

.....

JStyx v.0.2.0

Project Documentation

Table of Contents

1 JStyx	
1.1 Overview	1
1.2 Downloads	2
1.3 FAQs	4
1.4 Utilities	6
1.5 Tutorial	7
1.5.1 1. First Styx system	8
1.5.2 2. Reading and writing	11
1.5.3 3. Custom files	16
1.5.4 4. Next steps	18
1.5.5 5. Asynchronous files	19
1.5.6 6. Streams	20
2 Styx Grid Services	
2.1 Overview	21
2.2 Tutorial	23
2.2.1 Hello World	24
2.2.2 More config options	27
2.2.3 More complex services	29
2.2.4 Getting help	33
2.2.5 Workflow and scripting	35
2.2.6 Advanced stuff	39
2.2.7 Security	41

2.3 The configuration file	42
2.4 The SGS namespace	49
3 JStyx References	
3.1 JavaDocs	54
3.2 Source code	55
3.3 Sf.net site	56

1.1 Overview

What is JStyx?

JStyx is a pure-Java implementation of the [Styx](#) protocol for distributed systems. The JStyx libraries also include an implementation of the [Styx Grid Services](#) (SGS) system for building stream-oriented services and workflows.

What is Styx?

[Styx](#) is used in the [Inferno](#) and [Plan9](#) operating systems. It is essentially a file-sharing protocol, similar in many respects to NFS.

More information

The [FAQs](#) are a good place to start for more information. To learn how to use the JStyx software, see the [tutorial](#) . If you want to ask any questions, please use the [mailing lists](#) .

1.2 Downloads

Downloading and installing JStyx

The process of installing the JStyx software (which includes the [Styx Grid Services](#) software) is very simple on all platforms. In summary:

- Install a Java runtime environment and add the `java` program to your `PATH`.
- Download the JStyx distribution and unpack it.
- Set the `JSTYX_HOME` environment variable and add `JSTYX_HOME/bin` to your `PATH`.

You can either follow the instructions below or just go straight to the [downloads page](#) if you are impatient! ;-)

1. Set up Java

JStyx is a set of Java classes and so we need a Java runtime environment to run the software:

1. Download and install a Java runtime environment version 1.4.2 or above from [here](#) .
2. Add the `bin` directory of the Java installation to your `PATH` environment variable.
3. Test this by opening a new terminal window or command prompt and typing `java -version`. The version number of the Java installation should be printed to the console. This should work from any directory on your system.

2. Install JStyx

The JStyx software is essentially the same on all platforms because it consists of a set of platform-independent Java libraries. The only difference between the different distributions is the scripts that are used to launch JStyx programs (these scripts are `.bat` files under Windows and shell scripts under Unix and are found in the `bin/` directory of the distribution).

Windows

1. Download the software in zip format from [here](#) .
2. Unzip the software into the directory of your choice (e.g. `C:\JStyx`).
3. Set the environment variable `JSTYX_HOME` to the location of the JStyx software (e.g. `C:\JStyx`).
4. Append the directory `%JSTYX_HOME%\bin` to the `PATH` environment variable.

Note: to set an environment variable in Windows, right-click on "My Computer" and click the "Advanced" tab. Click on "Environment Variables". A Google search will reveal more information if you need it.

Unix/Linux

(Note: these instructions might also work on Mac OSX, but this hasn't been tested yet.)

1. Download the software as a gzipped tar archive from [here](#) .
2. Unpack the software into the directory of your choice (e.g. `cd /home/jon/JStyx; tar xzf jstyx-0.2.0-unix.tar.gz`).
3. Set the environment variable `JSTYX_HOME` to the location of the JStyx software (e.g. `export JSTYX_HOME=/home/jon/JStyx`).
4. Append the directory `$JSTYX_HOME/bin` to the `PATH` environment variable (e.g. `export PATH=$PATH:$JSTYX_HOME/bin`).

If you use the C shell you can set an environment variable like this: `setenv JSTYX_HOME /home/jon/JStyx`.

Add the `JSTYX_HOME` and `PATH` environment variables to your startup scripts so that you do not have to set them every time you want to run the JStyx software.

3. Test the installation

To verify that your installation is working, open a new terminal window (or command prompt) and enter `SGSRun` with no arguments. If all is well, you should just see a usage message:

```
Usage: SGSRun <hostname> <port> <servicename> [args]
```

If you get an error message saying that the `SGSRun` program cannot be found, you have not set your `PATH` correctly. If you get an error message saying "NoClassDefFoundError" you have not set your `JSTYX_HOME` environment variable correctly.

Access to source code

You can get a source distribution from [here](#) . The `build-instructions.txt` file in this distribution contains instructions on how the software can be built from this source.

If you want the very latest version of the source code (bearing in mind that it probably will not have been tested properly, if at all!) you can get the code from Sourceforge's Subversion server at <https://svn.sourceforge.net/svnroot/jstyx/trunk/core>. Anybody can download the entire code base, but only developers can check changes back in.

1.3 FAQs

Frequently Asked Questions

General

1. [What is JStyx?](#)
2. [What can I use JStyx for?](#)
3. [Why should I use JStyx for my distributed system?](#)
4. [Where can I go for help?](#)

Download and installation

1. [How do I install JStyx?](#)
2. [What is the licence?](#)

Styx Grid Services

1. [What are Styx Grid Services?](#)

General

General

What is JStyx?

JStyx is a pure-Java implementation of the [Styx](#) protocol for distributed systems. Styx is used by the [Inferno](#) and [Plan9](#) operating systems. Styx is essentially a file-sharing protocol; it is similar in many respects to NFS but in Styx systems, files do not always represent bytes on the hard disk. They may represent a chunk of RAM, a physical device such as the screen, the interface to a device such as a digital camera or the interface to a program.

Both the Inferno and Plan9 operating system virtualize all resources as **files** and both use Styx as the protocol for accessing all these files, irrespective of the underlying resource they represent and their location (local or remote). Applications in Inferno and Plan9 do not know the difference between local and remote files: the underlying operating system routes all Styx messages to the correct location. Therefore the creation of distributed systems with Inferno and Plan9 is very easy. The idea behind JStyx is to allow similarly easy development of distributed applications in other operating systems.

What can I use JStyx for?

There are several potential uses of JStyx in the field of distributed systems:

- Java interface to Inferno/Plan9 systems (e.g. web interface to Inferno Grid)
- Standalone clients and servers (e.g. messaging systems). See JStyx tutorial.

- Styx Grid Services. See SGS tutorial.

Why should I use JStyx for my distributed system?

Lots of reasons:

- Lightweight - Styx messages are very short and add little bloat to the payload
- Built on robust framework ([Apache MINA](#))
- Platform independence
- Secure (in future! not in current public release)

Where can I go for help?

The first place to go for information is the [JStyx website](#) , which is where you are probably reading this FAQ ;-). You can join the [mailing lists](#) : the jstyx-users mailing list is the one to use for posting questions about all aspects of the use of JStyx. Please check the [archives](#) (and read through this FAQ!) before posting a new question.

Download and installation

Download and installation

How do I install JStyx?

See the instructions [here](#) .

What is the licence?

The JStyx software is released under a BSD-style Open Source licence: see [here](#) for the full licence text. Essentially you are allowed to do anything with the software, provided that you include the licence text with any redistribution.

Styx Grid Services

Styx Grid Services

What are Styx Grid Services?

See [these pages](#) .

1.4 Utilities

Useful utilities

The JStyx library comes with a set of useful utilities to aid with creating and debugging Styx applications.
TO BE CONTINUED.

StyxBrowser

StyxMon

1.5 Tutorial

Tutorial Introduction

This tutorial will guide you through the process of creating distributed applications using the JStyx library. We will start with very simple systems and build up to more complex ones, showing how even quite sophisticated applications can be created with little effort.

This is the tutorial for the JStyx library. The tutorial for the Styx Grid Services system is [here](#) .

Basic concepts

As you may be aware, in Styx systems *all* resources are represented as one or more virtual files. These resources may be literal files on disk, chunks of RAM, databases, physical devices or interfaces to programs. A Styx server may serve up any number of files in a hierarchical fashion, very much like a filesystem on a hard disk. In Styx, this hierarchy of virtual files is called a *namespace*.

In essence, the creation of a Styx server is very simple. You assemble a hierarchy of files and directories, then run a server program that listens for connections on a given port. Clients can then make connections to this server and perform standard file operations on the files in the namespace that the server is exposing. Most of these file operations will be very familiar: opening and reading files and directories, creating new files, writing and appending to existing files and so forth. All these operations are handled using high-level API calls in the JStyx library and you will never need to know the nuts and bolts of the Styx protocol. (If you do want to know more about the Styx specification, see <http://www.vitanuova.com/inferno/man/5/INDEX.html>).

