Styx Grid Services: Lightweight Workflow Middleware for Grid Environments

J.D. Blower[1], A.B. Harrison[2] and K. Haines[1]

1. *Reading e-Science Centre, Environmental Systems Science Centre,*

University of Reading, Reading RG6 6AL, UK

Email: {jdb, kh}@mail.nerc-essc.ac.uk

Tel: +44 (0)118 378 8741

2. *School of Computer Science, Cardiff University, Cardiff CF24 3AA, UK*

Email: a.b.harrison@cs.cardiff.ac.uk

Tel: +44 (0)29 20876964

Keywords: Styx, streaming, third-party transfers, WS-RF, Condor, Globus

Abstract

The service-oriented approach to performing distributed scientific research is potentially very powerful but is not yet widely used in many scientific fields. This is partly due to the technical difficulties involved in creating services and composing them into workflows. We present the Styx Grid Service, a simple system that wraps command-line programs and allows them to be run over the Internet exactly as if they were local programs. Styx Grid Services are very easy to create and use and can be composed into powerful workflows with simple shell scripts or more sophisticated graphical tools. Data can be streamed directly from service to service and progress can be monitored asynchronously using a mechanism that places very few demands on firewalls. Styx Grid Services can be used as an easy-to-use, uniform interface to Condor and Globus resources, permitting the creation of workflows that span various Grid resources and can be integrated with Web Services and the WS-RF specification.

1 Introduction

The use of service-oriented architectures (SOAs) in scientific computing is increasing. The principal advantage of the SOA approach is that scientists can access resources such as databases, high-end computing resources, laboratory equipment and sensor networks over the Internet without knowledge of the underlying infrastructure. Several independent services can be combined in a distributed application or *workflow* to solve a particular problem. For example, a scientist might wish to construct a workflow in which several pieces of data are extracted from databases in different locations, analyzed using a distributed computing resource, then finally visualized on his or her local machine.

At present, however, there are very few examples of scientific communities that work routinely in this way. The reasons for this include:

- The creation of such services is beyond the technical expertise of most scientists and so very few services exist for scientists to use.
- The process of constructing workflows from these services is also technically challenging.

- Many service providers employ complex security frameworks that are hard to use.

Web Services, although designed primarily for business-to-business interaction, are seeing increasing use in distributed scientific computing. They provide significant advantages for creating loosely-coupled, interoperable services: they are accessed through XML messaging and are thus inherently cross-platform, they are self-describing (through Web Service Definition Language – WSDL – documents) and are a widely-accepted standard for distributed computing. However, Web Services have some important limitations in the context of scientific workflows. In particular, it is impractical to encode anything but a small amount of data in XML due to the processing time required and the inflating effect of doing so. Furthermore, scientific services are often long-running and so it is highly desirable to be able to monitor the progress and status of the service as it runs using asynchronous notifications. Solutions such as the Web Services Resource Framework (WSRF [12]) employ notification mechanisms that require the client to run a server process. This requirement means that clients that are behind stringent firewalls or Network Address Translation (NAT) systems will not receive these notifications.

We describe a service type that addresses the above issues: the *Styx Grid Service* or SGS. A Styx Grid Service is a service that wraps a command-line (i.e. non-graphical) program and allows it to be run remotely. The key features of the SGS system are:

- Scientists can write a program in their language of choice, then deploy it as a Styx Grid Service with minimal effort.
- Clients can run remotely-deployed SGSs exactly as if they were local programs.
- Simple workflows can be created using shell scripts. Clients can create more complex workflows through more sophisticated tools.
- SGSs can be used as a uniform front end to Condor pools, Globus resources and other distributed computing facilities, simplifying their use.
- Data can be streamed directly between service instances in the most compact binary form, meaning that the workflow enactor does not have to process these

intermediate data.

- SGSs can interoperate with Web Services through suitable adapters.

2 Styx Grid Services: overview

Our main goal in developing the Styx Grid Services system was to create remote services that are used exactly as if they were local programs. The basis of the system is the Styx protocol for distributed systems.

