Styx Grid Services: Lightweight, easy-to-use middleware for scientific workflows

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
jdb@mail.nerc-essc.ac.uk,
[2] School of Computer Science, Cardiff University, Cardiff CF24 3AA, UK

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 interoperate with Web Services and WS-Resources.

1 Introduction

The concept of "workflow" in e-Science terminology refers to the composition of high level modules (which are often distributed, Internet-based services such as Web Services) in order to create an application. For example, a scientist may wish to extract data from a number of data archives in different physical locations, perform some analysis on these data on a high-performance resource in another location, then produce some visualization of the end result on his or her local machine. The services in this workflow are mutually independent ("loosely coupled") and may be hosted by a number of different service providers.

In theory, this approach should allow scientists with little technical knowledge to create powerful distributed applications. In practice, however, there are – at the time of writing – very few examples of scientific communities that have started to work in this way on a routine basis. A large part of the reason for this is the paucity of services that are available for scientists to use.

Web Services provide very 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 trivial 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 OGSI [1] and the Web Services Resource Framework (WSRF, `http://www.globus.org/wsrf/`) 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.

If scientists are to adopt the workflow approach in their work, there must exist a set of useful services from which these workflows can be constructed. In order to achieve such a "critical mass" of services, it must be possible for *scientists* to be able to create such services with minimal or no help from dedicated technical staff. Several systems exist to make the task of creating Web and Grid Services easier (e.g. Soaplab [2] and GEMLCA [3]). However, these systems are still typically difficult for scientists to use, either because the scientists are not familiar with the Web or Grid Services model or because the systems are based on complex, heavyweight toolkits such as Globus (`http://www.globus.org/`), which are designed for application builders, not end users. Therefore, technical support is needed to create these services and the critical mass of useful services is never reached. Once created, it is important that the services be as easy as possible to use.

There is a clear demand from scientists [4] for simple, lightweight middleware that does not necessarily support every possible feature but that is easy to install, use and understand. This demand has resulted in the recent development of systems such as WEDS [5].

We have developed a solution that addresses all of the above issues. We focus on the process of creating services that are based on command-line programs (which may be tried-and-tested "legacy" codes) but the principles we describe could be extended to other service types. The solution we present deliberately moves away from the Web Services model but, as we shall demonstrate, still maintains a high level of interoperability.

We introduce the Styx Grid Services (SGS) system, a framework for wrapping command-line (i.e. non-graphical) programs and allowing them to be run as a service from anywhere on the Internet. The major advantages are:

- It is very easy to create SGSs that wrap command-line programs.
- Remote SGSs can be used *exactly* as if they were locally-installed programs.
- Workflows can be created using simple shell scripts or graphical tools.
- Data can be streamed directly between remote service instances.
- The software is very lightweight and quick to install (less than 5 MB, including all dependencies).
- The software places few demands on firewalls, requiring only one incoming port to be open on the server and *no* incoming ports to be open on client machines.

2 Styx Grid Services: background

Our main goal in developing the Styx Grid Services system was to create remote services that are just as easy to use as local programs. The basis of the system is the Styx protocol for distributed systems [6]. In Styx-based systems *all* resources are represented as files, analogous to the representation of the mouse as the file `/dev/mouse` in Unix variants. Styx is a file-sharing protocol that can operate over a large number of transports such as TCP/IP and UDP. It forms the core of the Inferno and Plan9 operating systems, in which applications communicate with all resources using Styx, without knowing whether these resources are local or remote (in Plan9, Styx is known as "9P"). We developed an open source, pure-Java implementation of Styx (JStyx, `http://jstyx.sf.net`) and used it as the base for the SGS system.

All resources in Styx systems are represented as a file hierarchy, which is known as a *namespace*. We have defined a namespace that represents a command-line program [7]. Clients interact with this program by reading from and writing to the files in this namespace over the network. For example, the SGS namespace contains an `inputs/` directory, into which clients write the input files that the program will consume.

Due to this filesystem-like structure, every resource on a Styx server can be represented very naturally as a URL. For example, the file that represents the standard output stream of instance `1` of the the `mySGS` service can be represented by the URL `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 to enable direct transfer of data between services (see Sect. 4.1).

