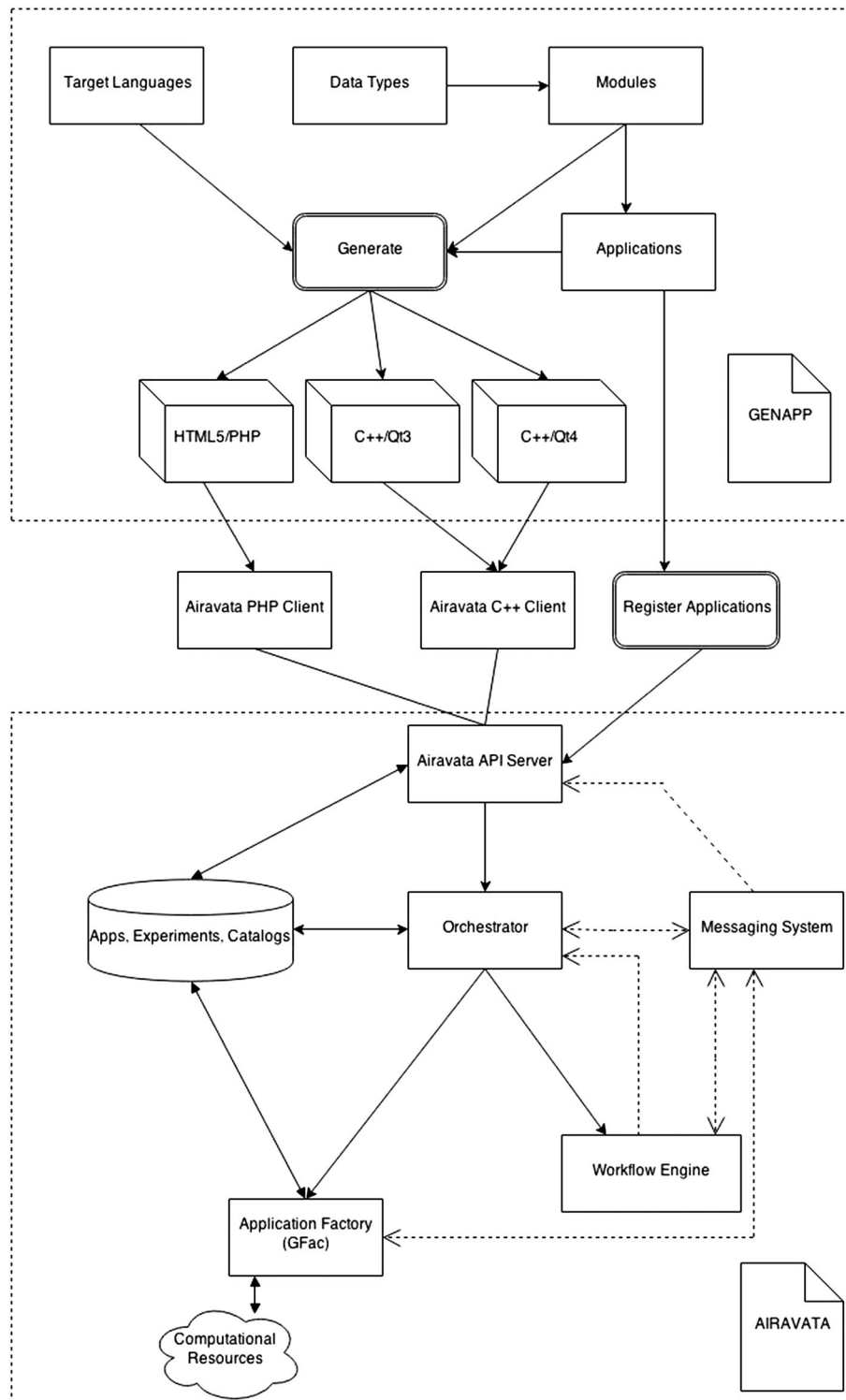


Figure 5. Overview of GenApp–Airavata integration. GenApp (top) produces applications (middle) that call the appropriate Airavata (bottom) client as part of their execution of modules. The Airavata API provides managed submission to a diverse set of computational resources.