Tutorial contents

You can follow this tutorial online or, if you prefer to work from a printed copy, you can download a [PDF version](#) of this entire website. The tutorial sections are:

- [Your first Styx system](#)
- [Reading and writing Styx files](#)
- [Creating new types of file](#)
- [More complex namespaces](#)
- [Asynchronous files](#)
- [JStyx and data streaming](#)

1.5.1 1. First Styx system

Your first Styx system

In this tutorial you will create a basic Styx server and client. This will introduce you to the main classes of the JStyx software and how they are used.

A very simple Styx server

We will create a Styx server that serves up a single file. The contents of the file are held in memory on the server. The namespace of this system is extremely simple:

```
    /      (The root of the namespace)
    |
  readme  (The only file exposed by the server)
```

The full source is contained in the [SimpleServer](#) class, but these are the important lines (see full source for comments):

```
StyxDirectory root = new StyxDirectory("/");
InMemoryFile file = new InMemoryFile("readme");
file.setContents("hello");
root.addChild(file);
new StyxServer(9876, root).start();
```

In these five lines, we create a root directory for the namespace, then create and add a file with the name "readme" that contains the string "hello". Note that the file is an [InMemoryFile](#), which is an instance of the general superclass for all files on a Styx server, [StyxFile](#). Finally, we create and start a Styx server, passing it the root of the namespace.

You can run the server by changing to the bin directory of your JStyx installation and running:

```
JStyxRun uk.ac.rdg.resc.jstyx.tutorial.SimpleServer
```

(The JStyxRun script sets up the classpath, then runs the main method of the provided class.) You will probably see some logging messages printed to the console.

A very simple Styx client

It is just as simple to write a client program that can read the contents of the file exposed on the server. The full source is in the [SimpleClient](#) class, but these are the most important lines:

```
StyxConnection conn = new StyxConnection("localhost", 9876);
try
{
    conn.connect();
    CStyxFile readmeFile = conn.getFile("readme");
    System.out.println(readmeFile.getContents());
}
catch (StyxException se)
{
    se.printStackTrace();
}
finally
{
    conn.close();
}
```

We create a [StyxConnection](#) to the server and call the `connect()` method to make the connection and perform the relevant handshaking. (Note that you might need to edit the hostname and port to suit your system.) We then get a handle to the "readme" file: this handle is an instance of the [CStyxFile](#) class. (The "C" means "Client", to avoid confusion with the server-side [StyxFile](#) class.) We read the contents of the file as a String, then print them out. Finally, we close the connection.

You can run the client by changing to the `bin` directory of your JStyx installation and running:

```
JStyxRun uk.ac.rdg.resc.jstyx.tutorial.SimpleClient
```

You should see the string "hello" printed out, perhaps in amongst some logging messages.

Serving up files on disk

TODO (talk about the `FileOnDisk` and `DirectoryOnDisk` classes)

Summary

In this section of the tutorial, we have created a simple Styx server and client and have passed some data between them. From the server point of view, the key classes are [StyxFile](#), which is the superclass for all virtual files on a Styx server, and the [StyxServer](#) class itself. In client-side code, the most important classes are the [StyxConnection](#) class, which represents the connection to the server, and the [CStyxFile](#) class, which we use to interact with files on the server.

In the [next section](#) of the tutorial we will look at different ways of reading from and writing to Styx files.

1.5.2 2. Reading and writing

Reading and Writing Styx files

The most common tasks (from the client's point of view at least) in a Styx system are reading from and writing to files. There are several ways to do this, each with advantages and disadvantages. In this section of the tutorial, we'll go through the options.

getContents() and setContents()

The easiest way to read from and write to files is to use the `getContents()` and `setContents()` methods, as used in the [SimpleClient](#) from earlier in this tutorial. These methods are suitable if the entire contents of the file can fit sensibly in a `String`, i.e. for relatively small data volumes.

Once you have a handle to a [CStyxFile](#) object, you can call `setContents()` and `getContents()` to write and read the entire contents of the file as `Strings`:

```
file.setContents("hello JStyx world");
System.out.println(file.getContents());
```

Note that both `setContents()` and `getContents` can throw [StyxException](#)s and so you will have to catch this or re-throw it from the method. If you run this code the string "hello JStyx world" should be printed out. (Try running the [SimpleServer](#) again and try this out. You can adapt the [SimpleClient](#) class to produce the client code.)

InputStreams and OutputStreams

Another easy-to-use option for reading and writing is through streams. This is probably one of the most familiar ways of dealing with I/O to Java programmers. In essence, once you have a [CStyxFile](#) object you can turn it into an `InputStream` or `OutputStream` by using the wrapper classes [CStyxFileInputStream](#) and [CStyxFileOutputStream](#) respectively. You can then use standard stream I/O to get data from and to the files on the Styx server.

Character-based I/O can be achieved by further wrapping these streams in [CStyxFileInputStreamReader](#) and [CStyxFileOutputStreamWriter](#) objects. These convert the streams into character streams by using the UTF-8 character set. These Readers and Writers can then be wrapped yet again as `BufferedReader`s and `BufferedWriter`s to allow, for example, reading and writing data a line at a time from a remote file.

Using URLs to get handles to streams

You can get a handle to a Styx file on a remote server using a URL. For example, the URL of a file called `readme` in the root directory of a Styx server on `localhost`, port `9876` would be `styx://localhost:9876/readme`. You can use this URL to get an `Input-` or `OutputStream` for interacting with this file, as in this code snippet:

```
URL url = new URL("styx://localhost:9876/readme");
InputStream is = url.openStream();
OutputStream os = url.openConnection().getOutputStream();
```

Note that you do not have to instantiate or open a [StyxConnection](#) before you do this. This is done automatically in the protocol handler for the `styx://` URLs.

In order to make Java recognize `styx://` URLs, you have to add the string `uk.ac.rdg.resc.jstyx.client.protocol` to the system property `java.protocol.handler.pkgs`. This is done automatically by the `JStyxRun` script in the `bin/` directory of the `JStyx` distribution. If you don't set this property, you will get `MalformedURLExceptions` when trying to create URL objects from `styx://` URLs.

download() and upload()

The `download()` and `upload()` methods of the [CStyxFile](#) class provide convenient methods for copying data from a remote Styx file to a local `java.io.File` or vice-versa.

Some technical details

The above methods of reading and writing completely hide the details of the Styx protocol mechanisms from the user. In order to understand the remainder of this section of the tutorial, you will need to know a little about how Styx works.

The most important thing you need to know is that when you read from - or write to - Styx files, you do so in chunks. When you read from a file, you are actually making lots of individual requests for data. By default, `JStyx` reads and writes data a maximum of 8KB at a time. So, if you are downloading a file of 1MB in size, you are actually making at least 128 separate requests for 8KB of data. (It is possible to choose a different maximum message size at the point of making a connection to a server: see the various constructors for the [StyxConnection](#) class. However, it is generally recommended to stick with the default message size unless you know what you're doing.)

When the server receives a request for a chunk of data, it can respond with a chunk of *any* size from zero bytes to the requested chunk size. If the server responds with zero bytes, this means that the end of the file has been reached. Clients can make requests for any chunk size up to the maximum allowable on the connection.

This feature of the Styx protocol has several advantages, including the fact that it is easy to download data from arbitrary positions in the remote file. However, it means that reading and writing large amounts of data are rather slower than with a system (e.g. HTTP) that simply opens a socket connection and passes the data in one long stream. The speed can be significantly increased by selecting a larger maximum message size when making the connection to the server (64KB is suggested as a maximum) or by using

an "accelerated download" by making several simultaneous read requests, thereby attempting to saturate the connection (see the `download(File file, int numRequests)` method). However, Styx file transfer rates generally do not exceed HTTP transfer rates for static files.

read() and write()

There may be situations in which you want to have more control over the reading and writing of files: perhaps you want to read or write data from or to a specific position in the remote file. In this case you can use the `read()` and `write()` methods of [CStyxFile](#).

The `read()` method takes as an argument the offset (position) in the remote file from which you wish to read data. It returns a [ByteBuffer](#) of data, but this is not the normal `java.nio.ByteBuffer` to which you might be accustomed, although it is very similar. This is a `ByteBuffer` from the [MINA](#) framework, which is the networking software that underlies JStyx. MINA `ByteBuffers` are obtained from a pool and returned to the pool when they are no longer needed. This means that `ByteBuffers` are not continually being created and garbage-collected. This gain in efficiency comes at a price: when using the `read()` method of [CStyxFile](#) you must remember to call the `release()` method on the `ByteBuffer` that is returned, once you have finished with the data.