The Styx protocol itself deliberately does not mandate any particular security mechanism. In JStyx, we secure systems using transport-layer security (TLS), using public key certificate-based authentication and (optional) encryption of network traffic. This encryption is transparent to applications that use JStyx.

When Styx clients and servers interact they typically use *persistent connections*: the client connects to the server and leaves the connection open for as long as it needs. This means that the client can receive asynchronous messages from the server without requiring any incoming ports to be open through its firewall. Also, the client does not need a public IP address so it does not matter if the client is behind a NAT router. This is how we solve the problem of asynchronous notification that was discussed in Sect. 1 above. A single Styx server can handle multiple tasks (such as asynchronous messaging and file transfers) and so servers only need to have a single incoming port open through the firewall. This helps to make the deployment and use of Styx systems very easy.

3 Wrapping programs as Styx Grid Services

Neither service providers nor end-users need to know anything about the technical details discussed in Sect. 2 above. The process of wrapping a command-line

program as a Styx Grid Service is very simple. A short XML description of the program in question is constructed. This description is a complete specification of the program, specifying the command-line parameters and input files that the program expects and the output files that the program produces. (There is other optional information that can be added, but that is beyond the scope of this paper.) A server program is then run that parses the XML file and sets up the SGS namespace. A single server can host many Styx Grid Services. Note that the executable itself cannot be read over the Styx interface.

3.1 Executing SGSs just like local programs

Once the program is deployed as a Styx Grid Service, it can be run from anywhere on the Internet, *exactly as if it were a local program*. For example, consider a program called `calc_mean` that reads a set of input files (perhaps from a set of scientific experiments), calculates their mean and writes the result to an output file. If this service were deployed on the server `remotehost.com`, listening on port 9092, and the user has a set of input files (called `input1.dat`, `input2.dat` etc.) the user would run the service by entering the following command:

```
SGSRun remotehost.com 9092 calc_mean input*.dat -o mean.dat
```

The `SGSRun` program is a general-purpose command-line client for any Styx Grid Service and it performs the following tasks: It connects to the server and downloads the XML description of the Styx Grid Service that it is being asked to run. It uses this description to parse the command-line arguments that the user has provided. If these are valid, it creates a new instance of the service and sets its parameters, based on these command-line arguments. It then uploads the necessary input files, starts the service running and downloads the output data as soon as they are produced. If the SGS uses the standard streams (stdout, stderr and stdin) these are redirected to and from the console as appropriate.

It is an easy task to create a simple wrapper script called `calc_mean` on the client. This wraps the `SGSRun` program and contains the location and port of the remote server. Then this wrapper script can then be treated *exactly* as if it were the `calc_mean` program itself.

4 Creating workflows from Styx Grid Services

4.1 Using shell scripts as workflows

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 (e.g. loops and conditionals). Let us consider a simple workflow of two Styx Grid Services. The first is the `calc_mean` service from the above example. The second SGS,