Airavata requires applications to be registered in a catalog before they can be executed. Using a simple utility, Register Applications, all GenApp executable modules can now be registered with the Airavata Server automatically. This registration step is a one-time task and can be achieved with a single command to execute the registration utility. Once the registration utility has been executed, job submissions can be redirected to Airavata.

GenApp executes short jobs locally and only uses Airavata for long-running jobs, and a configuration option has been provided based on a variable setting in the input JSON file to enable or disable job submission to Airavata. There is a global default for resources defined, that is, local, Airavata or SSH, along with a module-specific override. This provides one the option to execute the required modules on the required resource for maximum resource optimization.

Jobs are submitted to Airavata through Apache Thrift-based [12] Airavata clients written in PHP and C++, which make calls to the Airavata API. The execution steps include creating an Airavata project, creating an experiment (Airavata's terminology for a specific instance of a module execution) within the project, launching the experiment, checking its status, and retrieving the output(s). The experiment creation step includes specifying the relevant GenApp module, which had been registered by the 'Register Applications' utility, as the executable and the user inputs retrieved in JSON format as the input. Airavata's Orchestrator creates a process for the experiment. If the experiment requires it, multiple processes are created. GenApp may request to submit the application to multiple resources concurrently and use the fastest available one. A failed process is retried on the same or alternative resource. A failed process can also be retried using alternative protocols (e.g., Globus Resource Allocation Manager (GRAM), Uniform Interface to Computing Resources (UNICORE), or SSH). Further, scheduling of computational resources is performed at this step. A process is then handed off to Airavata's GFAC, which breaks down the process into set of tasks required to accomplish by a combination of GFAC input handlers, output handlers, and a provider. Each task might have a lower-level implementation like Job, Data Movement, Data Analysis, which have their own invocation identifiers. After launching the experiment, the status is checked for completion. The experiment ends with either 'COMPLETED' state or 'FAILED' state. On successful completion, the output(s) are retrieved and displayed to the user. In case the experiment fails, the appropriate error message is retrieved and displayed to the user. Status updates through the system are rolled up from the lowest-level Data Movement, Job Submission, and Data Analytics. The Task states are inferred based on the associated task, and Process status is inferred based on all the tasks forked. Finally, Experiment Status is inferred based on all the processes. Extensive error handling mechanisms have been implemented to notify the user in case of any exceptions in any step of the execution. The overview of the two-step HTML5/PHP submission mechanism with Airavata managed execution can be seen in Figure 6.

