The Center for Simulation of RF Wave Interactions with Magnetohydrodynamics (SWIM) Computational Framework

Randall Bramley Anne Faber Samantha Foley Yu (Marie) Ma

September 23, 2006

This documents the framework design and implementation for the SWIM project. Although one section is titled "usage" the framework is not yet ready for production use. Lee Berry is working on installing AORSA + CQL3D in the subversion repository and we will put together a working example with all of the parts as soon afterwards as possible. Section 4 has an example set of scripts for a non-parallel application, and those are downloadable from the SWIM site's SVN repository using the command

```
svn co https://cswim.org/svn/cswim
```

If your SWIM login is different from your local workstation login, use

```
svn co --username yourusername  https://cswim.org/svn/cswim
```

(Note: This will checkout all of the codes. See cswim.org on how to checkout just the code you want.)

1 Design Goals

Design goals were agreed upon at the start of the SWIM project. They include:

- Handle job management and monitoring on different platforms, especially unique DOE petascale ones
- Data managment, movement, location, and sophisticated query
- Minimal perturbation to ongoing scientific research using the extant constituent codes and software
- Maximize reliability of the systems codes
- Extensibility, programmability, and longevity
- Leverage existing technology and interoperate with other fusion simulation efforts

Implementation choices have been driven by these design goals. One choice has been to use a primarily *scripting* based framework system. This allows us to use different Graphical User Interface (GUI) and workflow composition systems while retaining a large base of intercomponent interactions defined by SWIM runs. Scripts have had greater longevity than GUIs, and some of the scripts used to coordinate and launch HPC runs in fusion simulations date back decades.

Another principle driven by the goals is the *decoupling* of parts of the SWIM framework as much as possible (this is actually a form of the software component dictum that required interfaces should be as minimal as possible). Foremost, no simulation should fail unnecessarily because some other component failed. A sophisticated data managment system is provided through the portal, but decoupling means it can be used independently of the portal - and some runs can be made without using the data manager at all. Of course in that case you lose the ability to track and automatically archive the files associated with a run.

Also as much as possible SWIM uses utilities that have proven reliable, are widely used, and which seem likely to continue to be maintained and developed by other projects. Included in this list are the Python language, the Gridsphere portal, and MySQL for data management underpinnings.

As another example, using Python (versus, e.g., Java) for the scripts was chosen because current DOE HPC platforms all support some form of Python, even if they do not have Java yet.

2 SWIM Framework and Utilities

The primary software design decision in SWIM was to use scripts whenever possible, instead of requiring the use of a particular technology. This does mean there are multiple levels and categories encompassed by the term "script" but it does let us re-use existing harness, configure, and run scripts that the different components already have available. A SWIM portal (Section 2.4) has been started, but for the most part a SWIM run can be made without using the portal at all - it is intended as a convenience and an organizer, not as a single required entry point.

The SWIM portal-based approach is consonant with what is being done in other national scientific collaborative efforts, and for job submission interfaces provided by some supercomputer centers. An overview of portals and the tradeoffs in choosing to use them are in Section 2.4, and the particulars for the SWIM portal are in Section 2.6.

Utilities that are near ready to be used are the job manager, the data manager, and the event system. Each needs more development however - in particular the authentication and launch part of the job manager are not yet implemented. Some utilities, like the data manager, are accessible both through the portal and independently via other interfaces (e.g., through Perl or "web services" interfaces). The job manager has two parts, one for submitting jobs via the portal and the other for negotiating and dealing with batch managers on HPC resources. The second one can be and is implemented via Python scripts rather than being embedded as part of the portal itself.

2.1 Data Management

Data management needs for SWIM are complex due to the nature of computation codes involved. Many codes have established some basic data staging pipelines and set up preferred I/O formats to use data management tools like MDSplus [2].

Predefined directory structures for running simulations can be used to keep track of data files directly. The Integrated Plasma Simulator (IPS) Design Description document proposes the following directory layout of control and input files in user space. How to set up and populate such a simulation directory is yet to be discussed.

```
IPS_run_XYZ
  |-- simulation_setup
    |-- system_config
    |-- simulation_wide_data
      |-- simulation_control_file (this is a file not a directory)
      |-- initial_plasma_state (t0) (this is a file not a directory)
      |-- machine_definition (may be empty for now)
      |-- simulation_events_waveforms
      (scheduled events, source and control waveforms)
    |-- component_inputs
      |-- RF
        |-- RF_component (the component may need generic files)
          |-- RF_config
          |-- RF_required_list
          |-- RF_outfile_list
        |-- code_inputs (code specific files e.g. AORSA)
          |-- AORSA_required_list
          |-- AORSA_outfile_list
          |-- Standard AORSA input files  ...
      |-- FokkerPlanck
        |-- FP_component
          |-- FP_config
          |-- FP_required_list
          |-- FP_outfile_list
        |-- code_inputs (code specific input files e.g. CQL3D)
          |-- CQL3D_required_list
          |-- CQL3D_outfile_list
          |-- Standard CQL3D input files  ...
      ...
      |-- <other components> ...

  |-- simulation_results
    |-- history
      |-- t0 (identifier in milliseconds)
        |-- plasma_state
        |-- components
          |-- RF
          |-- Fokker Planck
          ...
          |-- <other components>
      |-- t1 (identifier in milliseconds)
        |-- plasma_state
        |-- components
          |-- RF
          |-- FokkerPlanck
```

```
        ...
        |-- <other components>
    ...
    |-- tN ...<other time steps>
  |-- restart
    |-- t0 (identifier in milliseconds)
      |-- RF
        |-- input_files
        |-- internal_state
      |-- Fokker Planck
        |-- input_files
        |-- internal_state
      ...
      |-- <other components>
    |-- t1 (identifier in milliseconds)
      |-- RF
      |-- Fokker Planck
      ...
      |-- <other components> ...
    ...
    |-- tN ...<other time steps>

|-- work
  |-- plasma_state (current plasma state tn)
  |-- RF
    |-- RF_component (the component may need generic files)
      |-- RF_config
      |-- RF_required_list
      |-- RF_outfile_list
      |-- RF_log
    |-- code_inputs (code specific input files e.g. AORSA)
      |-- AORSA_required
      |-- AORSA_outfile_list
      |-- AORSA standard input files ...
    |-- RF_component_script (?)
    |-- executable (?)
    |-- code_outputs
      |-- AORSA standard input files ...
      |-- AORSA_log
    |-- scratch
  ...
  |-- <other components> ...
```

