Data streaming, workflow and firewall-friendly Grid Services with Styx

Jon Blower[1*], Keith Haines[1], Ed Llewellin[2]

[1] Reading e-Science Centre, Environmental Systems Science Centre, University of Reading, Harry Pitt Building, University of Reading, Whiteknights, Reading RG6 6AL
[2] Department of Earth Sciences, University of Bristol, Wills Memorial Building, Queen's Road, Bristol BS8 1RJ

[*] Corresponding author: email address jdb@mail.nerc-essc.ac.uk

Abstract

We present the Styx Grid Service (SGS), a remote service type that allows data to be streamed directly between service instances. The Styx Grid Service is built upon the Styx protocol, an established protocol for building distributed systems. Through the use of persistent connections, clients can monitor progress, status and other service data without requiring inbound ports to be open through the firewall. Security is assured through the use of SSL to authenticate and encrypt communications between systems. SGSs can interoperate with other service types such as Web Services in a workflow through the use of suitable wrappers. We present two case studies of the use of the SGS. In the first, Styx Grid Services are used in the streaming of large amounts of data between Web Services in a workflow. In the second case study we show how the Styx Grid Service architecture can be used in collaborative visualization and computational steering.

1 Introduction

SOAP-based Web Services are not always appropriate for building certain types of distributed system. Where a distributed system involves the transfer of large amounts of data, it is usually undesirable to require that the data be encoded into XML. Furthermore, when several services are composed in a workflow, it is not efficient for the data to follow the same network path as the SOAP messages as this will result in large amounts of data being processed unnecessarily by the workflow enactor. In many cases, the best solution is to stream data directly between remote services. This is particularly important in the environmental sciences, where datasets are commonly on the terabyte scale.

Another near-universal problem afflicting designers of distributed systems is that of firewalls. In many distributed systems the end-user's client machine is required to have one or more inbound ports open through whatever firewall(s) it happens to be behind, in order to enable such features as asynchronous notification of progress, status or other service data. This is not always possible: in many situations (especially in non-academic environments or when working "away from the office") the user has little or no control over the firewall.

In this paper we will describe the Styx Grid Service (SGS), a remote service type that allows data to be streamed directly between service instances across the Internet, whilst making as few demands on firewalls as possible. We will describe two case studies illustrating the use of Styx Grid Services. The first shows how data can be streamed directly between services whilst progress and status data are monitored asynchronously. The second case study demonstrates the use of Styx Grid Services in collaborative visualization.

We will show how Styx Grid Services can interoperate with other remote resources through the use of standard protocols such as SOAP and HTTP: this allows SGSs to be combined with Web Services in a workflow environment. SGS servers can be secured to a very high level, using X.509 certificates for authentication and SSL for transport-level encryption.

2 The Styx protocol

Styx [5] is a well-established protocol for building distributed systems: it is a key component of the Inferno [3] and Plan 9 [4] distributed operating systems. It 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, physical device or data stream. Styx can therefore be used as a uniform interface to access diverse resource types such as databases, sensors and digital cameras.

Since all resources in a Styx system are represented as files, the Styx command set is very small, consisting of only 13 commands, of which "open", "read", "write" and "close" are perhaps the most commonly-used. Whereas in RPC-style Web Services the resources are accessed through a set of methods (i.e. an API), Styx resources are accessed through a hierarchy of files, known as a *namespace*. Clients interact with the resource by reading from and writing to these files using the Styx protocol. For example, the namespace for a temperature sensor might contain a file called "`stream`" which can be read to obtain a stream of live temperature data. In the Inferno operating system [3], this can be achieved very simply on the command line by executing "`cat stream`".

Connections between clients and servers in a Styx system are typically persistent. The client and server perform a mutual handshake and, after that, Styx messages can pass back and forth freely between the client and server until either peer (usually the client) closes the connection. The client sends messages and the server responds (not necessarily immediately) with corresponding replies. Importantly, any number of messages can be awaiting reply at any one time. In this way, the client can interact with several files on a Styx server simultaneously.