2.1 The Styx protocol

Styx [10] is a well-established protocol for building distributed systems: it is a key component of the Inferno [5] and Plan 9 [9] operating systems (in Plan 9, Styx is known as "9P": the current version of Styx is equivalent to 9P2000). In Inferno and Plan 9, applications communicate with all resources using Styx, without knowing whether the resources are local or remote. Styx is essentially a file-sharing protocol, similar in some ways to NFS. However, in a Styx system the "files" are not always literal files on a hard disk. They can represent a block of RAM or the interface to a program, database or physical device. Styx can therefore be used as a uniform interface to access diverse resource types. Whereas in Remote Procedure Call (RPC)-style Web Services the resources are accessed through a set of methods, Styx resources are accessed by reading from and writing to a hierarchy of files, which is known as a *namespace*.

We developed a pure-Java implementation of the Styx protocol (JStyx [1]) and used this as the basis for the Styx Grid Services software..

2.2 The SGS namespace

The namespace of a typical Styx Grid Service server is shown in Fig. 1. A server can host several Styx Grid Services, which are each represented as a directory in the namespace. A more detailed description of the purpose of the files in the namespace can be found on the project website [2].

Due to the filesystem-like structure of the namespace, every resource on an SGS server can be identified naturally and uniquely as a URL. For example, the file that represents the standard output stream of instance `1` of the the `mySGS` service can be identified by `styx://<server>:<port>/mySGS/instances/1/outputs/stdout`. This is very important in the context of workflows: these URLs are passed between services in a workflow as data pointers, to enable direct transfer of data between services (see Sect. 4.1).

2.3 Security

The Styx protocol itself deliberately does not mandate any particular security mechanism. The JStyx software permits the SGS server to run in two modes, each with a different security mechanism:

In *daemon mode*, the SGS server runs as a persistent daemon (server process) as a generic user, exchanging Styx messages on a single network socket. In this mode, users authenticate through a custom mechanism and the SGS server administrator must maintain a dedicated user database. Traffic between the client and the server can optionally be encrypted using SSL (Secure Sockets Layer).

In *tunnelled mode*, users log on to the SGS host through the Secure Shell (SSH), then execute an SGS server process that runs *with the permissions of the user in question.* This process exchanges Styx messages on its standard input and output streams and these messages are directed to and from the client over the network through the secure SSH tunnel. There is no dedicated user database. If Globus security is required, GSI-SSH can be used in place of SSH: in this case, users authenticate through X.509 proxy certificates [3].

In either case, the SGS server only requires a single incoming port to be open through its firewall: a single server program handles all tasks such as file transfers, control messages and notifications. Since clients interact with the server through persistent connections, clients require *no* incoming ports to be open: furthermore the client can be behind a NAT router. This is how we solve the problem of asynchronous notification that was discussed in section 1 above. Hence SGS servers can be deployed easily and securely.

3 Creating Styx Grid Services

Neither service providers nor end-users need to know anything about the technical details discussed in section 2 above. The process of wrapping a command-line program as a Styx Grid Service is very simple. Let us take the example of a simple visualization program called `makepic` that reads an input file and creates a visualization of the results as a PNG (Portable Network Graphics) file. The names of the input and output files are specified on the command line, for example:

```
makepic -i input.dat -o pic.png
```

This program can be deployed on a server as a Styx Grid Service as follows. The `makepic` program is installed on the SGS server. The service provider creates a simple XML configuration file that describes the program completely in terms of its inputs, outputs and command-line arguments (see Fig. 2). The SGS server program parses this configuration file and sets up the SGS namespace (Fig. 1). The `makepic` executable file itself cannot be accessed directly by clients.