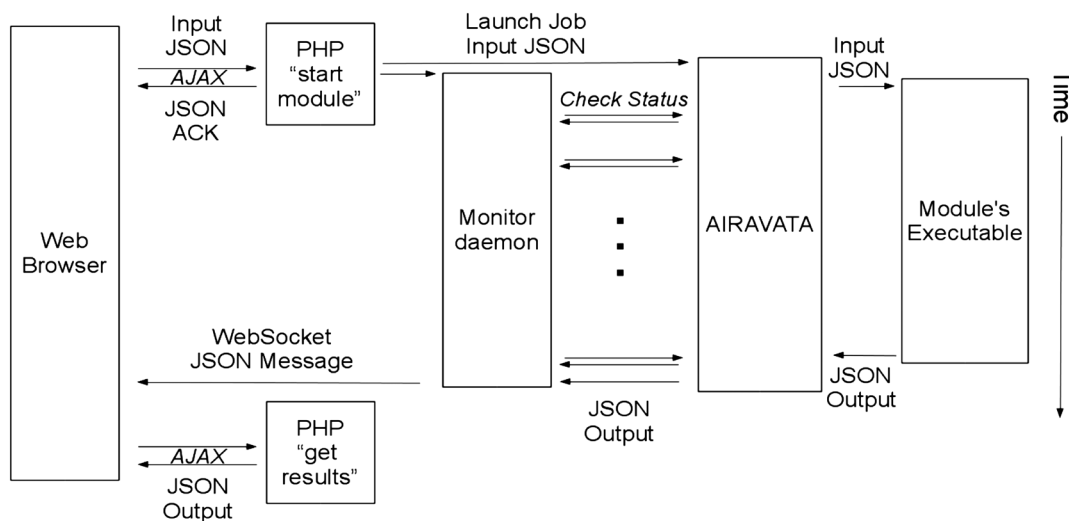


Figure 6. Two-step Airavata-managed HTML5/PHP execution. The module's execution is launched on the first AJAX call from the client which receives an acknowledgment (ACK) and monitored via a 'Check Status' poll. Airavata manages the submission of the module's execution on remote compute resources.

4. MESSAGING

Messaging facilities provide the executing module a mechanism to update the user interface with intermediate results, progress, and message box information. To enable messaging, appropriate code must be explicitly added to the module's driver script or within the module's executable depending on the granularity of messaging desired. Messaging is not mandatory, but does improve the end user experience by providing timely information during execution. All messaging consists of JSON-formatted information, and the full set of the module's output field definitions are available along with additional attributes specific to progress and message boxes. Messaging facilities are currently only supported in the HTML5/PHP target language.

4.1. WebSocket

Messaging in the HTML5/PHP target is shown in Figure 7. The executable sends a JSON output to a User Datagram Protocol (UDP) server to provide status updates. UDP was chosen for these messages to not delay the executable's progress, which may be running on an expensive cluster compute resource ('fire and forget'). Reliable TCP-based messaging can be implemented utilizing the same mechanism. The UDP server writes to a ZeroMQ [13] socket, which is running a Ratchet-based [14] server. The Ratchet server runs a subscribe/publish message relay on a WebSocket [15]. When the HTML5 web client 'submits' to run a module's executable, it produces a unique ID defining the 'topic' for the subscription, which is passed in JSON to the executable and also registered with the WebSocket server. The executable sends this 'topic' as part of the JSON output to the UDP server. When the client receives the final output from the executable, it unsubscribes from the 'topic' on the WebSocket. This mechanism provides live updates to the client. The Ratchet server additionally writes the last message received from an executable to a Mongo [16] database for last message refresh to a reattaching client (Section 6).

4.2. Progress

An executing module can provide the framework with an overall progress indication by including a top level name/value pair of '_progress' assigned a decimal value between 0 and 1 to each UDP message sent. The '_progress' attribute will be stored in the database keyed with the assigned GUID assigned to each submission and is thus accessible to the job manager.

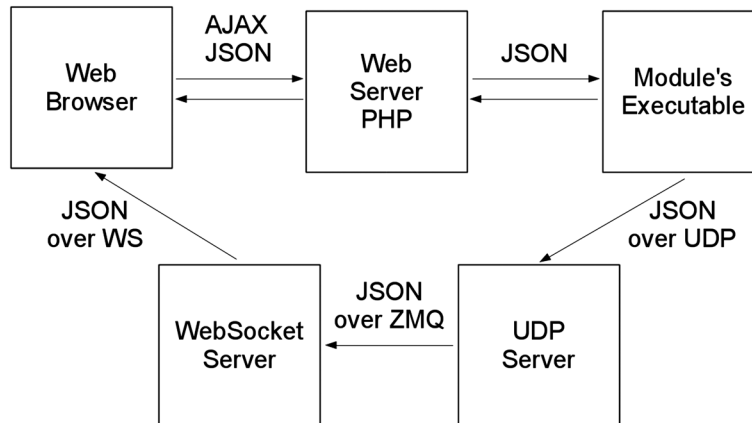


Figure 7. Messaging in the HTML5/PHP target language. Messages are sent from the executable via UDP to allow the executable to continue and are received by the client web browser via WebSockets (WS). ZMQ is a ZeroMQ socket interface.