2.1 Styx and firewalls

The use of persistent connections helps to solve a common problem in Grid computing: that of how a client can receive asynchronous messages from the server when the client is behind a firewall that disallows inbound connections on most or all ports. Since it is the client that initializes the connection and the connection is left open, the server can send replies back to the client on the same connection at any time. This is particularly useful for the monitoring of service data in a Styx Grid Service: see section 3.4. Styx servers (including Styx Grid Service servers) only require a single inbound port to be open through the firewall and Styx clients require no inbound ports to be open at all.

2.2 Implementations of Styx

Until recently, the only widely-available implementations of Styx were built into the Inferno and Plan 9 operating systems themselves. To address this, the Reading e-Science Centre has developed the JStyx library (http://jstyx.sf.net), a pure-Java implementation of Styx that provides high-level abstractions of Styx concepts, removing the need for developers to understand the mechanics of the protocol in order to write Styx servers and clients. For example, data can be read from and written to Styx files very easily using Java InputStreams and OutputStreams respectively. The JStyx distribution includes example programs and useful debugging tools. C and Python implementations are also available, although these are in an earlier stage of development and only cover client-side code.

2.3 Performance

The message-based nature of Styx means that it cannot match dedicated file-transfer protocols such as HTTP and GridFTP for raw speed of transfer of large amounts of data from a static file on a remote server. When a client downloads a file from a Styx server it does so chunk-by-chunk (one chunk per message). This adds a small data overhead to the transfer process: each time a client requests a chunk of data from a server it sends a message of 23 bytes to the server, and each chunk of data sent back to the client from the server contains a header of 11 bytes in addition to the data.

The message size (chunk size) is mutually agreed by the client and server during the handshake when the connection is initiated and can be chosen to suit the application. For most applications a size of 8 KB is typical but if large amounts of data are expected to be transferred on the connection a larger size (e.g. 64 KB) may be chosen. Increasing the message size decreases the number of messages that need to be generated, thereby decreasing the total amount of data exchanged.

The transfer speed in Styx can be further increased by sending several simultaneous read requests over the same connection, each requesting data from a different part of the source file. The disadvantage with using this "accelerated download" with Styx is that it only works if the remote file is seekable: a Styx file is not necessarily a literal file on disk so one cannot always download data from an arbitrary position in the file.

The results of a comparison between HTTP and Styx are shown in table 1 (we have not yet compared Styx with GridFTP). Whilst Styx does not quite match HTTP for transfer speed, the difference is often not important in data streaming applications. When reading from a live data feed from a running model, for example, the limiting factor is often the rate at which the model can output new data, rather than the speed of data transfer across the network.

HTTP	100%
Styx 8KB, basic	39%
Styx 64KB, basic	80%
Styx 8KB, accelerated	92%
Styx 64KB, accelerated	95%

Table 1: Sustained data transfer rates (as percentage of HTTP transfer rate) achieved across a LAN between two machines using HTTP and Styx, showing the effect of the message size and simultaneous requests in Styx. Message sizes of 8 KB and 64 KB were tested, using 1 ("basic") and 10 ("accelerated") simultaneous read requests. The download speed increases with increasing message size and the number of simultaneous read requests, with the download speed reaching 95% of HTTP when using 10 simultaneous requests with 64 KB messages. The HTTP server was Apache and the JStyx library was used to create the Styx server. Java clients were used in all tests. No connections were secured.

2.4 Scalability

The use of persistent connections raises questions about the scalability of Styx systems. We have not yet tested exactly how many simultaneous clients can be served by a Styx server. The Inferno Grid is a Condor-like cycle-stealing system that uses Styx as its communication protocol and is known to scale to several hundred simultaneous clients, and could probably be scaled further. Based on this, we might estimate a Styx server to be able to service up to the order of a thousand clients simultaneously. For the case of the Styx Grid Service we believe that this level of scalability is more than acceptable.