EventMD	
EventID	Unique event ID
topic	Event topic as required by the event channel
ts	Timestamp of when the event is published
publisher	User ID of who published the event
component	Computation component ID which published the event
host	Machine host name from where the event is published
action	Progress notification or computational request
message	Additional arbitrary user messages

Table 1: FSP Event Metadata

A comprehensive predefined directory structure can be straightforward for locating data files of a particular simulation run, but could become cumbersome as the data volume grows. Supplementary metadata management based on Relational DataBase Management System (RDBMS) techniques is beneficial in this case, where more sophisticated search queries and more flexible data storage schemas are supported. Additionally, new data sharing and manipulation requirements continue to emerge as the community framework evolves. For example, when different computational components are combined to work together and interact with each other under a distributed environment, input and output files are often shared and duplicated among them, and likely in groups. The ability to define an aggregate of files, and subsequently archive, locate, and retrieve them individually or as a bundle, is beyond the scope of data management tools already in use. Moreover, external notification mechanisms like event systems will be deployed in the framework for different components to communicate with each other on information like computation status and file operations, a desired data management system should in turn interact with such an external system for not only information exchange but also logging purpose.

For the system flexibility and extensibility, a customized data management solution for SWIM is developed under Obsidian [8] composable architecture. This data manager mainly subscribes to the event channel and responds accordingly to various data management requests described in event messages. In addition, all related event messages are archived in the back-end MySQL RDBMS for logging purpose, together with Event specific metadata as defined in Table 1, which are available as query conditions for looking up particular events.

Two initial types of data management requests have been identified as requiring further actions from the data manager: one for archiving physical metadata of computation component I/O files, the other for aggregating files in collections as specified by users. Once every file of computational significance has been recorded in the metadata database through Obsidian *Physical Location Tracker*, arbitrary aggregates of these files can be defined through *Logical Collection Manager*, though most commonly according to job executions of computation components. As an example depicted in Figure 1, several input and output files are involved in two runs of the same computation. Two file collections can then be respectively defined for those from each run, enabling later queries to find basic metadata about all files in the same run as a given file. A web service interface has been provided by the SWIM data manager in WSDL descriptions to enable interactions between the Portal and the back-end metadata database. However, A SWIM user does not *have* to use the data manager and can use another data management method like directory/file naming

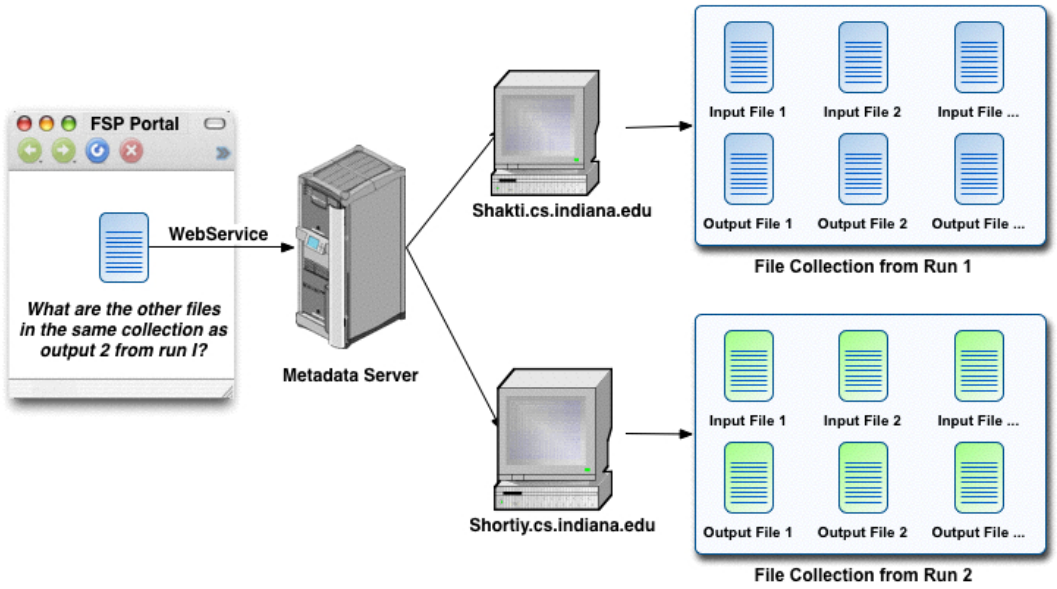

Figure 1: FSP Example of Defining I/O File Aggregations

conventions in place or along with it.

The data manager is used for internal information management in the portal but end users need not deal with that. In general three types of metadata are defined:

1. Notifications and events published by any component or portal utility
2. File announcements and creation. Metadata includes name, location, user ID of the file owner, which component created it, timestamp, purpose of the file, and a SWIM job ID.
3. Aggregates of files. It is convenient at times to deal with a collection of files such as "all output files from a given SWIM run", instead of each file individually. Aggregates can be overlapping with a single file belonging to several at the same time. So "aggregate" does not imply the collection is a directory or even on the same machine.

Portal pages have been written that use the event data management to display published events of types notification, data, and job management dynamically. Additionally there is a portal page with a simple GUI that allows users to enter complex queries, such as a request to view all files belonging to user bramley created over the past week, which were used as inputs to an RF component. If a type of query seems likely to be common it can be added it as a predefined one, available to single users or the whole project (via the customization capability of portals).