3.1 Executing SGSs just like local programs

Once the `makepic` program is deployed as a Styx Grid Service, clients can run the service from remote locations, exactly as if the `makepic` program were deployed on their local machines. They do this using the `SGSRun` program, which is a generic client program for running any SGS:

```
SGSRun <hostname> <port> makepic -i input.dat -o pic.png
```

where `<hostname>` and `<port>` are the host name (or IP address) and port of the SGS server respectively. The `SGSRun` program connects to the SGS server and downloads the XML description of the `makepic` program (Fig. 2). `SGSRun` uses this configuration information to parse the command line arguments that the user has provided. It then knows that the local file `input.dat` is an input file and uploads it automatically from the user's machine to the SGS server before the service is started. Having started the service, `SGSRun` knows that `makepic` will produce an output file called `pic.png`, which it downloads to the user's machine. (For programs that use the standard

streams – `stdout`, `stderr` and `stdin` – these are redirected to and from the console as appropriate.)

It is a very easy task to create a simple shell script (or batch file in Windows) called `makepic` that wraps the `SGSRun` program and contains the host name and port of the SGS server, so the user can simply run:

```
makepic -i input.dat -o pic.png
```

exactly as before. The `makepic` script (which calls the remote Styx Grid Service) behaves identically to the original executable.

4 Creating workflows with shell scripts

Given that remote SGSs can be executed exactly like local programs, workflows can be created with simple shell scripts. Workflows are simply high-level programs and so it is natural to use a scripting environment to create them. This allows SGSs to be combined easily with local programs and permits the use of all the programming features that the scripting language provides. Let us consider a simple workflow of two Styx Grid Services. The first is a service called `calc_mean` that takes a set of input files and calculates their mean (this service might, for example, calculate the mean of an ensemble of climate forecasts). The second SGS is the `makepic` service from section 3 above, which is used to visualize the mean data. The shell script (workflow) that would be used to take a set of input files, calculate their mean and plot the result would be:

```
calc_mean input*.dat -o mean.dat
makepic -i mean.dat -o graph.png                                    (1)
```

Note that this is *exactly the same script* as would be used to invoke the programs if they were installed locally. (This assumes that the user has created wrapper scripts called `calc_mean` and `makepic` that invoke the `SGSRun` program as described above.)

4.1 Third-party data transfers

The above "workflow" (shell script) is very simple but not optimally efficient. The intermediate file `mean.dat` is not required by the user: it is simply uploaded to the `makepic` service as soon as it is downloaded. This wastes time and bandwidth. The intermediate file can be *passed directly between the services* with only a minor change to the script:

```
calc_mean input*.dat -o mean.dat.sgsref
makepic -i mean.dat.sgsref -o graph.png                    (2)
```

The `.sgsref` extension is a signal to the system to download a *reference* (URL) to the output file and place it in the file `mean.dat.sgsref`. This reference is then passed to the `makepic` service, which downloads the real file directly from the `calc_mean` service. Hence this intermediate file does not pass through the workflow enactor (i.e. the shell environment on the client's machine): See Fig. 3. This is a *third-party data transfer*.

4.2 Data streaming using the pipe operator

Let us imagine that the `calc_mean` program outputs data to its standard output, instead of writing to an output file. Similarly, imagine that the `makepic` program reads data from its standard input and outputs the picture to its standard output. The command required to execute the workflow (with both local programs *and* Styx Grid Services) is:

```
calc_mean input*.dat | makepic > graph.png                 (3)
```

Here, the intermediate data are being streamed to the local client, then streamed back out to the `plot` service. We can ensure that the intermediate data are streamed directly between the services with a minor change to the command:

```
calc_mean input*.dat --sgs-ref-stdout | makepic > graph.png  (4)
```

The `--sgs-ref-stdout` flag is a signal to send a reference (URL) to the standard output of the `calc_mean` service to the standard input of the `makepic` service. The

`makepic` service then downloads the intermediate data directly from the `calc_mean` service.

5 SGS and Grid resources

In the above examples the `calc_mean` and `makepic` programs ran directly on the respective SGS servers. The SGS system can also be used as an easy-to-use interface to clusters, Condor pools and Globus resources. The Styx Grid Services that are deployed on these resources can be composed into workflows using simple scripts exactly as above.