2.5 Security

The Styx specification itself deliberately does not mandate any particular security mechanism, permitting a number of different methods to be used. Transport-level security fits very naturally with the Styx model and its use of persistent connections. The JStyx library provides an option to use SSL for message encryption and X.509 certificate-based mutual authentication of clients and servers. (The Inferno operating system uses a different transport-level security method that also employs public-key certificates.) Each file on a Styx server has Unix-like read-write-execute permissions for the owner, group and all users. It is therefore possible to secure Styx systems at the level of individual files.

3 The Styx Grid Service

The Styx Grid Service is designed as a means for wrapping a binary executable and providing a method for remotely accessing streams of data and service data (state data) such as the progress of a long-running service. Through the use of Styx and persistent connections, clients of SGSs do not require inbound ports to be open through their firewalls. The SGS specification and implementation have been designed so that executables can be written and used without any knowledge of how the Styx Grid Service wrapper works.

3.1 Specification

Figure 1 shows the namespace exposed by a typical SGS server. Note that not all of these files will be exposed by every SGS: some may not require any input files or parameters to be set; others may choose not to provide access to the standard error stream, for example.

3.2 Running a SGS

The steps involved in running a typical Styx Grid Service from the point of view of the client are as follows:

1. Connect to the SGS server.
2. Read from the `clone` file of the required service: this causes the server to create a new instance of the service and return the ID number of the instance to the client.
3. Upload the required input files to the newly-created SGS instance by writing into the `inputFiles/` directory.
4. Set the parameters of the SGS by writing values into the files in the `params/` directory.
5. Set the source of the data to be piped into the standard input of the service by writing the URL to the `stdin` file (see section 3.3).
6. Start the service by writing the word "start" into the `ctl` file of the instance.

7. Get data from the standard output and error of the service by reading from the `stdout` and `stderr` files.

8. Get service data (e.g. notifications of progress and status changes) by reading from the files in the `serviceData/` directory (see section 3.4).

3.3 Data streaming

A key feature of the Styx Grid Service design is the ability to stream data directly between remote service instances. The simplest way to achieve this is as follows: The underlying executables are written so that they read data from their standard input stream and write data to their standard output, in the manner of a Unix filter. If the executables were running on the same machine, the output of one could be streamed into the input of the other using the pipe operator (e.g. "`prog1 | prog2`").

When the executables are wrapped as Styx Grid Services the downstream service can be instructed to read its input data from the output stream of the upstream service by writing the URL of the output stream into the `stdin` file of the downstream service before the downstream service is started (step 5 in section 3.2). Note that any URL can be specified as the input source for an SGS, including HTTP and FTP URLs; GridFTP support will be added in future. This allows SGSs to be used with other remote service types and data sources. The URL is a simply a pointer to a data source and the downstream service does not care whether it represents a live data stream or a pre-existing file.

An important feature of data streaming is that a chain of services can execute concurrently: downstream services can start processing data while upstream services are still in progress. This can lead to large performance increases over systems that use intermediate files to transport data between services (in which the upstream service generally has to finish before the downstream service can start).

It is also possible to expose other output files as streams: the SGS wrapper monitors these files, looking for changes to their length as the service is running. In this way a service can effectively output many streams of data in addition to the standard output and error streams.

3.4 Status monitoring

The files in the `serviceData/` directory of an SGS instance exhibit a special behaviour that allows them to provide asynchronous updates of changes to the service data (e.g. progress and status updates). The first time a particular client sends a request to read an element of service data the server immediately returns the data to the client. The next time the same client sends a request to read from the same file (provided that the client has not closed the file) the server will not reply until the value of the service data element changes. In this way, the client can send read requests for any number of service data elements and be notified immediately when any of their values change.

On the SGS server, each element of service data is represented by a literal file on disk that is written to by the executable. The SGS wrapper monitors this file for changes and sends these changes to the clients. An alternative means of implementing this would be for the executable to use the Styx protocol to write to the appropriate file in the instance's namespace directly. However, this would in general require the executable to be modified, which may not always be possible or desirable.