Although object storage is a more modern and flexible concept, currently all the codes and components targeted by SWIM are file-based: consuming input and name-list files, producing output files. Files are a familiar concept to most users. SWIM provides some utilities to help coordinate, track, and locate files. Even without distributed computing, running multiple codes as part of an integrated simulation requires file management beyond what is currently done by the individual codes. The file specification lists in **SWIM_component_required** and **SWIM_component_results**

consist of one file per line (which can be extended to allow wildcarding later). With each file listed is an action to take: notify, save, or none. As the names suggest, the corresponding actions are

- *notify*: the framework will publish an event that the portal and other subscribers can use and act upon it - e.g., start a component that requires the file to exist as a prerequisite.
- *save*: performs a notify action, and additionally signals the data manager to archive the file.
- *none*: no notification or action is required. The file may be listed because it is needed before a component can run.

2.2 Job management

Job management has three parts, which are decoupled and implemented in scripts in keeping with the SWIM overall software approach. The simplest is a user interface for monitoring, and that is part of the SWIM portal's event monitoring. The second part is the portal part that takes a specified component (which at this stage are separate executables) and launches it on a specified machine using the user's credentials for authentication. This part has been reused from the National Fusion Grid Collaboratory, and is probably the most onerous one for end users to get set up - but for current NFGC users SWIM will re-use their existing grid credentials.

The third part of job managment is the negotiation with (possibly remote) batch management systems. The batch management script **batch_mgmt_script.py** listed in Appendix 3 does this using a Python class *fsp_job*. The class creates *fsp_job* objects which provide a common interface to PBS, SLURM, or Unix processes. This shields users from the details of exactly how to run a program on a specific machine. More importantly it provides a mechanism later on for us to submit a series of executions without having to wait through the batch queue for each one separately. Note that this part is *not* yet implemented, but the batch manager negotiator can be upgraded to it without requiring changes to the rest of the SWIM software.

The batch negotiator includes active monitoring, checking with the batch manager (or host OS) for the status of a job and publishing it via the event system back to the SWIM portal - or to another event channel subscriber.

Information about the system the job is running on, what kind of job it is, the resource requirements of the job and exact script to run are necessary for job launch. This information is stored in the **SWIM_config** file as a text file of name-value pairs. It is then parsed by the batch negotiator and run according to the given parameters.

The batch negotiator allows running different codes on various systems just by changing a few parameters in the **SWIM_config** file. It also allows configuring the codes to run with different versions of MPI or with different batch managers if they are available on one machine but not another.

A detailed description of the batch negotiator, its data fields, and design follows but most users can safely skip over them.

- Data member, *my_vals*, is a dictionary type that stores the following data:
  - *batch_mgr* : can be SLURM, or PBS, if something else is specified the job will be run as a regular program. Programs are run differently by different batch managers. For

example, the command to submit a job to the queue is *srun* in SLURM, and *qsub* in PBS.
  - *num_nodes* : this is only used when *batch_mgr* is set to SLURM or PBS. It is needed to specify the number of nodes needed to run the program at submission time.
  - *executable* : this is the name of the script file to run. The script file must contain the full executable statement. For example: "ls". It is recommended to use full paths.
  - *jobid* : this will be set when the job is submitted. It is used to monitor and cancel the job.
  - *status* : the status is set at runtime and is updated by the monitor function. There are five states that are used:
    * done : this indicates that the job no longer needs to run. In the *__init__* function the state is set to done. This corresponds to SLURM's CD and TO states, PBS's E state, and the UNIX command *ps*'s X and Z states.
    * trans : this indicates that the job is in some kind of transition. This corresponds to SLURM's CG state, PBS's T state, and the UNIX command *ps*'s T state.
    * failed : this indicates there was some sort of failure that occured while the job was running. It only corresponds to the SLURM states F and NF.
    * waiting : this indicates that the job is waiting to use resources, sleeping or in the queue. This corresponds to SLURM's PD state, PBS's Q, W, S and H states, and the UNIX command *ps*'s W, S and D states.
    * running : this indicates that the job is actually running. This corresponds to the R state in all implemented batch managers and UNIX command *ps*.
  - *mpi_job* : this allows us to do special handling for mpi jobs as opposed to serial jobs.
  - *Jpype* : this is the location of Jpype on the system. Jpype is needed to send events to the portal.
  - *event_channel* : this is the location of the event channel. It is needed to publish events that can be picked up by the portal.

- Member functions: *submit_job*, *monitor_job*, *remove_from_q/kill_job*, *__init__*. All the functions can print debugging information to standard output, in addition to, publishing events.

  - *__init__* : reads the configuration file (**SWIM_config**), stores the data in *my_vals*, sets the Jpype location, and connects to the event channel.
  - *submit_job* : submits the job to the queue or runs the job (if no *batch_mgr* is specified). How, where and what is run depends on the values in *my_vals* that was set up in the *__init__* function. The *job_id* is set at this point to allow the script to monitor and cancel the job automatically.
  - *monitor_job* : this function will run continuously until the job finishes. It queries the batch manager or OS for the status of the job. The *job_id* is typically used to query for the status. An event with the state and CPU time used is published at regular intervals. The interval can be sent in via a paramter in the framework, or the default will be used.
  - *remove_from_q/kill_job* : these functions will remove the job from the queue if it has not run yet or kill the job if it is currently running. If the job has finished, the function will not do anything. *kill_job* just calls *remove_from_q*.

Table 2 shows the correspondances among the states as defined by the different tools. "PROCS" simply uses Unix processes rather than a batch manager, and SWIM is the provided batch negotiator.