called `plot`, takes a single input file and turns it into a graph. The shell script (workflow) that would be used to take a set of input files, calculate their mean and plot a graph of the result would be:

```
calc_mean input*.dat -o mean.dat
plot -i mean.dat -o graph.gif                                    (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 `plot` that invoke the `SGSRun` program as described above.)

Direct data passing. 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 `plot` 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
plot -i mean.dat.sgsref -o graph.gif                             (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 `plot` 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 client's machine). See Fig. 1.

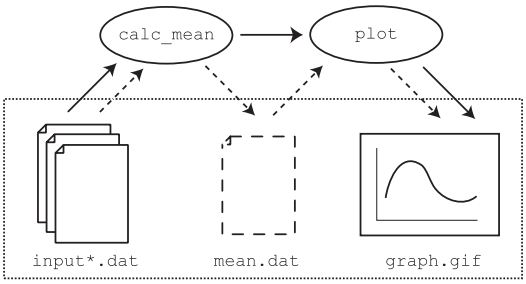

Fig. 1. Illustration of direct data passing between Styx Grid Services. The ellipses are Styx Grid Services and the dotted box represents the client's machine. The dashed arrows represent data transfers that result from the workflow in script 1 in section 4.1. The intermediate file `mean.dat` is not required by the client and so the workflow can be arranged (script 2 in section 4.1) so that this file is passed directly between the SGSs (solid black arrows).

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 `plot` 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 | plot > graph.gif                    (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 | plot > graph.gif   (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 `plot` service. In this way the intermediate data are streamed directly between the services, across the Internet.

Weaknesses of this approach. The inputs and outputs of SGSs are files and so the "workflow engine" (i.e. the shell environment) performs no type checking on these entities. The reponsibility of checking for validity of inputs is left to the services themselves. Secondly, although constructs such as loops are supported by the shell, the variables used to control these loops cannot be read directly from the outputs of SGSs. 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 the above two problems.

4.2 Using graphical workflow tools

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 and the service can be steered [7].

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 (including Web Services), but cannot currently interface directly with Styx Grid Services. We have developed two ways for achieving this:

1. **Brokering:** A separate Web Service is created that accepts SOAP messages and uses the information therein to communicate with an SGS server [7].
2. **"SOAP over Styx":** The Styx Grid Service itself is modified to accept SOAP messages that are written directly to a special file in its namespace using Styx. The SGS describes itself using a WSDL document that is also readable via a special file. This WSDL document defines service operations that encapsulate the messages and data to be written to the files in the SGS namespace. So for example, to tell the SGS to read its input data from a certain URL, the client invokes the `setStdin(String url)` operation that is defined in the WSDL. We have built support for this into WSPeer [8], the Peer-to-Peer oriented Web Service framework that is used by Triana.

4.3 Wrapping SGSs as WS-Resources

The Web Services Resource Framework (WSRF) is a recent specification which addresses the need to handle resources that maintain state across service invocations. "WS-Resources" are resources that are exposed and manipulated via a Web Service. A Styx Grid Service is exposed as a WS-Resource by transforming its configuration information (Sect. 3) into *ResourceProperties*, which are QName/value pairs of a specified data type that are used to describe a WS-Resource in WSDL. SGSs define certain properties which map directly onto WSRF specifications. 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 [9] specification. The `serviceData/` directory of the SGS namespace contains state data which clients can subscribe to and receive notifications of changes from. These are exposed as WS-Notification [10] topics.

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. The second is to use the Styx protocol itself to send and receive XML, as described in Section 4.2. 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 Sect. 2. 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, while the actual data in the stream is application specific.

5 Conclusions

We have introduced 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.

A key strength of the SGS system is that it is very easy to create and use services: 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 ref. GR/S27160/1.

References

1. Tuecke, S., et al.: Open Grid Service Infrastructure (OGSI) Version 1.0. Technical Report GFD-R-P.15, Global Grid Forum (2003)
2. Senger, M., Rice, P., Oinn, T.: Soaplab - a unified Sesame door to analysis tools. In Cox, S., ed.: Proceedings of the UK e-Science Meeting. (2003) ISBN 1-904425-11-9.
3. Kacsuk, P., Kiss, T., Goyeneche, A., Delaitre, T., Farkas, Z., Boczkó, T.: A high-level grid application environment to Grid-enable legacy code. ERCIM News **59** (2004)
4. Chin, J., Coveney, P.V.: Towards tractable toolkits for the Grid: a plea for lightweight, usable middleware. UK e-Science Technical Report UKeS-2004-01, http://www.nesc.ac.uk/technical_papers/UKeS-2004-01.pdf (2004)
5. Coveney, P.V., Vicary, J., Chin, J., Harvey, M.: WEDS: a Web services-based environment for distributed simulation. Phil. Trans. R. Soc. A **363** (2005) 1807–1816
6. Pike, R., Ritchie, D.M.: The Styx architecture for distributed systems. Online, http://www.vitanuova.com/inferno/papers/styx.html (1999)
7. Blower, J., Haines, K., Llewellin, E.: Data streaming, workflow and firewall-friendly Grid Services with Styx. In Cox, S., Walker, D., eds.: Proceedings of the UK e-Science Meeting. (2005) ISBN 1-904425-53-4.
8. Harrison, A., Taylor, I.: 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. (2005)
9. 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.
10. 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.