3.5 Computational steering

Certain parameters of an SGS can be marked as "steerable" and each steerable parameter is backed by a file on the disk of the server. The underlying executable reads from this file as often as is necessary in order to pick up the current value of the parameter. Through the SGS interface, clients update the contents of this parameter file as the program is running. An alternative system could be used in which the SGS wrapper communicates directly with the executable using Styx, which would require the executable to understand the Styx protocol.

3.6 Styx Grid Services in JStyx

The JStyx library (http://jstyx.sf.net) provides useful tools for creating SGS servers and clients.

3.6.1 Creating an SGS server

Exposing existing binary executables as Styx Grid Services is straightforward with the JStyx library. Service providers first create an XML configuration file (figure 2) containing the location of the executables, the parameters that the users can set, the input files required, the service data that the service will expose, the input and output streams that will be accessed through the SGS interface and any documentation files. A utility program is then run that parses this configuration file and runs the SGS server. No further configuration is required.

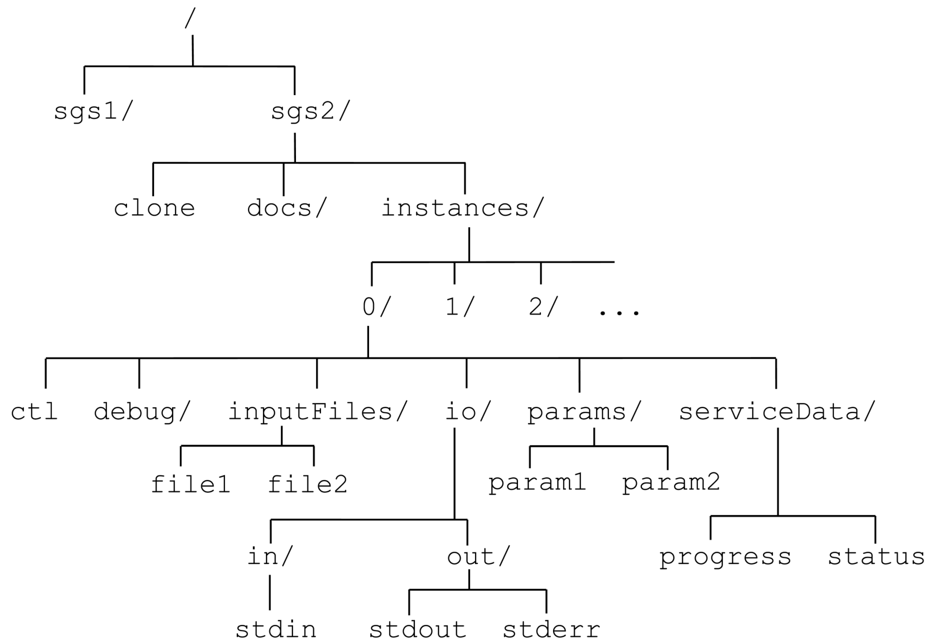

Figure 1: The virtual filesystem (namespace) exposed by a typical SGS server. Directories are denoted by a slash at the end of the name; not all directories are shown expanded. This server exposes two Styx Grid Services called `sgs1` *and* `sgs2`*, with the contents of the* `sgs2` *service being shown. Any file in the hierarchy can be represented by a URL: for example, the* `progress` *file of the first service instance of the* `sgs2` *service can be represented as* `styx://<server>:<port>/sgs2/instances/0/serviceData/progress`*. See text for an explanation of the purpose of each file.*

```
<gridservice name="lbflow"
  command="/path/to/lbflow -i input.sim"
  description="Lattice Boltzmann sim.">
  <streams>
    <outstream name="stdout"/>
    <outstream name="stderr"/>
  </streams>
  <serviceData>
    <serviceDataElement name="status"/>
    <serviceDataElement name="exitCode"/>
  </serviceData>
  <inputfiles allowOthers="yes">
    <inputfile path="input.sim"/>
  </inputfiles>
</gridservice>
```