SWIM	SLURM	PBS	PROCS
done	CD,TO	E	X,Z
trans	CG	T	T
failed	F,NF		
waiting	PD	Q,W,S,H	W,S,D
running	R	R	R

Table 2: Possible Run States for a SWIM Component

Three Python dictionaries translate native batch manager states to a uniform SWIM state identification:

- SLURM_STATES = {"CD":"done", "TO":"done", "CG":"trans", "F":"failed", "NF":"failed", "PD":"waiting", "R":"running"}
- PROC_STATES = {"X":"done", "Z":"done", "T":"trans", "W":"waiting", "S":"waiting", "D":"waiting", "R":"running"}
- PBS_STATES = {"E":"done", "T":"trans", "Q":"waiting", "S":"waiting", "W":"waiting", "H":"waiting", "R":"running"}

2.3 Event channel and decoupling

One of the design goals is robustness of the framework facilities, and SWIM decouples utilities as much as possible. The main mechanism is using an "event channel", a daemon service running at a well-known location. Even different portlets running within the same portal try to use the event channel whenever possible, instead of internal portal mechanisms.

The SWIM project uses the WS-Messenger (WSMG) [7, 9] web service, developed by the Extreme Lab at Indiana University, as an event channel. Supporting both the WS-Eventing and WS-Notification specifications, WSMG is a publish-subscribe system that allows components or portal utilities to publish notifications about events that occur during a run of a code. A notification consists of a message and a topic. To receive notifications, a utility must subscribe to a specific topic. This is a many-to-many relationship, as a subscriber can subscribe to many different topics and a publisher's notifications may be consumed by many different subscribers.

Using a publish-subscribe event channel like WSMG allows us to decouple the event handling from the actual running of codes. WSMG runs on a separate "broker" machine and maintains a database of current subscriptions. Publishers send events consisting of a message and a topic to the broker, which then sends the events to all subscribers interested in that particular topic. The publisher does not need to know which, if any, subscribers are receiving its events.

WS-Messenger is currently being used in the Linked Environments for Atmostpheric Discovery (LEAD)[4] project to allow remote applications to send notification messages to each other.

2.4 General portal ideas and terminology

The term *portal* originally was applied to job launch facilities provided by the San Diego Supercomputer Center and NCSA [1, 3]. Portals are now used for job launch at remote facilities,

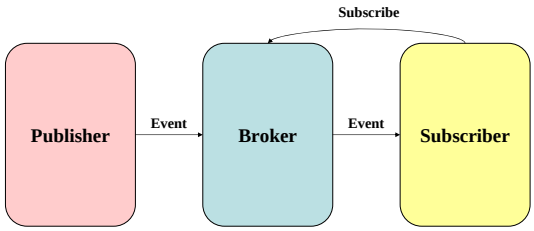

Figure 2: Event Channel

coordination and collaboration within specific application areas like climate [**?**] and weather [4] modeling, monitoring scientific instruments [5], and managing large scale data sets.

A portal is a web-based service running at some well-known location, which is generally independent of where clients are running, managed resources are located, or large scale scientific data sets are located. A user can launch a set of jobs on a portal, logoff, and then reconnect later from a different machine to check results or continue the session. A distinguishing feature of portals is *customization*, meaning each user can arrange the interface for their own interests and the server will maintain that information across sessions for the user. Unlike most web services, that user-dependent information is stored on the server side rather than in cookies on the user's client machine. Customization can include what material to view, how the user interacts with the interface, and most other aspects - but in practice most portal users simply use the default.

Certain services that run within a portal "container" are called *portlets*, and they have access to the information stored in the container. Part of customization typically includes which portlets a user starts by default. Portlets are written in Java and have to implement a container-specified portlet interface, so that the portal can interact with them uniformly. An end user generally should not need to write or edit portlets, and instead new ones are added by the portal maintainers. Adding a portlet requires some restart of the web server hosting the portal, a heavy-weight task.

2.5 Portal advantages and disadvantages

Using a portal allows reuse of utilities developed across a broad range of application areas. One of the most frequently used portlets is *MyProxy*, [**?**] which on logon holds a user's authentication credentials and can present them to other (potentially remote) services on a user's behalf. This is necessary because as a Unix process the portal is running as some a virtual user, and several real users may be logged in simultaneously. *MyProxy* holds "proxy certificates" and so does not need to actually store a user's password on the portal host.

While *MyProxy* handles the problem of authentication, *authorization* (determining which users can access which resources) is a separate and more difficul problem, with several different solutions in the portal world.

Portals provide centralization, so a user's full work arena is available from any site. This

avoids the problem of having different utilities, codes, and working resources scattered on different machines, but does require the portal to run on a reliable server with good network connectivity. Some portal implementations use techniques from commercial web servers, using load-balancing and fail-over so that if one server host fails another can seamlessly take up the load. SWIM is unlikely to encounter the scalability problems that some portals have (thousands of simultaneous users) so load balancing is unlikely to be a problem. Another advantage is that portal technology is being developed by several CS research groups and companies, letting us use future upgrades and utilities as they are developed.

Disadvantages of using a portal approach, and how we are trying to mitigate them, include

- Authentication with remote resources is via "grid certificates", the grandchild of Globus certificates. These are X.509 certificates. SWIM will automatically generate the proxy certificate so that a user can login once per session and only have to provide a password once. But a preliminary step is needed to have a grid certificate created for the user. Grid certificates are accepted at PPPL for the TRANSP and collaboratory projects, and are being used as part of ORNL's Teragrid participation.

- Centralization of services potentially could lead to not being able to do any work if the portal host is down or inaccessible. We currently have fail-over capability between two machines at IU, located at Indianapolis and Bloomington for geographic dispersion. Both servers are on at least three major wide area networks (iLight, AT&T, ESNet, Abilene). But being able to do fail-over among machines located under different administrative domains is still an open research issue, which IU is working on in the context of other portals.