5.1 Running jobs on clusters and Condor pools

By installing the SGS server software on the head node of a distributed resource management (DRM) system such as Condor [11] or Sun GridEngine, parallel jobs can be run through the SGS interface. This is particularly useful when the user wishes to execute the same program many times over a number of input files. This is known as "high-throughput computing" and is commonly used in Monte Carlo simulations and parameter sweep studies. In such a study a user might wish to execute a Styx Grid Service over a large number of input files. Normally this would require the user to manually upload the input files, name them in some structured fashion and create an appropriate job description file in a format that is understood by Condor or the particular DRM system in question.

In the SGS system, the running of these high-throughput jobs is very simple. Let us imagine that the user has a set of input files for the `makepic` program in a directory called `inputs` on his or her local machine. The user simply runs the `makepic` Styx Grid Service as before but, instead of specifying a single input file on the command line, the user enters the name of the `inputs` directory:

```
makepic -i inputs -o pic.png
```

where `makepic` is the script that wraps the `SGSRun` executable as described above.

The whole `inputs` directory (containing the input files) is automatically uploaded to the SGS server. The server notices the presence of an input directory where it was

expecting a single file. It takes this as a signal to run the `makepic` program over each file in the input directory, producing a picture for each file. The SGS server uses the underlying DRM system (e.g. Condor) to run these tasks in parallel on the worker nodes. The progress of the whole job is displayed to the client as the individual tasks are executed. The client then downloads these output pictures automatically and places them in a directory called `pic.png` on the user's local machine.

5.2 Running jobs on Globus resources

Many resource providers (including the UK National Grid Service) use the Globus toolkit as secure middleware to allow users to run jobs. The Styx Grid Services system can be used as a simple interface to Globus resources in two ways:

- The SGS server runs on the Globus resource in "tunnelled mode" (section 2.3 above) and the user runs the Styx Grid Services through a GSI-SSH tunnel.
- A meta-scheduler such as Condor-G is employed. The SGS system is used to submit jobs to Condor-G as in section 5.1 above and Condor-G forwards the jobs to a Globus resource.

In both cases, users can run jobs on Globus resources exactly as above, i.e. in the same way in which they would run local programs. Both methods require the user to have a valid proxy certificate.

5.3 A workflow using heterogeneous Grid resources

To illustrate all of the above concepts, we shall give an example of a workflow that brings together various Grid resources in order to solve a particular scientific problem. This example is taken from the authors' own research into ensemble forecasting of the Earth's climate. In an ensemble forecast, many different simulations of the Earth's climate are run, each with a different set of initial conditions, to give a probability distribution of different climate scenarios. These initial conditions are generated using a local program called `gen_inputs`. Each initial condition is used to drive a climate simulation (`climate_sim`), which runs on one of a number of high-performance clus-

ter resources that are load-balanced through Condor-G. The whole ensemble is then analyzed on a Globus resource using a program called `analyze_ensemble`.

`climate_sim` is installed as a Styx Grid Service on the Condor-G head node. `analyze_ensemble` is installed as an SGS on the head node of the Globus resource and is accessed through GSI-SSH. The user has created wrapper scripts for both of these SGSs. The workflow (shell script) to perform the analysis is:

```
gen_inputs -o inputs
climate_sim -i inputs -o outputs.sgsref
analyze_ensemble -i outputs.sgsref -o stats.dat
makepic -i stats.dat -o stats.png
```

In the first line, the input files are generated on the user's local machine and placed in the `inputs` directory. In the second line, all these input files are uploaded to the Condor-G scheduler, which runs the `climate_sim` program over each input, balancing the load amongst a number of clusters. Note that the client does not download the actual output files from the ensembles: the `.sgsref` extension ensures that the client downloads *references* (URLs) to these files instead. When all the ensembles are complete the references to the output files are uploaded to the Globus resource, which downloads the output files and runs the `analyze_ensemble` program, outputting a set of ensemble statistics, which are downloaded to the client. Finally, these output statistics are visualized using the `makepic` service from the above examples.