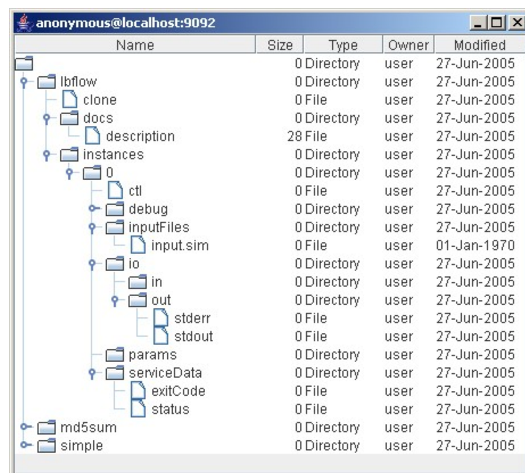

Figure 2: Left: Section of XML configuration file for the `lbflow` *Styx Grid Service. The file specifies the command to execute, the streams that are made available, the service data (state data) that can be monitored and the input files required. This service requires no command-line parameters. Right: View of the namespace of the resulting SGS server (cf. figure 1) using the StyxBrowser application (supplied with the JStyx library).*

3.6.2 Creating a client program

The simplest way for a client to interact with the SGS server is to use the SGS Explorer program that is provided with the JStyx library. Using this program, clients can view the services exposed by the server, browse the documentation, create new service instances and interact with these instances. The SGS Explorer automatically generates a GUI for each service. The GUI can ascertain (through the Styx interface) the parameters and input files that the service requires, as well as the streams and service data that can be read. The generated GUI then displays only those controls and indicators that are needed to interact with the SGS. The use of the SGS Explorer is illustrated in section 5.

The Taverna (http://taverna.sf.net) workflow system is capable of interacting with Styx Grid Services and can create workflows that are composed of a mixture of SGSs and other service types such as Web Services. Similar functionality is in the process of being built into Triana (http://trianacode.org). Custom client programs can be created using the API in the JStyx library.

3.7 SGS and Web Services

A Styx Grid Service can be called via a Web Services wrapper [2]. This allows any Web Service-based workflow engines to interact with SGSs (see section 4). The process is very simple: in addition to the SGS server, a Web Service server is run, containing Web Services that interact with the SGS server. The steps performed by the Web Service implementation are as follows (cf. section 3.2):

1. The client sends a SOAP message to the Web Service containing the name of the SGS to run, the parameters to pass to the service and the URL from which the input data are to be read.

2. The Web Service creates a new instance of the SGS, noting its identifier.

3. The Web Service sets the parameters of the SGS instance and the URL of the input data.

4. The Web Service starts the SGS running.

5. Without waiting for the SGS to finish, the Web Service returns the URL to the new service instance to the client.

Note that the Web Service will reply to the client almost immediately, even though the SGS instance might run for a considerable period of time. The URL to the new service instance that is returned to the client in the final step can be used to query service data such as progress, status and so forth. This could be done via another Web Service or directly using Styx.

This wrapping gives users the choice of accessing a SGS through SOAP messaging or directly through Styx. This principle could be easily extended in order to wrap a Styx Grid Service as a stateful WS-Resource for compatibility with WSRF (http://www.globus.org/wsrf).

3.8 Security

At the time of writing, the precise security model for the Styx Grid Service has not been decided upon. The flexibility of security in Styx (section 2.5) means that fine-grained control of access to the resources exposed by an SGS is possible. One could easily ensure, for example, that only the creator of a SGS instance can access its files. Alternatively, one could grant read-only access to all users so that others can monitor progress and see output data, but cannot stop the service running or steer it. Another model might allow anyone with a valid UK e-Science certificate to interact with an SGS instance.