- Centralization also means the failure or loss of one utility (portlet) could bring down all the other portlets. We are addressing this by using an *event channel* for decoupling as much as possible (see Section 2.3).

- Portals entail a lot of heavy-weight machinery, especially when a user just wants to do some quick debug cycling or exploratory runs. This is mitigated partly by the multilevel scripting approach. The plan is to be able to run any constituent code as a component independently of any other, and even without access to the portal. Whenever a component script makes a call to a portal utility like publishing a message to the event channel, if the portal is not accessible the script falls back to a local mechanism like writing a log file. The portal should be seen as a set of utilities that help compose, run, and diagnose SWIM runs - but it is possible to make such runs even without using the portal.

- A lot of work and effort is being put into a computer technology that may turn out to be a flash in the pan and outdated within a few years. Mitigating this is the support of portals in commercial settings by major companies like IBM, Sun, and lately MicroSoft, and in HPC scientific settings by the NSF, NIH, and DoE. The multi-level scripting approach also helps and in principle at least each portlet can be converted to the next generation container/utility technology. E.g., portlets were preceded by "servlets" in Java, and there are automatic utilities to convert a servlet to a portlet. As another example, the data manager portlet uses an underlying MySQL database system and also provides scripting and "web services" interfaces, so running it independently of the portal is possible (and is done routinely in the CIMA project).

Overall, the plan is to provide the useful utilities that portals have but to not have the project tied to that particular technology. Portals have been around for only about 12 years now, while some control and scientific application management scripts have been around for 40 years.

2.6 SWIM portal implementation

SWIM is using the Gridsphere portal system [**?**], because it seems to be the most widely used portal in distributed scientific computing. The implementation currently includes fiveite utilities or portlets. A standalone *event channel* [9] is used to help decouple different utilities and the various component runs required.

2.6.1 Current Portlets

MyProxy. As described earlier, this handles authentication of users: after logging in and providing a portal password, under the covers myproxy generates a "grid certificate" for the user and that is used to launch jobs and transfer data on behalf of the user. A user should never have to deal with or even be aware of this portlet!

Job Manager. The portal remotely launches jobs using the batch management script. The launch and monitoring of the job is taken care of by the script automatically and uses a configuration file (**SWIM_config**) for specific information about how to run the code. While the code is running, events are published to the portal for the user to view. Killing a job is also possible from this utility.

Data Manager. This portlet is responsible for locating and moving files as needed for a run. It automatically generates metadata and stores it in a relational database, allowing users to perform queries. Details are in Section 2.1.

2.6.2 Event Viewer

The SWIM portal has pages for viewing events (see Section 2.3) as they are published. Separate pages are provided for each type of event so that a user does not get information overload with messages appearing on the web page. Messages are "pushed" to the portal page, appearing without user action required. Because events are stored by the data manager, a user can also do filtering on the messages and request, e.g., only those from a particular user or machine. This kind of query uses a "pull" approach, that is, requires a user to enter the request or refresh the page to update it.

2.6.3 Overall script

The major task of the SWIM portal is to run the IPS script that specifies which components to run, performs time looping, and makes decisions based on physics or other criteria. Although the portal launches and runs this overall management script (and the data manager archives it), that does not imply or require that the script run on the portal host. Just what needs to go into this script is specified in the IPS Design document written by Don Batchelor, and is still evolving. The general principle is to keep individual code component run scripts free of SWIM-specific actions and ideas, and concentrating those actions into the overall script.

It can be edited directly on the portal page, or (a more likely usage) edited locally and uploaded to the SWIM portal. The first case is more likely for quick debugging and experiments since web interfaced editors generally are poor.

3 SWIM Framework Usage

Creating a component and using it in a SWIM run is straightforward. The functions which the framework requires and the use of the Plasma State Component for passing data between components is detailed in the IPS Design Document [**?**]. Here the basic directory and scripting requirements are given. Three steps are required, but most computational codes have the more difficult parts (run scripts and build structures) already done independently of SWIM. Steps are:

1. Put each code into a separate directory. In the LSA example in Section 4, these are named after the component they contain and are all under the *lsa/* directory.
2. Create five files in the component's directory, with the following convention:

   (a) **SWIM_component_required** a list of the required input files. This should include ones that are component-specific, i.e., ones that are not strictly related to intercomponent interactions. That is so the framework can assure all required run-time inputs are available before potentially wasting time in batch queues or otherwise.

   (b) **SWIM_component_results** a list of the output files. This is actually not used until the end of the run, because a component may not know a priori how many or even what files will be created. E.g., a file may be created for each time step, and the number of timesteps is not known until convergence has been achieved or an error tolerance met.

   (c) **SWIM_files** the list of files that the SWIM framework needs for correct operation. For the LSA example, this includes just the five files in this list.

   (d) **SWIM_config** a file containing any additional information required to run the component. That includes application and site-dependent information such as the project directory, the batch management system (PBS, SLURM, sshd), whether or not the job is parallel, and the number of MPI processes to use. This can be expanded for any other information the user needs.

   (e) **SWIM_runit** a Python script that actually encapsulates the component and makes calls on the framework.

Once all of these files are in place and the code is built, the SWIM framework does the following to run the job:

1. job launch script moves any input files that need to be moved and checks to make sure the files are in the right spots.