6 Interfaces to SGS

The command line scripting interface to the SGS system that is described above is perhaps the simplest way of creating SGS workflows. In some cases, however, there are significant advantages in using more sophisticated graphical tools to interact with services and create workflows. In particular, graphical interfaces can provide richer interactivity with the SGS server: progress and status can be monitored graphically, input parameters can be set using graphical controls and the service can be steered [4]. Furthermore, to provide greater interoperability with other Grid environments it can be advantageous to wrap SGS in standards compliant interfaces using ubiquitous

protocols such as HTTP. In the following subsections we describe current bindings to SGS in other systems.

6.1 Using graphical workflow tools

The Taverna workbench (`http://taverna.sf.net`) is a graphical workflow system that was designed for performing *in silico* experiments in the field of bioinformatics, but it is sufficiently general to be useful to other communities. We have worked with the Taverna developers to incorporate support for Styx Grid Services into the software. Using Taverna, the user can build workflows by mixing diverse service types, including Web Services and SGSs.

The Triana workflow system (`http://trianacode.org`) is a graphical workflow environment that can interface with many different service types but cannot currently interface directly with Styx Grid Services. As a result we have developed two ways of achieving integration using Web services which are supported in Triana. The first mechanism uses brokered architecture. A separate Web Service is created that accepts SOAP messages and uses the information therein to communicate with an SGS server. This is described in more detail in [4].

The second mechanism uses WSPeer [6], the Peer-to-Peer oriented Web Service framework that is used by Triana. WSPeer has a binding to the Styx protocol for delivering SOAP messages over Styx. This allows the Styx Grid Service itself to accept SOAP messages that are written directly to a file in its namespace representing the service. When the SGS is deployed using WSPeer, a WSDL document is generated and placed into the SGS namespace so it can read by clients. This WSDL document defines service operations that encapsulate the messages and data to be written to the files in the SGS namespace. For example, the WSDL will contain an operation for setting the input to the SGS which might have a signature such as `setStdin(String url)`. To tell the SGS to read its input data from a particular URL, a client can invoke the operation.

6.2 Wrapping SGSs as WS-Resources

The Web Services Resource Framework (WS-RF) is a recent specification which addresses the need to handle network exposed entities that maintain state across multiple service invocations. Styx Grid Services fall into this category because they display stateful characteristics - they are created, have a lifetime, and have available properties such as the current status of the wrapped program. WS-RF uses the *WS-Resource* abstraction to represent resources that are exposed and manipulated via a Web Service. A WS-Resource contains a service endpoint and a resource identifier. A client with a suitable WS-Resource can invoke the service at the endpoint specified and decorate the SOAP header of the request with the resource identifier. The service maps the identifier to some back end resource - in our case an SGS instance - and invokes the requested operation on the resource.

WSPeer supports WS-RF and can be used to expose an SGS as a WS-Resource. This is achieved by transforming the SGS configuration information (see section 3) into *ResourceProperties*. These are QName/value pairs of a specified data type that are used to describe a WS-Resource type in the service WSDL as an XML schema element. Therefore a service that is being deployed as a front-end to a type of SGS - the `makepic` program for example whose configuration file is depicted in Figure 2 - will parse this configuration file for any program specific parameters and combine this with the standard SGS properties represented in the namespace under the `time/` and `serviceData/` directories. These properties are used to populate an XML schema element which is a template of the particular SGS type, and is inserted into the service WSDL. Service clients can read this schema to determine the available properties of an SGS.

There are a number of synergies between the WS-RF suit of specifications and the properties defined by an SGS. For example, the `time/` directory in the SGS namespace, which houses files containing data pertinent to the lifetime of the service, can be mapped onto the properties defined in the WS-ResourceLifetime [8] specification. Likewise the `serviceData/` directory of the SGS namespace contains state data which are easily mapped to ResourceProperties or can be exposed as WS-Notification [7] topics that clients can subscribe to and receive notifications of changes from.