4.3. Message boxes

An executing module can direct the client to produce a message box by including a name/object pair of ‘_messagebox’ and the object must contain two name/value pairs of ‘icon’ and ‘text’. Default icons of ‘information’, ‘warning’, ‘skull’, and ‘toast’ are currently provided by the framework, but the framework user can supply their own icons. The ‘icon’ supports any HTML5-compatible image file, and the ‘text’ is the text that will appear in the message box. An example is shown in Figure 8.

5. FILES AND PROJECTS

The preferred mechanism for data input and output within the framework is formatted in JSON. This is not always practical for possibly large binary data. When a module execution is submitted, a unique run directory is created for its execution. Files associated with the submission are placed in this directory and made available to the executing module. These files come from the local file system for the Qt/C++ GUI-generated code and from the clients’ machine via an AJAX POST for the HTML5/PHP-generated code. Execution run on an SSH-accessible resource is currently available via a shared file system, but if a shared file system is not available, an additional transfer of files could be required. Airavata submissions do require an additional transfer to the destination compute resource.

Existing and legacy applications may require specific locations of certain input files and place output in files. These could be handled with custom wrappers, but one of the goals of this framework is to simplify usage for the application developer. Specifically, the existing SASSIE [3] code has cases where outputs from a previously executed module are expected to be available to a subsequent one. Therefore, we created a mechanism to utilize project names to assign fixed directories. Projects are only available to logged-in users. Sharing of a project directory among module execution opens up the possibility of multiple modules executing simultaneously; therefore, we serialize access to these directories for execution via a lock stored in the database preventing further submissions to the project. A message box warning (section 4.3) is produced if a project is locked when an end user attempts to submit. Projects locked for a user can be identified and managed in the job management interface (Section 6). The module definition JSON has an optional attribute to override the project directory and give each submission a unique directory as well as an option to ignore locking.

Collections of files in project directories can be downloaded via the file manager interface individually via links or in variously compressed tar or zip archives. An example of the HTML5/PHP file manager interface is shown in Figure 9.



Figure 8. Message box test message in the HTML5/PHP target language. The JSON inset is added to the UDP messaging or final output JSON to trigger the client to display the message box.

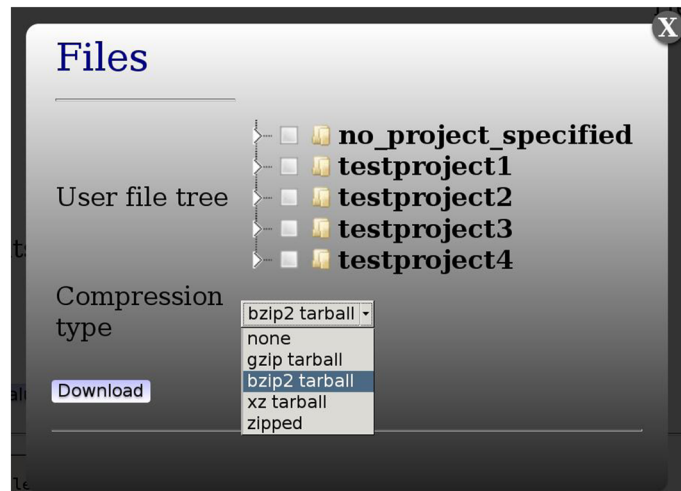


Figure 9. File manager interface. Each project directory is listed and expandable as a tree. Branches and individual files can be selected and then downloaded in various formats as shown. The 'none' compression type provides a list of file links for individual download.

6. JOB MANAGEMENT

The job manager allows the end user to monitor the execution and retrieve results of running and previously executed modules. When a module is executed, a GUID is assigned and is used as a key to store job status information in the database. Job statistics currently recorded include GUID, module name, menu entry, project name, user name, working directory, executable name, compute resource, start and end times, and an array of status (such as 'started', 'running', 'finished', and 'failed') with timestamps. Although all job statistics are recorded, only jobs submitted when a user is logged in will be visible via this job manager interface. The user is presented with a table of jobs previously submitted. The table includes the name of the module, the project, the start and end times, the duration of the job, the remote IP used to submit the job, and the execution resource. A screenshot of the job manager is shown in Figure 10, which shows the first four columns of data