   (a) **SWIM_runit** is called from job launch script.

      i. in **SWIM_runit**, the run is initialized using the *initialize* function (from **Framework.py**):

         A. check for a valid executable

Name	Type	Inputs	Outputs
BasicInfo	Informational	mat.in	mat.html + GIF files
Scale	Filter	mat.in, methods	mat.out, summary.html
Reorder	Filter	mat.in, methods	mat.out
Splib	Solver	mat.in, methods	results.html, PNG files
SuperLU	Solver	mat.in, methods	info.html

Table 3: Sample LSA Components

B. create a new directory for the run with specified input files and executable

ii. **SWIM_runit** calls step (from **Framework.py**):

A. *job*, a *batch_mgmt_script_fsp_job* object is created to run job

B. *job* is submitted to the queue via *submit_job*

C. *job* is monitored via *monitor_job*, status events are published at regular intervals until the run has ended

iii. **SWIM_runit** calls finalize (from **Framework.py**):

A. data manager is notified of output files created during this run via event channel

4 Example: The Linear System Analyzer

As a brief example of how an application can be assembled and run in the SWIM framework, a set of serial codes have been wrapped up as separate executables. The codes are a subset drawn from the Linear System Analyzer (LSA) [6], an early component-based problem-solving environment for developing solution strategies for sparse linear systems of equations. The modules that have been adapted are in Table 3. Each reads in a sparse linear system's coefficient matrix and right hand side vector in "coordinate-wise" format from a file called *mat.in*. Most also take an input parameter file called *methods*. Three general categories of LSA components have been identified based on their dataflows. A **filter** component modifies the linear system and produces another one in mat.out. An **informational** component gives data such as the number of nonzeros, a measure of symmetry, etc. A **solver** (tries to) solve the linear system, and also produces some information garnered about the linear system during the solve - the main purpose of the LSA.

Several other solvers and filters are possible and were in the original LSA, but these are sufficient to put together simple workflows. An example is in Figure 3, where the moving of one component's output file to another's input file is shown by an arrow. That example workflow would allow a user to compare the effects of scaling and not scaling the coefficient matrix, for both an interative (Splib) and sparse direct (SuperLU) solver.

The LSA is not a high-performance system and all of the codes are serial. The intent is to allow comparison of algorithms and preconditioning, not for scalable parallelism. It has proven useful

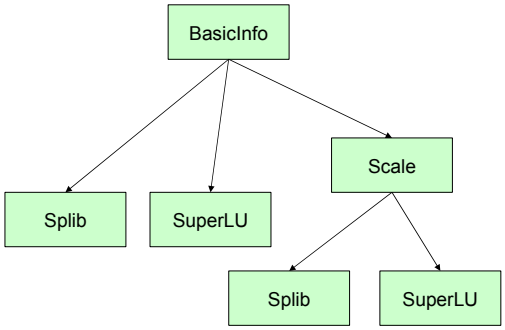

Figure 3: Example LSA workflow

as a guide in choosing a HPC solution strategy (filtering, preconditioning, solver method) but does not supply or replace one.

The LSA provides a simple example of scripting an application using Python and for using framework services.

A Example Component Setup: Scale in LSA

The example LSA application has components for BasicInfo, Scale, Reorder, Splib, and SuperLU. For the Scale component, the contents of the five necessary SWIM files is:
SWIM_component_required:

```
Scale   = none
methods = save
mat.in  = notify
```

SWIM_component_results:

```
mat.out         = save
summary.html    = save
```

SWIM_component_files:

```
SWIM_files                  = none
SWIM_component_required     = none
SWIM_component_results      = none
SWIM_config                 = none
SWIM_runit.py               = none
```

SWIM_component_config:

```
Jpype = san/fsp/lib/python
batch_mgr = ssh
executable = ./Scale
num_nodes = 1
```

Recall that for the file listings, none means take no action and save asks the framework data manager to archive and track the file. The component execution file **SWIM_runit** for Scale is in Listing 1.

Listing 1: SWIM_runit file for Scale component in LSA

```
#-----------------
# Global modules
#-----------------
import os, sys
from time import time, ctime
#-----------------
# Local modules
#-----------------
sys.path.append('../PythonUtils')
from FSPutils import publish_event

import Framework
```

```
# Allows for debugging print statments when requested
debug_on = False

if len(sys.argv) > 1:
    if sys.argv[1] == "-SWIM_debug":
        debug_on = True

Scale_services = Framework.Services()

#------------------------------------------------------------------
# Set up a subdirectory to run the component in, and put required
# files within that directory.  General initialization.
#------------------------------------------------------------------

newdir = Scale_services.initialize('Scale', debug = debug_on)

#--------------------
# Run the component
#--------------------

Scale_services.step('Scale', newdir, debug = debug_on)

#-----------------------------------------
# Clean up and finalize the component
#-----------------------------------------

Scale_services.finalize('Scale', newdir, debug = debug_on)
```

In red are the interactions with the Framework: first a "framework services" object is created for Scale, then it is used to initialize, step (run), and finalize. Part of initialization is to create a new subdirectory to run this particular instance in and then that is used in calls to step and finalize. Most scripts should be almost this simple. An example of one that does additional processing is Splib, which generates several graph files. After calling finalize, Splib has the code fragment in Listing 2 to read a list of graph files, and use *matplotlib* to create PNG files of them.

Listing 2: Splib extra code for creating graph files

```
#-------------------------------------------------------------------
# Postprocessing local to this component. This could be done by subclassing
# the services object, but this shows it's not necessary to follow the exact
# same run script for each component. This processes some xgraph files made
# of the convergence of iterative solvers, by converting them for the python
# package matplotlib and then using it to create PNG images of the graphs.
#-------------------------------------------------------------------
try:
    import matplotlib
    matplotlib.use('Agg')
    try:
        from pylab import *
```

```
        try:
            os.chdir(newdir)
        except:
            print '*** Cannot chdir to working subdirectory'

        try:
            import cvt
        except:
            print '*** cvt not found on import'

        try:
            graph_listing = open('graph_files', 'r')
        except:
            print '*** listing of files to graph not found'

        for line in graph_listing.readlines():
            cvt.cvt(line[:-1])
        graph_listing.close()
    except:
        print 'Cannot convert plotfiles to PNG'
        print 'error msg:', sys.exc_info()[0]
except:
    print 'Cannot convert plotfiles to PNG'
    print "error msg returned was ", sys.exc_info()[0]
```