WSPeer is capable of wrapping an SGS as a WS-Resource in two ways. The first way (brokering) involves creating a WSRF service that receives SOAP messages over HTTP and translates the information therein into Styx messages, which it sends to a separate SGS server (which may be on the same host). The second is to use the Styx protocol itself to send and receive XML. In this case the WS-RF service that is exposing the SGSs as WS-Resources is deployed at a Styx endpoint. The ability of WSPeer to use the Styx protocol directly allows clients that are behind firewalls and NAT systems to receive WS-Notification messages via the notification mechanism described in section 2, thus overcoming the problem of clients needing to provide an accessible network address to receive notifications.

While it is useful to expose SGS functionality according standard specifications, we do not attempt to wrap the SGS data streams in XML for performance reasons. For example, an output stream exposed as a ResourceProperty consists of a URI from which data can be received. However, the actual data in the stream is application specific.

7 Discussion

We have described a new type of Internet service, the Styx Grid Service (SGS). SGSs wrap command-line programs and allow them to be run from anywhere on the Internet exactly as if they were local programs. SGSs can be combined into workflows using simple shell scripts or more sophisticated graphical workflow engines. Data can be streamed directly between SGS instances, allowing workflows to be maximally efficient. We have shown that Styx Grid Services can operate as part of a Web Services or WSRF system through the use of methods including broker services.

The SGS system has some important limitations:

- The entities that are passed between services in a workflow are files and are not typed. There is no way for the workflow enactor (e.g. a shell script) to tell whether the type of the output from one service matches the type of the input of the next service. It is up to the services themselves to check that their inputs are valid.

- Although constructs such as loops are supported by the shell scripting interface (and graphical workflow tools), the variables used to control these loops cannot be read directly from the outputs of SGSs.
- Data transfer rates between services through the Styx protocol are slower than through HTTP or GridFTP [4]. The SGS system may therefore not be optimal for problems that involve very large file transfers.

An important subject of future research would be to use the SGS approach to wrap entities such as classes and functions, rather than whole executables: in this case, inputs and outputs could be strongly typed and could also be captured by workflow engines, solving some of the above problems.

The SGS system has been designed so that services and workflows are easy to create. It is well within the reach of most end-users (scientists) to do so with no help from dedicated technical staff. Problems connected with firewalls and NAT routers are vastly reduced compared with other systems, allowing for easy deployment and use. We believe that the Styx Grid Services system represents a significant step forward in increasing the usability of service-oriented systems and workflows in science.

Acknowledgements

The authors would like to thank Tom Oinn for incorporating the SGS framework into Taverna and Vita Nuova Holdings Ltd. for technical help with the Styx protocol. This work was supported by EPSRC and NERC, grant refs. GR/S27160/1 and R8/H12/36.

References

[1] J. Blower, The JStyx library, Online, http://jstyx.sf.net (2005).

[2] J. Blower, Styx Grid Services, Online, http://www.resc.rdg.ac.uk/jstyx/sgs (2006).

[3] J. Blower and K. Haines, Building simple, easy-to-use Grids with Styx Grid Services and SSH, in: Proceedings of the UK e-Science Meeting.

[4] J. Blower, K. Haines and E. Llewellin, Data streaming, workflow and firewall-friendly Grid Services with Styx, in: S. Cox and D. Walker, eds., Proceedings of the UK e-Science Meeting, ISBN 1-904425-53-4.

[5] S. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. Trickey and P. Winterbottom, The Inferno Operating System, Online: http://www.vitanuova.com/inferno/papers/bltj.html (1997).

[6] A. Harrison and I. Taylor, WSPeer - An interface to Web Service hosting and invocation, in: HIPS-HPGC Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models.

[7] OASIS, Web Services Base Notification 1.2 (WS-BaseNotification), Online, http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-03.pdf (2004), draft 03.

[8] OASIS, Web Services Resource Lifetime 1.2 (WS-ResourceLifetime), Online, http://docs.oasis-open.org/wsrf/wsrf-ws_resource_lifetime-1.2-spec-pr-02.pdf (2005), public Draft 02.