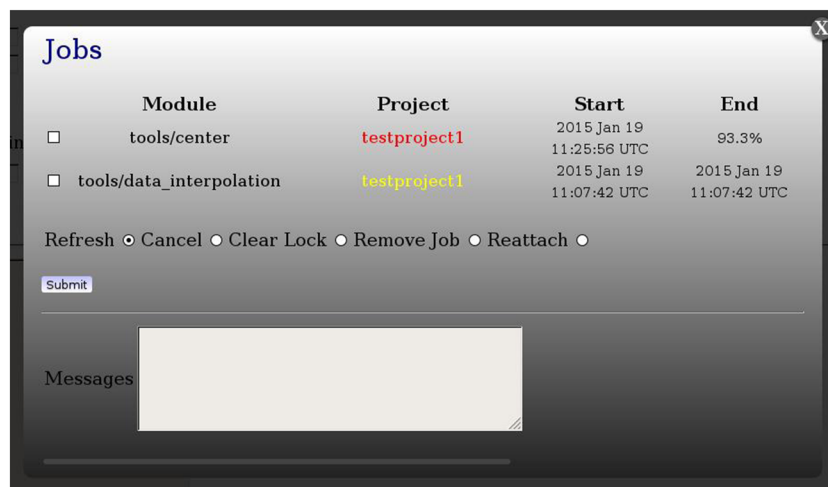


Figure 10. Job management interface. Each module submission is presented on a row of the table. In addition to the shown columns are duration of the job, the remote IP used to submit the job, and the execution resource. The project marked in red is locked (in this case because it is running). The project marked in yellow is also locked because another job is running under the project. By selecting a job or jobs, one can clear an existing lock, remove a job, or reattach to a running or completed job.

(the remaining columns are accessible via scrolling). The project in red is locked and running, the project shown in yellow is locked also but not running. Running jobs that provide progress updates (Section 4.2) will display the ‘_progress’ attribute value from the database as a percentage in the ‘End’ column. Options are provided in the job manager to ‘Cancel’, ‘Clear lock’, ‘Remove jobs’, and to ‘Reattach’ to a job. ‘Reattaching’ can be performed from multiple clients on various devices simultaneously to running or completed jobs.

7. FUTURE

GenApp currently generates HTML/PHP, Qt3/C++, and Qt4/C++ applications. Qt5/C++ is generated but has not yet been fully tested. Qt5/Android and Qt5/iOS and possibly JAVA are planned. New features are currently added on an as-needed basis as we build up our initial library of executable modules, initially in the SAS field. The user interface is based upon collections of typed fields as defined in the module definition file. We believe that advanced user interface capability definition is possible in the module definition file without requiring changes to the underlying module’s code, and these will be added as required. We are currently working on improvements to the file management capabilities. We envision an advanced input widget to allow selection of files from a combination of staged files and those present on the client workstation and possibly other URL documents. The Airavata integration is being updated to the current releases of Airavata and GenApp. These will then be tested with a recent eXtreme Science and Engineering Discovery Environment (XSEDE) allocation (TG-MCB140255). We also have a requirement to run on Department of Energy (DOE) resources and are looking at Airavata and Simple API for Grid Applications (SAGA) [17] as options.

8. CONCLUSIONS

The GenApp framework produces user interfaces for JSON wrapped modules generating GUI-based and web-based applications. A key feature of GenApp is the fact that GenApp is designed to be easily extended and adaptable to new target languages and environments without requiring modification of the wrapped executable module’s code. Although there are other frameworks to generate user interfaces, we are unaware of any that are designed to generate them in arbitrary target languages and execution environments. If an interesting future ‘target language’ or environment appears, we believe this framework is capable of generating code for it. The initial community consists of chemical and chemical biology researchers involved in SAS studies. The framework could easily extend to other scientific disciplines. GenApp’s web-based execution has been improved with better handling of AJAX calls and the ability to have multiple instances attach to running or view results from previously run jobs. New progress and message box support provide an improved user experience. The integration of Airavata provides access to a diverse set of computational resources. In addition to the intellectual contributions, the work described is a good demonstration of bringing computational science and open-source software experience to next-generation students.