B Batch Manager Script

The batch manager negotiator script described in Section 2.2:

Listing 3: batch_mgmt_script.py

```
"""
---------------------------------
States and their correspondances.
---------------------------------

  SWIM   |  SLURM  |      PBS      |  PROCS
-----------------------------------------
  done   |  CD,TO  |   E           |  X,Z
 trans   |   CG    |   T           |  T
 failed  |  F,NF   |               |
waiting  |   PD    |   Q, W, S, H  |  W,S,D
running  |   R     |   R           |  R

"""

import sys
import os
import time
import subprocess as sp
import Events

SLURM_STATES = {"CD":"done",
 "TO":"done",
 "CG":"trans",
 "F":"failed",
 "NF":"failed",
 "PD":"waiting",
```

```
 "R":"running"}

PROC_STATES = {"X":"done",
 "Z":"done",
 "I":"trans",
 "W":"waiting",
 "S":"waiting",
 "D":"waiting",
 "R":"running"}

PBS_STATES = {"E":"done",
 "I":"trans",
 "Q":"waiting",
 "S":"waiting",
 "W":"waiting",
 "H":"waiting",
 "R":"running"}

class fsp_job:   #our new job class
    # A dictionary containing configuration information read from the config file
    my_vals = { "batch_mgr" : "",
                "num_nodes" : 1,
                "executable" : "hostname",
                "jobid" : "0",
                "status" : "done",
                "mpi_job" : False,
                "Jpype" : "/usr/bin",
                "event_channel" : "shortly.cs.indiana.edu:12345",
                "utilpath" : "."
                }

    def parse_config (self):
        try:
            # This assumes that the config file is in the current dir
            config_file = open("SWIM_config", "r")

            if config_file:
                line = config_file.readline().strip()
            else:
                line = ""

            while line:
                name, val = line.split("=")
                name = name.strip()
                val = val.strip()
                if self.my_vals.has_key(name):
                    if name == "num_nodes":
                        self.my_vals[name] = int(val)
                    elif name == "mpi_job":
                        self.my_vals[name] = bool(val)
                    elif name == "Jpype":
                        self.my_vals[name] = os.path.expanduser(val)
                    else:
                        self.my_vals[name] = val
                else:
                    print "unrecognized name in name-value pair: %s = %s\n" \
                        % (name, val)
                if config_file:
                    line = config_file.readline().strip()
                else:
                    line = ""

            config_file.close()
        except Exception, ex:
            print "problems parsing config file: %s" % ex


    #states = something.... see notes above
```

```
def submit_job(self, debug_on = False):
    # submits job to batch_mgr, if one is present
    try:
        if self.my_vals["batch_mgr"] == "SLURM":
            if self.my_vals["mpi_job"]:
                # "sam_run" calls mpi_run.py if it is an mpi job
                cmd = "srun -N %d -b sam_run" % self.my_vals["num_nodes"]
            else:
                # Grab nodes, run, and grab stderr to get the jobid
                cmd = "srun -N %d -b %s" % (self.my_vals["num_nodes"], \
                       self.my_vals["executable"])   #assume it has ./ if needed

            #using subprocess
            p1 = sp.Popen(cmd, shell=True, stderr=sp.PIPE)
            s = p1.stderr.read()
            g = s.split()
            self.my_vals["jobid"] = g[2]
            self.my_vals["status"] = g[3]

        elif self.my_vals["batch_mgr"] == "PBS":
            scriptfile = open("scriptfile", "w")
            scriptfile.write("#!/bin/tcsh\n")
            scriptfile.write("#PBS -l ncpus=" + str(self.my_vals["num_nodes"]) + "\n")
            scriptfile.write("#PBS -l walltime=00:05:00\n")
            scriptfile.write("#PBS -l mem=50mb\n")
            scriptfile.write("#PBS -r n\n")
            scriptfile.write("#PBS -N FSPjobname\n")
            scriptfile.write("#PBS -q sgi\n")
            scriptfile.write("#PBS -V\n")
            scriptfile.write("cd " + os.getcwd() + "\n")
            scriptfile.write(self.my_vals["executable"] + "\n")
            scriptfile.close()
            #cmd = "qsub -l nodes=" + self.my_vals["num_nodes"] + " scriptfile"
            cmd = "qsub scriptfile"
            os.system("pwd")
            p1 = sp.Popen(cmd, shell=True, stderr=sp.PIPE, stdout=sp.PIPE)
            s = p1.stdout.read()
            g = s.split(".")
            self.my_vals["jobid"] = g[0]

        else:
            #just run
            # need to get pid... storing it in jobid should be fine
            cmd = self.my_vals["executable"]

            #new version using subprocess
            #print "starting the script"
            p1 = sp.Popen(cmd, shell=True)
            #print "done running script"
            self.my_vals["jobid"] = str(p1.pid)

        if debug_on:
            print cmd
            print "my jobid is: %s" % self.my_vals["jobid"]

        message = "job %s started" % self.my_vals["jobid"]
        Events.publish_event(message, topic='FSP_job')
    except Exception, ex:
        print "submit_job failed with exception %s" % ex

def monitor_job(self, interval=1, debug_on = False):
    # checks the status of the job
    try:

        if self.my_vals["batch_mgr"] == "SLURM":
            cmd = "squeue -j " + self.my_vals["jobid"] + " -h -o \" %.2t %.10M\" "

            #new: the squeue command prints to stdout and stderr
```

```
            #extract status from output
            p1 = sp.Popen(cmd, shell=True, stdout=sp.PIPE, stderr=sp.PIPE)
            out = p1.stdout.read()
            while out:
                s, t = out.split()
                s = s.strip()
                t = t.strip()
                self.my_vals["status"] = SLURM_STATES[s]
                message = "The status of job %s is %s. -- The cpu time is %s."  % \
                    (self.my_vals["jobid"], self.my_vals["status"], t)
                if debug_on:
                    print "monitor:\n" + message
                Events.publish_event(message,topic='FSP_job')
                time.sleep(interval)
                p1 = sp.Popen(cmd, shell=True, stdout=sp.PIPE, stderr=sp.PIPE)
                out = p1.stdout.read()

        elif self.my_vals["batch_mgr"] == "PBS":
            cmd = "qstat -r %s" % self.my_vals["jobid"]

            #extract status from output
            p1 = sp.Popen(cmd, shell=True, stdout=sp.PIPE, stderr=sp.PIPE)
            out = p1.stdout.read()
            while out:
                lines = out.split("\n")
                info = lines[5].split()
                s, t = info[9], info[10]
                s = s.strip()
                t = t.strip()
                self.my_vals["status"] = PBS_STATES[s]
                message = "The status of job %s is %s. -- The cpu time is %s."  \
                   % (self.my_vals["jobid"], self.my_vals["status"], t)
                if debug_on:
                    print "monitor:\n" + message
                Events.publish_event(message,topic='FSP_job')
                time.sleep(interval)
                p1 = sp.Popen(cmd, shell=True, stdout=sp.PIPE, stderr=sp.PIPE)
                out = p1.stdout.read()

        else:
            cmd = "ps -o s,cputime --pid %s --no-heading" %  self.my_vals["jobid"]

            p1 = sp.Popen(cmd, shell=True, stdout=sp.PIPE, stderr=sp.PIPE)
            out = p1.stdout.read()
            while out:
                s, t = out.split()
                s = s.strip()
                t = t.strip()
                self.my_vals["status"] = PROC_STATES[s[0]]
                message = "The status of job %s is %s. -- The cpu time is %s."  % \
                    (self.my_vals["jobid"], self.my_vals["status"], t)
                if debug_on:
                    print "monitor:\n" + message
                Events.publish_event(message,topic='FSP_job')
                time.sleep(interval)
                p1 = sp.Popen(cmd, shell=True, stdout=sp.PIPE, stderr=sp.PIPE)
                out = p1.stdout.read()

        self.my_vals["status"] = "done"
        message = "The status of job %s is %s." % \
                  (self.my_vals["jobid"], self.my_vals["status"])
        if debug_on:
            print "monitor:\n" + message
        Events.publish_event(message,topic='FSP_job')
    except Exception, ex:
        print "monitor_job failed with exception %s" % ex

def remove_from_q(self, debug_on = False):
```

```
        #removes the job from the queue
        # scancel for SLURM, and qdel for PBS
        # scancel and qdel will remove the job from the queue if waiting
        try:
            if self.my_vals["batch_mgr"] == "SLURM":
                cmd = "scancel " + self.my_vals["jobid"]
            elif self.my_vals["batch_mgr"] == "PBS":
                cmd = "qdel " + self.my_vals["jobid"]
            else:
                #kill -9
                cmd = "kill -9 " + self.my_vals["jobid"]
                os.system(cmd)
                cmd = "ps -f -p " + self.my_vals["jobid"]

            p1 = sp.Popen(cmd, shell=True, stdout=sp.PIPE, stderr=sp.PIPE)
            out = p1.stdout.read()
            message = out
            if debug_on:
                print "rfq:\n" + out
            Events.publish_event(message,topic='FSP_job')
        except Exception, ex:
            print "remove_from_q failed with exception %s" % ex

    def kill_job(self, debug_on = False):
        #kills a running job -- which is exactly what remove from queue does
        self.remove_from_q(debug_on)

    def __init__(self):
        #gets the info from the config file
        self.parse_config()

        #set the path to Jpype
        sys.path.append(self.my_vals["Jpype"])

        try:
            Events.set_default_broker(self.my_vals["event_channel"])
        except Exception, ex:
            print "events not working: %s" % ex

if __name__ == "__main__":
    my_job = fsp_job()

    my_job.submit_job()
    time.sleep(2)
    my_job.monitor_job(0.5)
    time.sleep(2)
    my_job.kill_job()
```