[9] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey and P. Winterbottom, Plan 9 from Bell Labs, Online, http://www.cs.bell-labs.com/sys/doc/9.html (1995).

[10] R. Pike and D. M. Ritchie, The Styx architecture for distributed systems, Online, http://www.vitanuova.com/inferno/papers/styx.html (1999).

[11] The Condor Project, Condor, Online, http://www.cs.wisc.edu/condor/ (2006).

[12] The Globus Alliance, The WS-Resource Framework, Online, http://www.globus.org/wsrf/ (2005).

Figure Captions

Fig. 1: Example namespace (virtual filesystem) exposed by an Styx Grid Services server. This namespace corresponds with the XML configuration file in Fig. 2. This SGS server exposes two services: `makepic` and `mySGS`. There are two instances of the `makepic` service, of which the first is shown expanded. The namespace contains directories for parameters, input files and output files, as well as a file to create new instances (`clone`) and a file to start and stop a particular service instance (`ctl`). The `serviceData` directory contains files that can be read to give the status of the service. The `time` directory contains files that pertain to the lifecycle of a service. See section 2.2.

Fig. 2: Portion of the configuration file on a Styx Grid Services server, describing the `makepic` program that is deployed. This specifies that the program expects one input file, whose name is given by the command-line argument following the "-i" flag. The program outputs one file, whose name is given by the command-line argument following the "-o" flag, and also outputs data on its standard output stream. The namespace that results from this configuration file is shown in Fig. 1.

Fig. 3: Illustration of direct data passing between Styx Grid Services. The ellipses are Styx Grid Services and the dotted box represents the client's machine. The dashed arrows represent data transfers that result from the workflow in script 1 in section 4. The intermediate file `mean.dat` is not required by the client and so the workflow can be arranged (script 2 in section 4) so that this file is passed directly between the SGSs (solid black arrows).