4 Case study 1: Simple workflow

This case study was presented at the e-Science All Hands Meeting 2004 [1, 2]. It was based on an earlier version of the Styx Grid Service design, which was then known as the "Inferno Grid Service". Since then, the use of Inferno in the system has been replaced with the pure-Java JStyx libraries so the reference to Inferno is no longer appropriate. The principles behind the system, however, have remained identical.

In the case study, we use Triana to execute a simple distributed workflow of three services in order to calculate one-point correlation maps of oceanographic data (figure 3). The first service in the workflow extracts data from an archive. The data are streamed to (i.e. downloaded by) the second service, which applies a temporal filter to remove the effects of the seasonal cycle from the data. The filtered data are then streamed to a third service, which calculates the one-point correlation data. Finally, the correlation data are streamed back to a visualization applet in Triana. The volume of data transferred in this final step is much smaller than in the other steps so

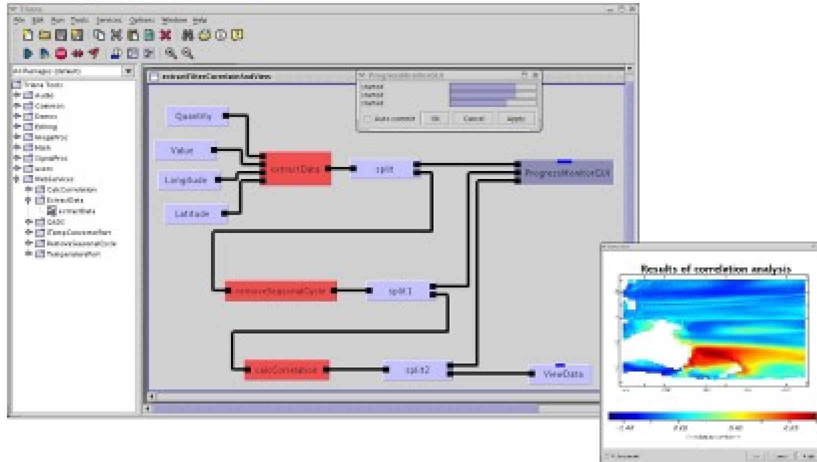

Figure 3: Screenshot of Triana executing a workflow of three SGSs that are wrapped as Web Services (red boxes) and displaying the results as a one-point correlation map (see section 4). A simple tool displays the progress and status of the running services; note that all three services are running concurrently.

it does not matter if the user has a slow internet connection. See [2] for more details.

We were able to use an unmodified version of Triana because we had wrapped each SGS in a Web Services wrapper in the manner described in section 3.7. In this way, neither the workflow engine nor the Web Service wrappers handled the relatively large amounts of data (tens of megabytes) that were streamed between the services.

5 Case study 2: Collaborative visualization

The power and convenience of the Styx Grid Service architecture can be shown by another case study. In this case study, we shall demonstrate that multiple clients can consume a data stream, allowing simultaneous visualization of a running simulation.

We created a Styx Grid Service from a pre-existing executable known as "`lbflow`", a Lattice Boltzmann simulation of fluid flow through porous media. Its scientific application is to investigate how gas flows under pressure through different types of permeable volcanic rock. If gas can flow easily (i.e. permeability is high) pressure can readily be dispersed, reducing the chance of occurrence of an explosive volcanic eruption. Conversely, a low permeability can lead to a build-up of pressure inside a volcano and hence increase the likelihood of an explosion.

The `lbflow` program reads an input file containing the size of the domain, the geometry of the pore spaces (as a set of triangulated surfaces) and other parameters of the simulation. It outputs its results (the velocity and pressure of the fluid at each point in the domain) as a VTK-format text file on its standard output stream. A companion viewer program ("`lbview`") reads these data through its standard input stream and generates a visualization of the results that is updated every time the simulation outputs a new set of results for a timestep. On a single (Unix-like) machine, the system can be run simply by piping the output of `lbflow` into the standard input of lbview (`lbflow -i input.sim | lbview`).