There are a few versions of the `write()` method. In each case you provide a byte array containing the data to write and specify the position in the remote file where you want the data to go. You can also specify whether you want the remote file to be truncated at the end of the data. If the byte array that you provide is larger than the maximum message size... *[TODO: I don't think JStyx checks for this at the moment!]* To save you worrying about how big the input array is, the `writeAll()` method allows you to write an array of any size: the data in the array will be split across several messages if necessary.

When using the `read()` and `write()` methods, the file is opened automatically in the correct mode. However, you should remember to `close()` the file when you have finished with it.

Asynchronous reading and writing

So far, all the methods we have used have been synchronous in nature. That is to say, the methods only return when their job is done. However, there may be situations in which there may be a significant time gap between sending a read request and actually getting the data back: this may not be because of a slow server, but by deliberate design of the Styx system (see the section of the tutorial on [asynchronous files](#) for example). Also, when writing graphical programs, you will want to keep the user interface responsive and it will be undesirable to have your program hang while waiting for data. You can solve this by firing off lots of threads but there is a neater way: use the asynchronous versions of the reading and writing methods.

There are a couple of ways of doing asynchronous reading and writing, but both are based on the idea that you send the read and write message using one method, which returns immediately, leaving your program to do other things. When the reply arrives, a specified callback method is called so that you can deal with it.

Using a change listener

The first way to use asynchronous reading and writing is by creating a class that implements the

[CStyxFileChangeListener](#) interface. (Or, for convenience, you might choose to subclass the [CStyxFileChangeAdapter](#) abstract class, which provides empty default implementations of all the methods in the interface.)

Having got a [CStyxFile](#), you register your change listener using the `addChangeListener()` method. Then you call one of the `...Async()` methods (e.g. `readAsync()`) and the relevant method in the change listener will be called when the reply arrives. For example, here is a code snippet that will read a file from a remote server:

```
public class DataReader extends CStyxFileChangeAdapter
{
    ...
    public void readFile(CStyxFile file)
    {
        // Register this object as a change listener
        file.addChangeListener(this);
        // Read the first chunk of data from the file
        file.readAsync(0);
        // This returns immediately
    }
    ...
    public void dataArrived(CStyxFile file,
        TreadMessage tReadMsg, ByteBuffer data)
    {
        // This method is called when the data arrive. The arguments to
        // this method contain the file that is being read, the original
        // read message and the data themselves.
        if (data.hasRemaining())
        {
            // We got some data back. Work out the offset (file position)
            // of the next chunk
            long offset = tReadMsg.getOffset().asLong() + data.remaining();
            // ... (Do something with the data here)
            // Now read the next chunk of data. This method will be
            // called again when the data arrive.
            file.readAsync(offset);
        }
        else
        {
            // We have reached end of file. Close the file.
            file.close();
        }
    }
    ...
}
```

Writing data is very similar, except that you use the `writeAsync()` method and, when the write confirmation arrives, the `dataWritten()` method of all registered change listeners will be called. These are all the asynchronous methods with their relevant callbacks in the [CStyxFileChangeListener](#) interface:

Note that errors from all asynchronous methods are caught in the `error()` method of the change listener.

One Golden Rule

When implementing callback functions (such as `dataArrived()`), you must be very careful to avoid using non-asynchronous (blocking) methods such as `read()` and `write()`. This will cause deadlock (you will block the thread that dispatches Styx replies). You can only use asynchronous methods within callback functions. The Javadoc comments for each function will tell you whether a method blocks, but in general, only methods called `xxxAsync()` will be guaranteed not to block. An exception to this is the `close()` method, which never blocks (it doesn't wait for the reply to the close request).

Using MessageCallbacks

Sometimes you might not want to use a [CStyxFileChangeListener](#): perhaps you want more control over individual Styx messages or you don't like the way that all errors are caught in the same `error()` callback in the change listener. In this case, you can create individual callback objects for each call to an asynchronous method.

To do this, you create an instance of the [MessageCallback](#) abstract class. This requires you to implement two methods: `replyArrived()`, which is called if the operation succeeds; and `error()`, which is called if an error occurs. (The `error()` callback is equivalent to the throwing of a [StyxException](#) in the synchronous methods). The following example will set the contents of the remote file to the given String (i.e. the asynchronous equivalent of `setContentts()`):

```
public void writeString(CStyxFile file, String str)
{
    // Write the string to the beginning of the file (offset=0).
    // The file will be truncated at the end of the string
    file.writeAsync(str, 0, new WriteStringCallback());
}
private class WriteStringCallback extends MessageCallback
{
    public void replyArrived(StyxMessage rMessage, StyxMessage tMessage)
    {
        // The arguments to this method are the request (the tMessage)
        // and the reply (the rMessage), but we don't always use them.
        System.out.println("Write confirmation arrived");
    }
    public void error(String errString, StyxMessage tMessage)
    {
        // The arguments to this method are the request (the tMessage)
        // and the error string
        System.err.println("An error occurred: " + errString);
    }
}
```

There are a number of `writeAsync()` methods that can be used: see the [code](#) or the [Javadoc](#) for the `CStyxFile` class.

1.5.3 3. Custom files

Custom files: Introduction

In the [first section](#) of this tutorial, we created a very simple Styx system which exposed a single file, an [InMemoryFile](#) that simply represented a section of RAM. Similarly, the [FileOnDisk](#) class is used to create a Styx file that represents a literal file on the local filesystem. The key to creating powerful distributed systems using Styx is to design and construct new types of virtual files that exhibit the correct behaviour.

In this section of the tutorial, you will learn how to create customized virtual files. In essence, this simply involves creating a subclass of the [StyxFile](#) class and overriding key methods such as `read()` and `write()`.

Custom file 1: WhoAmI

For our first example, let's create a file that, when read, will return the IP address and port number of the client that is making the connection. This file will therefore return data that is different for each client that is connected. This will be a read-only file. We'll call this class "WhoAmIFile". (See the full source code of the [WhoAmIFile](#) class, including full comments, [here](#).)

As always, we need to subclass the [StyxFile](#) class:

```
public class WhoAmIFile extends StyxFile
```

Note that the [StyxFile](#) class is not abstract: it provides methods to give (not very useful) default behaviour. Now we need to create a constructor:

```
public WhoAmIFile() throws StyxException
{
    super("whoami");
    this.setPermissions(0444);
}
```

The call to the superclass constructor sets the name of the file. Note that the superclass constructor throws a [StyxException](#) if the file name is illegal. We know that this is not the case here, but we will simply re-throw the exception anyway. Then we set the permissions of the file: we will not allow writing to this file, so we give it read permissions only (0444, i.e. `r--r--r--`).

Now we must override the `read()` method so that the IP address and port are returned to the client. This is very easily done:

```
public void read(StyxFileClient client, long offset, int count, int tag)
    throws StyxException
{
    String clientAddr = client.getSession().getRemoteAddress().toString();
    this.processAndReplyRead(clientAddr, client, offset, count, tag);
}
```

In the first line of this method we get the client's IP address and port as a `String`. Then we call `processAndReplyRead()` to return the data to the client.

Replying to the client

In the above example, we used the `processAndReplyRead()` helper method to process the read request and return the data to the client. This is a very useful method that is used when the *entire* contents of a file can be represented as a `String`, byte array or `ByteBuffer`. If the file cannot be represented in this way we have to work a little harder, as we shall see in the example below. In this case, we have to work out exactly what data we need to give to the client (based on the client's read request and the contents of the whole file) and call one of the `replyRead()` methods.

A read/write file

The above example implemented a read-only file...

1.5.4 4. Next steps

Next steps

In the [first section](#) of this tutorial, we created a very simple Styx system which exposed a single file, an [InMemoryFile](#) that simply represented a section of RAM. In this tutorial we will create a more complex namespace that includes many different resources that are exposed as Styx files.

More file types

Files on disk

As you might expect, it is easy to represent a file on the local filesystem as a Styx file. We do this using the [FileOnDisk](#) class that is provided with the JStyx library. As with all files that can be exposed in a Styx namespace, the [FileOnDisk](#) class inherits from the [StyxFile](#) class. Creating a [FileOnDisk](#) is very easy: the [source code](#) contains all the possible constructors, but the easiest way is simply to use the full path of the file, for example:

```
FileOnDisk localFile = new FileOnDisk("C:\\myfolder\\myfile");
```

Directories on disk...