9. RESOURCES

The software is currently stored on a subversion-integrated Trac Wiki (<http://trac.edgewall.org>) hosted on an Indiana University Quarry node <http://gw105.iu.xsede.org:8000/genapp>. A separate virtual machine containing multiple HTML5 application instances is hosted on another Quarry node. A 128-core, 256-GB-ram, 8-T K20m GPU cluster is installed at the University of Tennessee Knoxville dedicated to computations running under this tool. The Alamo cluster at the University of Texas Health Science Center in San Antonio will also make cycles available callable via Airavata. When usage demands, we will submit an XSEDE proposal for additional cycles to support the Science Gateways developed utilizing this tool.

ACKNOWLEDGEMENTS

This work was supported by NSF grant CHE-1265817 and XSEDE allocation TG-MCB140255 to E. H. Brookes and Google Summer of Code funding to N. Anjum. This work benefited from CCP-SAS software developed through a joint EPSRC (EP/K039121/1) and NSF (CHE-1265821) grant. J. E. Curtis acknowledges support from NIST. We thank the reviewers for their helpful comments.

REFERENCES

1. Brookes EH. An open extensible multi-target application generation tool for simple rapid deployment of multi-scale scientific codes. XSEDE'14. ACM. 2014. DOI: 10.1145/2616498.2616560
2. Perkins S. (Available from: <http://www.ucl.ac.uk/smb/perkins>).
3. Curtis JE, Raghunandan S, Nanda H, Krueger S. SASSIE: a program to study intrinsically disordered biological molecules and macromolecular ensembles using experimental restraints. *Computer Physics Communication*. 2012; **183**, 382–389. (Available from: <http://www.smallangles.net/sassie>).
4. Brookes EH. US-SOMO. (Available from: <http://somo.uthscsa.edu>).
5. Brookes EH, Singh R, Pierce M, Marru S, Demeler S, Rocco M 2012. UltraScan solution modeler: integrated hydrodynamic parameter and small angle scattering computation and fitting tools. XSEDE '12. ACM. DOI: 10.1145/2335755.2335839.
6. Marru S, Gunathilake L *et al.* 2011. Apache Airavata: a framework for distributed applications and computational workflows. Proc. Workshop Gateway Computing Environments. ACM.
7. Pierce M, Suresh Marru LG, Raminderjeet Singh DKW, Wimalasena C, C C. Apache Airavata: design and directions of a science gateway framework in Proceedings of the International Workshop on Science Gateways, Dublin, IE, June 3-5, 2014.
8. Standard ECMA-404. 2013 The JSON data interchange format. Geneva.
9. Papadopoulos PM, Katz MJ, Bruno G. 2001. NPACI rocks clusters: tools for easily deploying and maintaining manageable high-performance Linux clusters. In Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Y Cotronis, J Dongarra (Eds). Springer-Verlag, London, UK, 10–11.
10. Zhang H. Simulation of neutron data of intrinsically disordered proteins and nucleic acids: part II. ACNS 2014, Jun 1-5, 2014. Knoxville, USA. WORKSHOP.
11. Anjum N 2014. GSoC: *GenApp integration with Apache Airavata*. (Available from: <http://www.google-melange.com/gsoc/proposal/public/google/gsoc2014/nadeemanjum/5632763709358080>).
12. Apache Software Foundation, Thrift, (Available from: <http://thrift.apache.org/>).
13. ZeroMQ. (Available from: <http://zeromq.org>).
14. Ratchet. (Available from: <http://socketo.me>).
15. WebSocket. (Available from: <http://www.websocket.org>).
16. MongoDB. (Available from: <https://www.mongodb.org>).
17. Merzky A, Weidner A, Jha S, SAGA: A Standardized Access Layer to Distributed Computing Infrastructure, Software X, Elsevier: Cambridge, Mass, USA (accepted) 2015. DOI: 10.1016/j.softx.2015.03.001