References

[1] MCAT - a meta information catalog. http://www.npaci.edu/DICE/SRB/mcat.html. San Diego Supercomputer Center, NPACI Data Intensive Computing Environment.

[2] MDSplus. http://www.mdsplus.org/. Massachusetts Institute of Technology, Center for Nuclear Resarch (Padua, Italy) , Los Alamos National Lab.

[3] The NCSA HDF home page. http://hdf.ncsa.uiuc.edu/. The National Center for Supercomputing Applications.

[4] D. R. S. G. K. D. B. W. Beth Plale, Dennis Gannon and M. Ramamurthy. Towards dynamically adaptive weather analysis and forecasting in lead. In *ICCS Workshop on Dynamic Data Driven Applications*, Atlanta, GA, May 22-25 2005.

[5] R. Bramley, K. Chiu, J. C. Huffman, K. Huffman, and D. F. McMullen. Instruments and sensors as network services: Making instrument first class members of the grid. Technical Report TR588, Indiana University, Computer Science Department, December 2003. http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR588.

[6] R. Bramley, D. Gannon, T. Stuckey, J. Villacis, E. Akman, J. Balasubramanian, F. Breg, S. Diwan, and M. Govindaraju. *The Linear System Analyzer*, pages 123–134. Kluwer, 2000.

[7] I. U. Extreme Lab. Ws-messenger. `http://www.extreme.indiana.edu/xgws/messenger/index.html`.

[8] Y. Ma and R. Bramley. A composable data management architecture for scientific applications. In *Challenges of Large Applications in Distributed Environments (CLADE)*, pages 35–44, Research Triangle Park, NC, July 2005.

[9] C. H. Y. Huang, A. Slominski and D. Gannon. Ws-messenger: A web services based messaging system for service-oriented grid computing. In *6th IEEE International Symposium on Cluster Computing and the Grid*, Singapore, China, May 16-19 2006.