Fig. 1

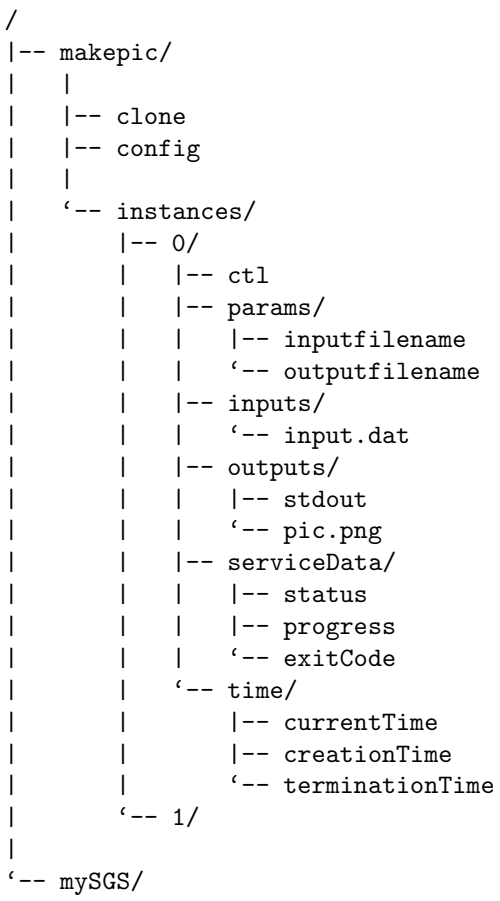
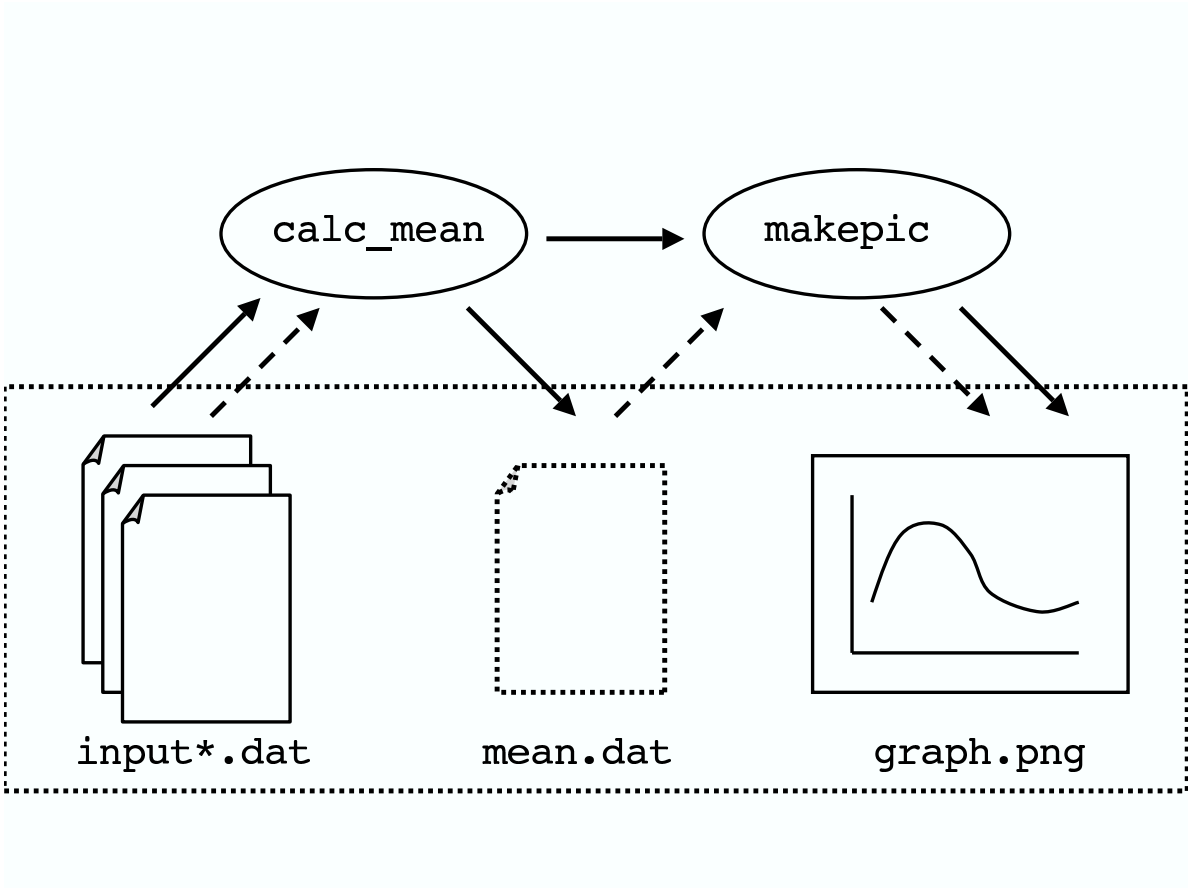

/
|-- makepic/
| |
| |-- clone
| |-- config
| |
| `-- instances/
| |-- 0/
| | |-- ctl
| | |-- params/
| | | |-- inputfilename
| | | `-- outputfilename
| | |-- inputs/
| | | `-- input.dat
| | |-- outputs/
| | | |-- stdout
| | | `-- pic.png
| | |-- serviceData/
| | | |-- status
| | | |-- progress
| | | `-- exitCode
| | `-- time/
| | |-- currentTime
| | |-- creationTime
| | `-- terminationTime
| `-- 1/
|
`-- mySGS/

Fig. 2

```
<gridservice name="makepic" command="/path/to/makepic">
  <params>
    <param name="inputfile" paramType="flaggedOption" flag="i" required="yes"/>
    <param name="outputfile" paramType="flaggedOption" flag="o" required="yes"/>
  </params>
  <inputs>
    <input type="fileFromParam" name="inputfile"/>
  </inputs>
  <outputs>
    <output type="fileFromParam" name="outputfile"/>
    <output type="stream" name="stdout"/>
  </outputs>
</gridservice>
```


calc_mean
makepic
input*.dat
mean.dat
graph.png