By wrapping the `lbflow` executable as a Styx Grid Service, the output data can be read over the Internet so that the viewer program can be run remotely from the simulation. Many viewers can read the output data simultaneously and produce a visualization of the output data. There is no difference from the viewer's point of view between downloading from a live stream and downloading from a static file. There is no (theoretical) limit on the number of viewers that can download data and hence collaboratively visualize the simulation. In this way, the data stream appears to be "forked" between different users.

Figure 4 shows the SGS Explorer application being used to interact with a remotely-running instance of `lbflow`. The SGS Explorer is used to connect to the server, create a new service instance, automatically generate a GUI for interacting with the instance and redirect the output of the service instance to the viewing program.

The `lbflow` executable is capable of being computationally steered in the manner described in section 3.5. Clients can adjust parameters such as the pressure drop across the domain and visu-

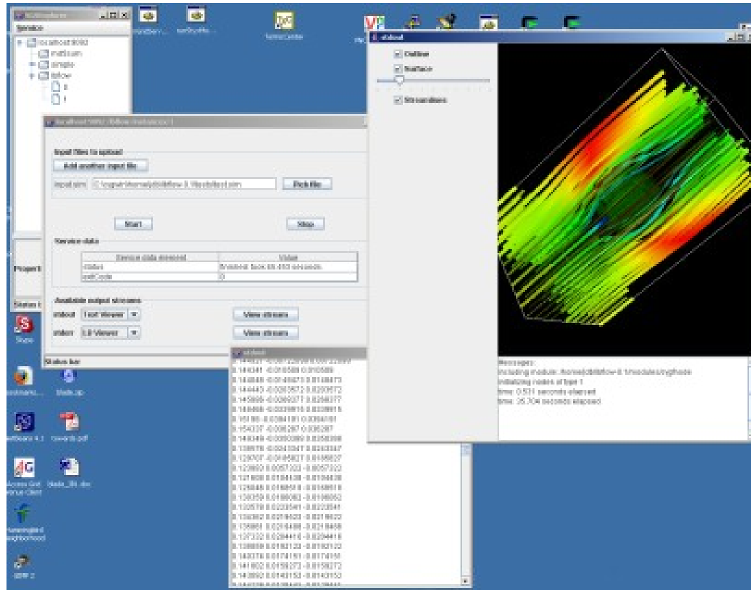

Figure 4: Screenshot of visualization of output from the `lbflow` *program using the SGS Explorer. The output stream is being viewed in two windows: firstly as a graphical interactive VTK window and secondly as a plain text stream, which is useful for debugging. The top-leftmost window is used for viewing the SGS server and creating new service instances, and the central window is used to set the parameters of the service, upload the input files and run the service.*

alize the results interactively. This allows the investigation of important phenomena such as the onset of turbulence in the gas as the Reynolds number is increased.

6 Conclusions

We have presented the Styx Grid Service, a remote service type that allows data to be streamed directly from service to service and the asynchronous monitoring of service data. It can be used for collaborative visualization and computational steering. It can be secured to a high degree and it can interoperate with other service types such as Web Services through standard protocols. SGS-based systems place few demands on firewalls. For the most up-to-date software and documentation, please see http://jstyx.sf.net.

Acknowledgements

The authors would like to thank Vita Nuova Holdings Ltd. for help with understanding the Styx protocol, Tom Oinn of the Taverna project for incorporating the SGS into the Taverna workbench and Trustin Lee of the Apache project for help with MINA, the architecture upon which the JStyx library is based.

References

[1] Blower, J., K. Haines, and A. Santokhee: 2004, Composing workflows in the environmental sciences using Inferno. *UK e-Science All Hands Meeting, Nottingham University*.

[2] Blower, J., A. Santokhee, K. Haines, R. Peppé, and C. Forsyth, 2004: Data streaming between Web Services using Inferno (poster). Online, http://www.resc.rdg.ac.uk/publications/posters/inferno_poster.ppt.

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

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

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