1.5.5 5. Asynchronous files

Blocking files

An important concept in Styx is that, when a client sends a message to read from (or write to) a file, the reply need not be sent immediately. TO BE CONTINUED.

1.5.6 **6. Streams**

Data streaming using JStyx

TO BE CONTINUED

2.1 Overview

What are Styx Grid Services?

Styx Grid Services (SGSs) are a means for wrapping command-line programs and allowing them to be used remotely over the Internet. When deployed as an SGS, a program can be run from anywhere on the Internet *exactly as if it were a local program*.

[This poster](#) is a good summary of the main capabilities of the SGS system. [This paper](#) is a more detailed description of the capabilities of the SGS system.

Why use the SGS system?

Styx Grid Services are useful when you want to run a program on a machine that is not your own (they are somewhat analogous to Web Services). You can then run the program from anywhere on the Internet exactly as if it were a local program. You might want to do this because:

1. The program requires a different operating system or architecture from your own machine
2. The program requires a larger amount of memory or processing power than you have on your own machine
3. The program requires access to a data store that you cannot access from your local machine
4. You want to allow other people to run the program from elsewhere on the Internet

Styx Grid Services are very simple to install and use and require a minimum of software (just a Java virtual machine and the JStyx libraries).

Styx Grid Services and workflows

SGSs can be composed into "workflows", in which a number of SGSs, perhaps in different locations, can be combined using very simple shell scripts to create distributed applications. Data can be streamed directly between the services along the shortest network path.

Let us consider a simple distributed application, consisting of two Styx Grid Services. The first is called `calc_mean` and reads a set of input files from a set of scientific experiments, calculates their mean and outputs the result as a file. The second SGS is called `plot` and it might be deployed in a completely different location from the first service. It takes a single input data 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
```

The important thing to note is that this is *exactly the same script* as would be used to run the programs if they were installed locally. The input files for each service have been detected and uploaded automatically and the output files have been automatically downloaded.

The intermediate file `mean.dat` can be *passed directly between the two services* (i.e. without being downloaded by the client) with a small change to the script:

```
calc_mean input*.dat -o mean.dat.sgsref
plot -i mean.dat.sgsref -o graph.gif
```

Getting Started

Download and install the JStyx software, as described on the [downloads](#) page. Then follow the Styx Grid Services [tutorial](#).

Further reading

Here are some publications about the SGS system in (roughly) decreasing order of usefulness:

Jon Blower, Andrew Harrison, Keith Haines, **Styx Grid Services: Lightweight, easy-to-use middleware for scientific workflows**, accepted for oral presentation at the *International Conference on Computer Science* 2006, and for publication in *Lecture Notes in Computer Science*. [\[download paper\]](#)

Jon Blower, Keith Haines, Ed Llewellyn, **Styx Grid Services: lightweight, easy-to-use middleware for e-Science**, Presented in the UK e-Science booth at *SuperComputing*, Seattle, 15-17 November 2005 [\[download presentation\]](#) [\[download poster\]](#)

Jon Blower, Keith Haines, Ed Llewellyn, **Data streaming, workflow and firewall-friendly Grid Services with Styx**, *Proceedings of the UK e-Science All Hands Meeting* 19-22 September 2005 [\[download paper\]](#) [\[download presentation\]](#)

2.2 Tutorial

Styx Grid Services Tutorial: Introduction

This tutorial will guide you through the process of creating Styx Grid Services from command-line programs and using them in a number of ways. Before you start the tutorial you must install the JStyx software as instructed [here](#) . No further software is required to follow this tutorial: all the Styx Grid Services software is included in the JStyx distribution.

More background information on the Styx Grid Services system can be found [here](#) .

The Styx Grid Services software is built on top of the JStyx core libraries. For more information on JStyx, please see [here](#) .

Basic concepts

In order to create a Styx Grid Service from an existing command-line program you need to write a short, machine-readable description of the program, including the input files it expects, the output files it produces and the command-line arguments it understands. This description is known as the "configuration file" and is written in XML. A full description of the format of this file can be found [here](#) , but for the moment you don't need to worry about this: the tutorial material includes an example configuration file and this is all we need for this tutorial.

Having created the configuration file, you will simply run a program that reads the file and automatically creates the Styx Grid Services server. You can then run *any* Styx Grid Service from the command line using a generic client program.

Tutorial contents

You can follow this tutorial online or, if you prefer to work from a printed copy, you can download a [PDF version](#) of this entire website. The tutorial sections are:

- [Wrapping the traditional "Hello World" application as a Styx Grid Service](#)
- [Brief aside: more configuration options](#)
- [Wrapping more complex applications as Styx Grid Services](#)
- [Getting help](#)
- [Workflow and scripting in SGS systems](#)
- [Advanced techniques](#)
- [Securing the SGS system](#)

2.2.1 Hello World

Styx Grid Services Tutorial: Hello World

In this first part of the tutorial we shall wrap the traditional "Hello World" program as a Styx Grid Service and execute it remotely. In order to do this we need to go through the following steps:

1. Create a machine-readable description of the program (the configuration file).
2. Create and run the SGS server, using this description.
3. Run the service using the SGS client software.

1. Creating the configuration file

The first thing that we need to do is to create a configuration file that contains a complete description of the HelloWorld program and how to run it. The HelloWorld program reads no input files or command line arguments and simply prints a "Hello World" message to the console (i.e. to the standard output stream).

The configuration file for this service is therefore very simple but you don't have to write it yourself. The JStyx distribution contains not only a suitable configuration file, but also a [HelloWorld program](#), implemented in Java. The provided configuration file (the `SGSconfig.xml` file in the `conf` directory of the distribution) contains descriptions of all the Styx Grid Services that are included in this tutorial but we are only interested in the part that describes the `helloworld` service for the moment:

```
<gridservice name="helloworld"
  command="JStyxRun uk.ac.rdg.resc.jstyx.gridservice.tutorial.HelloWorld"
  description="Prints Hello World to stdout">
  <outputs>
    <output type="stream" name="stdout" />
  </outputs>
</gridservice>
```

This simply specifies that the HelloWorld program is run using the command "JStyxRun uk.ac.rdg.resc.jstyx.gridservice.tutorial.HelloWorld" and that it prints data to its standard output stream. (*Important note for Windows users: the JStyxRun "program" is actually a batch file and so cannot be run directly with this command. You will need to change the `command` property for the `helloworld` SGS to "`cmd.exe /C JStyxRun uk.ac. ...`". This is because batch files are not executable on their own: they need to be run using the command prompt program.*)

2. Run the server

We now need to run the SGS server. The `GridServices` script runs the [server](#), passing in the `SGSconfig.xml` file. Assuming you have set your `PATH` correctly as described in the [installation](#)

[instructions](#) , you can run the server simply by entering:

```
GridServices
```

at a command-line prompt. If all is well, you will see some debug messages printed to the console, including the line "Creating StyxGridService called helloworld". The final message will read something like "Started StyxGridServices, listening on port 9092". See [below](#) for instructions for changing this port number if the default port of 9092 is not acceptable for any reason.

Note that you should **not** run the SGS server as the root user for security reasons.

3. Run the service using the SGSRun program

We shall now execute the helloworld service. The SGS distribution includes a Java program called SGSRun, which is a generic client program for *any* Styx Grid Service. It is run from the command line like so:

```
SGSRun <hostname> <port> <servicename> [args]
```

Open a new command prompt or terminal window. Assuming that you are working on the same machine that is running the SGS server, you can run the helloworld service by entering:

```
SGSRun localhost 9092 helloworld
```

You should see the "Hello World!" message printed to the console window. The SGSRun program has connected to the server, created a new instance of the helloworld service, run the service and downloaded the output data, which in this case was just the "Hello World!" string.

4. Create a wrapper script

One of the most important features of the SGS system is the ability to run remote services exactly as if they were local programs. The SGSRun program gets us most of the way there but we still have to specify the location and port number of the server. Assuming that these are fixed, it is an easy task to create a script that wraps the SGSRun program.

On Windows, we can create a batch file called helloworld.bat with the following contents:

```
@echo off
SGSRun localhost 9092 helloworld
```

On Linux/Unix the file would be a shell script called helloworld:

```
#!/bin/sh
SGSRun localhost 9092 helloworld
```

The `helloworld` wrapper script can now be treated exactly as if it were the HelloWorld program itself.

5. Exercise: Provide your own Hello World program

For convenience, an example HelloWorld program (written in Java) is provided with the SGS distribution. This is what we have been using so far. As an exercise, you might like to write a HelloWorld program in your language of choice and run that as an SGS instead of the Java program. For example, if you write and compile a C program called `helloworld` and place it in the `/usr/local/bin` directory, you would change the relevant part of the configuration file to:

```
<gridservice name="helloworld"
  command="/usr/local/bin/helloworld"
  description="Prints Hello World to stdout">
  ...
</gridservice>
```

and stop and restart the server process.

Java programmers might like to know that the string in the `command` attribute of the `<gridservice>` tag is passed directly to Java's `Runtime.getRuntime().exec()` method. The main thing to watch out for is that `.bat` files under Windows cannot be executed directly: they must be passed to the `cmd.exe` (Windows 2000/XP) or `command.com` (Windows 9x) program and so the `command` attribute must read as something like:

```
command="cmd.exe /C C:\programs\myscript.bat"
```

2.2.2 More config options

Styx Grid Services Tutorial: More configuration options

Normally you will not need to change the configuration file to follow this tutorial. However sometimes this is necessary to fit in with your system.

Changing the port number of the server

By default, the SGS server will run under port 9092. If this is not acceptable to you for any reason, you can change this in the configuration file. Look at the config file provided with the JStyx distribution (SGSconfig.xml). Before the part that contains the details of all the services, there is a section for configuring the server:

```
<server port="9092"></server>
```

Simply change the port number in this section to the number of your choice (the range of port numbers that you are allowed to choose is system-dependent).

Note that the SGS server only runs under a single port. Therefore, this is the only port that you will need to open through any firewalls that the server happens to lie behind. SGS clients need **no** incoming ports open: they simply need to be able to make an outgoing connection to the server.

Changing the location of cached files

While a Styx Grid Service is running, it creates a number of files on the server's hard disk. These files are mostly cached copies of the input and output files. By default, these files are kept in a directory called `StyxGridServices` in the home directory of the user that is running the SGS server (this is detected through Java's `user.home` system property. Under Unix-type systems this will be `$HOME` and under Windows it will be `C:\Documents and Settings\username`.)

If you would prefer these cached files to be kept elsewhere you can set this in the configuration file, for example:

```
<server port="9092" cacheLocation="/usr/local/sgs/cache"></server>
```

The `cacheLocation` must be a directory. If it does not already exist it will be created. You must make sure that the user that is running the SGS server has write permissions in this directory.

Enabling logging messages

(For developers only, really.) The logging behaviour of the system is controlled by the `log4j.properties` file in the `conf` directory of the distribution. To see debug messages for a particular class, change the logging level of that class to `DEBUG`. You can see the individual Styx messages that are exchanged between client and server by setting the logging levels of the `StyxServerProtocolHandler` and `StyxConnection` to `DEBUG`. This is not recommended for normal use as it will significantly slow the system down.

2.2.3 More complex services

Styx Grid Services Tutorial: Creating more complex services

We shall now create some Styx Grid Services from programs that are a little more complex than a Hello World program. The process for doing so is exactly the same:

1. Create a machine-readable description of the program (the configuration file).
2. Run the SGS server using this description.
3. Run the service using the SGS client software.

In this section of the tutorial we shall look at programs that read some input and produce some output.

1. A simple filter

A filter is simply a program that reads data from its standard input and writes to its standard output. Filters are very common in Unix and Linux systems.

We shall create a Styx Grid Service that wraps a filter program that reads lines of text from its standard input, reverses each line and prints the lines to the standard output. As with the HelloWorld program of the first part of this tutorial, this has been implemented in Java in the [Reverse](#) class.

In order to deploy this as a Styx Grid Service, we must create an XML description of this program. This is included in the configuration file that is provided with the JStyx distribution (the `SGSconfig.xml` file in the `conf/` directory) but the relevant portion is reproduced here:

```
<gridservice name="reverse"
  command="JStyxRun uk.ac.rdg.resc.jstyx.gridservice.tutorial.Reverse"
  description="Reads lines of input and outputs them with characters reversed">
  <inputs>
    <input type="stream" name="stdin"/>
  </inputs>
  <outputs>
    <output type="stream" name="stdout"/>
    <output type="stream" name="stderr"/>
  </outputs>
</gridservice>
```

This specifies that the SGS called "reverse" will read data from its standard input and write data to its standard output and error streams. Run the SGS server by entering:

```
GridServices
```

as before. Assuming that the server has started successfully, you can run the service (under Unix or Cygwin) by entering:

```
cat somefile.txt | SGSRun localhost 9092 reverse
```

(You may, of course, have to change the hostname and port of the server). The pipe operator `|` redirects the standard output from the `cat` program to the standard input of the `SGSRun` program. The `SGSRun` program streams this information to the SGS server, which passes it to the `reverse` program.

You can also run this from Windows from a command prompt:

```
type somefile.txt | SGSRun localhost 9092 reverse
```

As with the [HelloWorld](#) example, you can create a shell script or batch file called "reverse" that runs `SGSRun` with the correct hostname and port. This script can then be used in exactly the same manner as the `Reverse` program itself if it were installed locally.

2. Input and output files

We shall now create an SGS that behaves much like the `reverse` service from the above section, but works in a slightly different way. Instead of reading data from the standard input and writing to the standard output, our new SGS will read data from an input file and output to a different file. As before, it will read each line of text from the file, reverse it and write the reversed lines to the output file.

The entry in the XML configuration file is slightly more complicated:

```
<gridservice name="reverse2"
  command="JStyxRun uk.ac.rdg.resc.jstyx.gridservice.tutorial.Reverse"
  description="Reads lines of input and outputs them with characters reversed">
  <params>
    <param name="inputfile" paramType="flaggedOption"
      flag="i" required="yes" description="Name of input file"/>
    <param name="outputfile" paramType="flaggedOption"
      flag="o" required="yes" description="Name of output file"/>
  </params>
  <inputs>
    <input type="fileFromParam" name="inputfile"/>
  </inputs>
  <outputs>
    <output type="fileFromParam" name="outputfile"/>
  </outputs>
</gridservice>
```

(Note that we are using the same Java program: the `Reverse` class. If you look at the [code](#) for this class you'll see how it works.) Let's work through this section of the configuration file:

- The `<params>` section defines the two command-line parameters that are understood by the `Reverse` program. They are both "flaggedOptions", which means that they are specified through the use of command-line flags. The name of the input file will be given by the item following the `-i` flag

and the output file name will be given by the item following the `-o` flag.

- The `<inputs>` section specifies that the program will take a single input file. The `type="fileFromParam"` attribute specifies that the name of the input file is given by the value of the parameter called "inputfile", i.e. the value after the `-i` flag.
- The `<outputs>` section specifies that the program will produce a single output file. The `type="fileFromParam"` attribute specifies that the name of the output file is given by the value of the parameter called "outputfile", i.e. the value after the `-o` flag.

(See [this page](#) for a more complete description of the config file specification.) Having started the SGS server, you can run the `reverse2` service like this:

```
SGSRun localhost 9092 reverse2 -i somefile.txt -o output.txt
```

The `SGSRun` automatically uploads the input file (`somefile.txt`) to the server and downloads the output file (`output.txt`).

3. Fixed-name input and output files

Sometimes you might want to create an SGS from a program that expects fixed names for its input and output files. For example, the program may always read input from a file called `input.txt` and write output to `output.txt`. In this case you will not use command-line parameters to set the names of the input and output files.

To achieve this, in the config file we use the type "file" (instead of "fileFromParam" or "stream") to specify the name of the files. The following piece of XML configures an SGS called "replace", which reads lines of input from an input file called `input.txt`, replaces all instances of one string with another and writes the result to `output.txt`:

```
<gridservice name="replace"
  command="JStyxRun uk.ac.rdg.resc.jstyx.gridservice.tutorial.Replace"
  description="Replaces all instances of one string in a file with another">
  <params>
    <param name="stringToFind" paramType="unflaggedOption"
      required="yes" description="String to find"/>
    <param name="stringToReplace" paramType="unflaggedOption"
      required="yes" description="String to replace"/>
    <param name="verbose" paramType="switch" flag="v"
      longFlag="verbose" description="If set true, will print verbose output to
stdout"/>
  </params>
  <inputs>
    <input type="file" name="input.txt"/>
  </inputs>
  <outputs>
    <output type="file" name="output.txt"/>
    <output type="stream" name="stdout"/>
    <output type="stream" name="stderr"/>
  </outputs>
</gridservice>
```

Note the use of `unflaggedOptions` to specify the strings to find and replace. These are command-line

arguments that do not use a flag to signal their presence. Provided that you have a file called `input.txt` in your current directory, you can run the `replace` SGS as follows:

```
SGSRun localhost 9092 replace hello goodbye
```

This will replace all instances of the word "hello" with "goodbye" in the file `input.txt`, writing the results to `output.txt`. As you may have gathered from the above XML, you can use the command-line flag `-v` (or `--verbose`) to produce more verbose output.

Even though the names of the files are fixed, you can still pass references to input files and get references to output files. Type

```
SGSRun localhost 9092 replace --sgs-verbose-help
```

and you'll see the arguments that you have to set. For example:

```
SGSRun localhost 9092 replace hello goodbye
--sgs-ref-input.txt=readfrom:http://www.google.com --sgs-ref-output.txt
```

will read input data from `http://www.google.com` and write a reference to the output data into the file `output.txt`. (The "readfrom" part is actually unnecessary in this case by the way.) Note that you can also use this technique to stream data from a remote source into the standard input of an SGS (`--sgs-ref-stdin=readfrom:URL`).

4. Interactive programs

If you have a program that expect user interaction through the command line (i.e. the user enters data at the keyboard), you can expose this as a Styx Grid Service. In fact, you have already done so: the `reverse` service from section 1 above reads data from its standard input. Try running:

```
SGSRun localhost 9092 reverse
```

without piping any data to its standard input. The program will just sit and wait for you to type at the keyboard. Every line you type will be reversed by the `reverse` SGS and sent back to you, printed on the standard output (console window). This will continue until you enter an end-of-file command (Control-Z in Windows and Control-D in many other systems).

You could expose any interactive program in this way, including the Python interactive shell and the bash shell! Of course, there may be serious security implications connected with doing this if the program you are exposing as an SGS allows the user to enter data that can cause damage to your system. (This is true with any Styx Grid Service, of course.)

2.2.4 Getting help

Styx Grid Services Tutorial: Getting help

You can get help and usage information for any Styx Grid Service from the SGSRun program. Just typing SGSRun on its own will reveal a usage message that shows that SGSRun expects at least three arguments: the host name and port number of the remote server and the name of the Styx Grid Service itself.

Brief usage information

If you want to see the usage information (i.e. the expected command-line arguments) for a particular SGS, use the `--sgs-help` command-line switch. For example, to get usage information for the `reverse2` SGS that we created earlier in the tutorial, enter:

```
SGSRun localhost 9092 reverse2 --sgs-help
```

This will print out (to standard output) the description of the SGS (taken directly from the server's configuration file) and a brief usage message describing the command-line arguments. The above example will print out the following information:

```
Reads lines of input and outputs them with characters reversed

Usage: reverse2 -i <inputfile> -o <outputfile> [--sgs-help] [--sgs-verbose-help]
      [--sgs-debug] [--sgs-allrefs] [--sgs-lifetime <sgs-lifetime>]
```

Mandatory arguments are given in angle brackets (< >) and optional arguments are given in square brackets ([]). Note that, in addition to the the arguments that are expected by the `reverse2` program, there are a number of optional arguments whose names begin with `--sgs-`. These arguments are available for all Styx Grid Services.

Verbose usage information

More details on the usage of a particular SGS can be found with the `--sgs-verbose-help` switch. For example, running `SGSRun localhost 9092 reverse2 --sgs-verbose-help` will print out the following to the console window:

```
Reads lines of input and outputs them with characters reversed

Usage: reverse2 -i <inputfile> -o <outputfile> [--sgs-help] [--sgs-verbose-help]
      [--sgs-debug] [--sgs-allrefs] [--sgs-lifetime <sgs-lifetime>]

-i <inputfile>
```

```
    Name of input file

-o <outputfile>
    Name of output file

[--sgs-help]
    Set this switch to print out a short help message

[--sgs-verbose-help]
    Set this switch to print out a long help message

[--sgs-debug]
    Set this switch in order to enable printing of debug messages

[--sgs-allrefs]
    Set this switch in order to get URLs to all output files rather than
    actual files

[--sgs-lifetime <sgs-lifetime>]
    The lifetime of the SGS in minutes (default: 60)
```

As you can see, this gives a brief description of all the possible input parameters (these descriptions are taken from the server's configuration file). The meanings of the `--sgs-` parameters will be described later in this tutorial.

Troubleshooting

If a Styx Grid Service is not behaving as you think it should, try running it with the `--sgs-debug` switch. This will print out some debugging information that might lead you to the source of the problem. Even if you can't fix it yourself, this will be useful information that you can use to ask for help from the [mailing lists](#). In general, these mailing lists are the best place to go for help as your mail will reach many users and developers of the SGS software.

2.2.5 Workflow and scripting

Styx Grid Services Tutorial: Workflow and scripting

The earlier sections of this tutorial have shown how remote Styx Grid Services can be executed exactly as if they were local programs. This means that we can link SGSs together to form a distributed application (or "workflow") just as easily as we can link local programs together to achieve a goal. Styx Grid Services, like local programs, can be linked together with simple *shell scripts* (or batch files under Windows). [This paper](#) describes how Styx Grid Services can be used in this way.

A simple workflow

Let us create a very simple distributed application (or workflow) from two of the Styx Grid Services that we have already met: `HelloWorld` and `Reverse`. We are going to use the `HelloWorld` SGS to output the string "Hello World" and the `Reverse` SGS to reverse that string.

We can achieve this by piping the output from the `HelloWorld` SGS to the input of the `Reverse` SGS, just as if they were local programs:

```
SGSRun localhost 9092 helloworld | SGSRun localhost 9092 reverse
```

The output from this simple workflow should be "dlrow olleH". If we were to create wrapper scripts called `helloworld` and `reverse` (as discussed earlier in this tutorial) we could simply write:

```
helloworld | reverse
```

In the above example, both SGSs were running on the same server. If you are able, try running the SGS server on two different machines and performing the same workflow again, for example:

```
SGSRun machine1 9092 helloworld | SGSRun machine2 9092 reverse
```

Using input and output files

The above example demonstrated the use of the pipe operator to send the data between the two SGSs. You could of course send the data to an intermediate file and use the `reverse2` SGS, which reads input from a file rather than from its standard input:

```
SGSRun localhost 9092 helloworld > temp.txt  
SGSRun localhost 9092 reverse2 -i temp.txt -o reversed.txt
```

The file `reversed.txt` should now contain the string `"dlrow olleH"`.

Reading files from other servers

One of the strengths of the SGS system lies in the fact that you can pass input files by reference. In other words, instead of specifying an actual input file, you can specify a URL to a file on a different server.

For example, let's run the `reverse2` Styx Grid Service, using input data from the Web:

```
SGSRun localhost 9092 reverse2 -i readfrom:http://www.google.com -o output.txt
```

When this finishes, open `output.txt` and verify that it contains the contents of the Google home page (in HTML), but each line of text has its characters reversed.

IMPORTANT: You must use the syntax `"-i readfrom:URL"` rather than just `"-i URL"`. There is a good reason for this, which we won't go into now.

Let's have a quick look in more detail at what has happened in this example:

1. The `SGSRun` program connects to the server and creates a new instance of the `reverse2` service.
2. The URL was sent to the SGS server.
3. *The server* downloaded the data from that URL into a temporary file.
4. The server passed that file into the `reverse2` program.

It's important to note that the server must be able to "see" the data at the URL you specify. If the server cannot make a connection to that URL an error will be raised.

Streaming data between Styx Grid Services

Let's create a silly workflow of two Styx Grid Services. We're going to reverse the contents of a file, then do the same again so that the contents of the final result are identical to the original file:

```
SGSRun localhost 9092 reverse2 -i input.txt -o output1.txt
SGSRun localhost 9092 reverse2 -i output1.txt -o output2.txt
```

If you run this with some input file (or you could pass in data from a URL as above) you should be able to verify that `input.txt` and `output2.txt` have the same contents.

Let's pretend that we were working with large files and that we weren't interested in the intermediate file (`output1.txt`). We have wasted time and bandwidth by downloading `output1.txt` to our local machine and then immediately uploading it to the second service in the above workflow.

We can be more efficient by downloading (and then uploading) a *reference* to the intermediate file, with a small change to the workflow. We just add a `.sgsref` extension to any output file that we want to get a reference to. Then we can upload that reference exactly as if it were the file itself:

```
SGSRun localhost 9092 reverse2 -i input.txt -o output1.txt.sgsref
SGSRun localhost 9092 reverse2 -i output1.txt.sgsref -o output2.txt
```

You should be able to verify that this has the same overall effect as the previous workflow. If you examine the contents of the `output1.txt.sgsref` file you will find that it contains the string `"readfrom:styx://.../reverse2/instances/.../outputs/outputfile"`. This is a reference to the output file that was produced by the first SGS.

Streaming data using the pipe operator

Let's go back to the first example in this section of the tutorial. We printed the string "Hello World" then reversed it using two SGSs:

```
SGSRun localhost 9092 helloworld | SGSRun localhost 9092 reverse
```

What happened behind the scenes was this: the standard output from the `helloworld` service was redirected to the *local* console window. Instead of being printed out, it was redirected immediately to the remote `reverse` service. In other words the data made an unnecessary trip to our client machine and back out again.

As above, we can arrange for the data to be passed directly between the two services. However, this time we have no filename to which we can append the magic `".sgsref"` extension so what do we do? You can find out by using the help system: enter `SGSRun localhost 9092 helloworld --sgs-verbose-help` (see [Getting help](#)). There is a command-line switch `--sgs-ref-stdout`, which will cause a reference to the output data to be printed to the console window instead of the data themselves. It is this reference that is passed to the `reverse` service:

```
SGSRun localhost 9092 helloworld --sgs-ref-stdout | SGSRun localhost 9092 reverse
```

The string "Hello World" has been passed directly between the two services.

Obtaining the error code

You should now be getting the picture that you can create shell scripts (or batch files) that tie Styx Grid Services together to produce distributed applications. The `SGSRun` program behaves exactly like the program that has been wrapped as a Styx Grid Service. It even captures the error code from the remotely-running executable and returns this error code when it finishes. Therefore, you can trap this error code to see if the remote executable has finished successfully.

Disadvantages of the SGS system

The SGS system is a very quick and easy way to create workflows that are based on remote services. We have seen how data can be passed directly between services. However, unlike other workflow systems (e.g. Web Service-based ones), the units of information that are being passed around are *files*. In other systems, these units might be strings, integers or perhaps objects. This means that it is up to the individual services in an SGS workflow to verify that its input files are valid (the inputs and outputs are very weakly typed). Exactly the same problem is of course faced when using shell scripts to tie together local programs.

2.2.6 Advanced stuff

Styx Grid Services Tutorial: Advanced techniques

We have been through the process of setting up an SGS server and running some services using the general-purpose SGSRun client program. This is all that many people will need to know. However, there are some more techniques that you should know in order to get the most out of the system.

Monitoring progress and status

Clients of Styx Grid Services can monitor the changes in the state of the remote service. (This state data is known as *service data*.) All Styx Grid Services expose at least two pieces of service data: the first piece gives the status of the service:

(See the [StatusCode](#) class.) The second piece of service data gives the error code that is returned by the program that underlies the service instance. Service providers can also create custom pieces of service data: see [Configuration of Styx Grid Services](#) .

Monitoring of service data is handled in a rudimentary way in the SGSRun program. By default, no service data is monitored, but if you run SGSRun with the switch `--sgs-debug`, all updates to service data will be printed to the standard output stream. This can be useful when troubleshooting as you can get information about what has happened on the server. For example:

```
SGSRun localhost 9092 helloworld --sgs-debug
```

produces the following output (for example):

```
status = created
Hello world
status = finished: took 0.344 seconds.
exitCode = 0
```

Understanding the config file

The hardest part of setting up a Styx Grid Services server is creating the configuration file. Although an example config file is provided with the distribution, this cannot cover all possible permutations of configurations. Please read [this page](#) for a more complete description of how to write the configuration file.

Setting the lifetime of a Styx Grid Service

Whenever you run a Styx Grid Service, a new instance of that service is created on the server. This instance contains cached copies of all the input and output files that the underlying executable program reads and writes. In most cases, once the SGS has finished running, these cached files are no longer needed.

The `--sgs-lifetime` command-line option can be used to set the lifetime of a particular SGS instance (run `SGSRun` with the switch `--sgs-verbose-help` to see this). By default this is set to 60 minutes; after this time the SGS instance will be automatically garbage-collected on the server. You can set this to a longer or shorter time if you wish: for example, if you know that the SGS that you are running will take several hours to run, you will need to set `--sgs-lifetime` to a much larger number, or the service will be destroyed before it has finished.

In future releases, service lifetime will be handled better: it will be up to service providers, not clients, to decide on the best default lifetime for each service. Also, it will not be possible for services to be garbage-collected before they have finished running.

Creating custom clients

The `SGSRun` program is a general-purpose command-line client for any Styx Grid Service. It is possible to create other client programs using the JStyx API. A full discussion of this is beyond the scope of this tutorial, but in the meantime you might be able to figure out what to do by looking at the code for the `SGSRun` and `SGSInstanceClient` classes. Basically, the idea is that you create a class that implements the `SGSInstanceChangeListener.html` interface, and register this as a listener with an instance of `SGSInstanceClient`. You can then use the methods of `SGSInstanceClient` to start the service and subscribe to changes in service data and download output files, etc.

2.2.7 Security

Styx Grid Services Tutorial: Security

In the current public release (0.2.0), security is not implemented properly. There is no authentication of users nor encryption of network traffic. This is a high priority for addressing in future releases. Please email the [jstyx-users mailing list](#) if you would like this to be enabled urgently: this will encourage the developers and we might be able to send you an early version of a future release or direct you to the source code repository.

A security mechanism based upon secure authentication of users, Unix-style fine-grained access control to individual Styx files and SSL-based encryption of network traffic is under development.

2.3 The configuration file

Configuration of Styx Grid Services

In order to run a Styx Grid Services server, you will need to create a configuration file in XML. This section describes the format of this XML file and gives some examples of how to set up Styx Grid Services.

Overall structure

The overall structure of the XML file is quite simple. If you are not familiar with XML, don't worry. XML files are just text files with a defined structure. Important bits of information are placed between tags like so: `<name>Joe Bloggs</name>`. If this reminds you of HTML, there's a good reason for this. Modern, well-structured HTML (known as XHTML) is actually a "flavour" of XML.

The configuration file is described by a Document Type Definition (DTD). The DTD specification for the SGS configuration file is found in `conf/SGSconfig.dtd`. You don't need to worry about this: it is used internally by the SGS software to make sure that the configuration file is valid. If you create an invalid configuration file, this will be detected when you try to run the SGS server and an error message will appear.

The large-scale structure of the configuration file looks like this:

```
<sgs>

  <server address="sgs.myserver.com" port="9092"
cacheLocation="C:\StyxGridServices">
  ...
</server>

  <gridservices>
    <gridservice name="mysgs" ... ></gridservice>
    <gridservice name="anotherSGS" ...></gridservice>
    ...
  </gridservices>

</sgs>
```

Everything is contained between `<sgs>` and `</sgs>` tags. The information between the `<server>` tags specifies the server settings. The `<server>` tag itself has three possible attributes:

Note that the `<server>` tag can be omitted from the config file altogether. In this case, default values

will be chosen for all attributes and the server will be unsecured.

The contents of the `<gridservices>` tag are explained in the following sections.

Configuration of a Styx Grid Service

The `<gridservices>` tag is a container for all the `<gridservice>` tags. There is one `<gridservice>` tag for each Styx Grid Service that the server exposes. This tag contains all the information about the executable that the SGS is wrapping: the path to the executable, the command-line parameters that it expects, the input files it consumes, the output files it creates, plus some other things. The structure of the `<gridservice>` tag and its sub-tags looks like this:

```
<gridservice name="mysgs" command="C:\path\to\executable"
  description="A Styx Grid Service">
  <params>
    ...
  </params>
  <inputs>
    ...
  </inputs>
  <outputs>
    ...
  </outputs>
  <serviceData>
    ...
  </serviceData>
  <steering>
    ...
  </steering>
  <docs>
    ...
  </docs>
</gridservice>
```

The `<gridservice>` tag itself has three attributes. The name attribute gives a short name for the SGS that will be used to identify it. This name must be different from the names of all the other SGSs on this server. This name cannot contain spaces. The `command` attribute specifies the full path to the executable that will be run. A short, one-sentence description of the SGS can be placed in the optional `description` tag.

The sub-tags (children) of the `<gridservice>` tag specify different aspects of the Styx Grid Service. Most SGSs will only require a few of these tags to be used, as we shall see. We shall now go through each of these tags in turn and describe how to use them.

Parameters

Parameters are values that are set before an SGS is run. In the current system the parameters translate directly into the command-line arguments for the underlying executable. The parameters are specified between the `<params>` tags. This is perhaps the most complicated part of the SGS configuration but hopefully you'll see that it's not too difficult. The `<params>` tag is a container for zero or more

<param> tags. There is one <param> tag for each command-line argument that the executable expects.

Each <param> tag must contain a set of attributes:

(The Java Simple Argument Parser, [JSAP](#), is used to handle command line arguments in both the SGS server and client code. Therefore, the nomenclature used here reflects that used in JSAP.) Most of the attributes are explained adequately (I hope) in the above table. However, some attributes require further explanation:

There are three parameter types that the SGS system understands. They are named after the differing means of specifying their values on a command line through the use of arguments:

- A **switch** is an parameter that can either be true or false. On the command line (i.e. when running the executable outside the SGS framework) switches are arguments (*flags*) such as "-v" and "--verbose" whose presence alone is significant. They do not have an associated value.
- A **flaggedOption** is like a switch but it has an associated value. On the command line this value is given by the token after the flag: e.g. "-n 5" or "--number=5" are two different ways of setting the value of a certain parameter to 5.
- An **unflaggedOption** is a parameter whose value is given purely by the position of its associated argument on the command line, relative to other unflaggedOptions. The final unflaggedOption can be marked as *greedy*, meaning that it will consume the remainder of the command line.

It probably helps to look at some examples here. Let's say that we are wrapping an executable that reads a single input file and writes a single output file. The name of the input file is signified on the command line by the short flag "-i" or the long flag "--inputfile". The name of the output file is signified by the short flag "-o" or the long flag "--outputfile". Both of these arguments are compulsory. The <params> tag in the configuration file would look like this:

```
<params>
  <param name="inputfile" paramType="flaggedOption" required="yes"
    flag="i" longFlag="inputfile"/>
  <param name="outputfile" paramType="flaggedOption" required="yes"
    flag="o" longFlag="outputfile"/>
</params>
```

The usage of this executable is `myprog -i <inputfile> -o <outputfile>`.

Now let's look at an example in which we are wrapping an executable that reads a number of input files and writes a single output file. In this case, there are no command-line flags to help us: the first argument on the command line gives the name of the output file and the remaining arguments are the names of all the input files that must be read. The <params> tag in the configuration file would look like this:

```
<params>
  <param name="outputfile" paramType="unflaggedOption" required="yes"/>
  <param name="inputfiles" paramType="unflaggedOption" required="yes"
  greedy="yes"/>
</params>
```



This time both parameters are `unflaggedOptions` (parameters whose value is found by looking at a certain position on the command line). The first argument gives the name of the output file and the remaining arguments are consumed by the `inputfiles` parameter, which is set to be greedy.

As a final example, let's pretend that we are wrapping an executable (called `replace`) that searches for all instances of a certain string in a file and replaces those instances with another string. In addition, the user can tell the program to print verbose debug information by specifying the argument `"-v"`. Here is an example of running this executable from the command line:

```
replace -i input.dat -o output.dat Hello Goodbye -v
```

This would replace all instances of "Hello" in the file `input.dat` with the string "Goodbye" and write the result to `output.dat`, whilst printing verbose debug messages. The `<params>` tag in the configuration file would look like this:

```
<params>
  <param name="verbose" paramType="switch" flag="v"/>
  <param name="inputfile" paramType="flaggedOption" required="yes" flag="i"/>
  <param name="outputfile" paramType="flaggedOption" required="yes" flag="o"/>
  <param name="stringToFind" paramType="unflaggedOption" required="yes"/>
  <param name="stringToReplace" paramType="unflaggedOption" required="yes"/>
</params>
```

Not that only the order of the `unflaggedOptions` is important. Switches and `flaggedOptions` can be placed anywhere on the command line and can be specified anywhere between the `<params>` tags.

Inputs

Having specified the parameters that the executable expects, you'll be glad to know that we've done most of the hard work. The next thing we specify in the configuration file is the set of inputs from which the executable will read. An executable (and therefore a Styx Grid Service) can read input data either from its standard input stream or from files. In the case of files, the names of these files are either fixed or they can be set using a parameter (see above). The inputs are specified between the `<inputs>` tags in the configuration file.

The `<inputs>` tag is a container for zero or more `<input>` tags, with one `<input>` tag for each file or stream that provides input data. Each `<input>` tag contains exactly two attributes:

All file names are specified relative to the working directory of the executable. This may seem a little confusing, and indeed the design here is probably not optimal. However, hopefully some examples will

clear things up. We'll look at some examples when we've dealt with the `<outputs>` section of the configuration file.

Outputs

Output files and streams are specified in a very similar way to input files. An executable can output data as files or on one of its standard streams (stdout and stderr). In the case of output files, the names of these files can be fixed or specified by the value of a parameter.

The `<outputs>` tag is a container for zero or more `<output>` tags, with one `<output>` tag for each file or stream that contains output data. Each `<output>` tag contains exactly two attributes:

All file names are specified relative to the working directory of the executable.

Service Data

Service data is information about the state of a particular Styx Grid Service instance. For example, the status of a service is represented by a service data element (SDE), which can contain values such as "created", "running" and "finished". The "status" SDE is built in to the system and the user does not need to specify it in the configuration file. It is possible for users to create their own service data elements but this is considered an "advanced" topic and will not be described here (yet).

Steerable parameters

With some programs (e.g. fluid dynamics simulations) it is possible to adjust the values of some parameters while the program is running. The `<steering>` section of the configuration file allows this to be set up, but again this is an "advanced" (and rarely-used) topic and will not be described here at the moment.

Documentation

The Styx Grid Service framework allows service providers to provide access to free-form documentation about each service. This is specified between the `<docs>` tags in the configuration file. The `<docs>` tag is a container for zero or more `<doc>` tags. Each `<doc>` tag is a file or directory that contains documentation: if it represents a directory then all the files under that directory will be exposed for reading by clients.

The specification of the `<doc>` tag is very simple:

For example, let's say that we want to expose two documentation elements. The first is a directory of documentation files (say, a set of Word documents that describe the operation of the executable). The second is a simple one-paragraph description of the executable that is called "description.txt" in real life, but we want to expose it with the name "README". The documentation part of the configuration file would look like this:



```
<docs>
  <doc location="c:\myprog\docs\">
  <doc location="c:\myprog\description.txt" name="README">
</docs>
```

Examples

OK, we've gone through the nuts and bolts of the Styx Grid Service configuration file in some detail. Let's put it all together with a couple of examples. The sections you will have to worry about most are the parameters and the input and output files. Other sections are used a lot less, so it is those three sections which we shall focus on here.

Example 1: a simple Unix filter

As a first example, let's look at how we expose a very simple program as a Styx Grid Service. We'll take the example of the `md5sum` program, a program found on most Unix-like systems. The `md5sum` program reads data from its standard input and calculates a "digest" of the data in the form of a large number which is printed out (usually as a hexadecimal string) to its standard output. (The MD5 digest is usually used as a "checksum": the MD5 digest of a file is a large number that is highly unlikely to have been produced by any other file.). Programs that behave in this way (i.e. that read data from standard input and write data to standard output) are sometimes known as *filters*.

The entire configuration file that is required to expose the `md5sum` program as a Styx Grid Service is as follows (the first two lines just declare that this is an XML file and that it conforms to the specification given in `SGSconfig.dtd`):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sgs SYSTEM "SGSconfig.dtd">

<sgs>

  <gridservices>

    <gridservice name="md5sum" command="/usr/bin/md5sum"
      description="Calculates the MD5 checksum of data that are read from
standard input">

      <inputs>
        <input type="stream" name="stdin"/>
      </inputs>

      <outputs>
        <output type="stream" name="stdout"/>
        <output type="stream" name="stderr"/>
      </outputs>

    </gridservice>

  </gridservices>

</sgs>
```

Working down this file: The `<server>` tag is omitted and so [default values](#) are chosen for the server settings. We specify a single Styx Grid Service called `md5sum` and specify the full path to the executable that we are wrapping. The SGS takes no parameters, but reads data from its standard input and writes data to its standard output and standard error streams.

Example 2: replace

In the "Parameters" section [above](#) we specified the parameters taken by an executable that reads an input file, replaces all instances of one string with another, then writes the resulting output file. We've actually already done the hardest bit of creating the configuration file in this case: all we need to do now is to specify the input and output file in the configuration document. The information below must be placed within the `<gridservices>` tag in a complete configuration file such as that given in example 1 above:

```
<gridservice name="replace" command="C:\path\to\replace.exe"
  description="Finds and replaces a string in a file">

  <params>
    <param name="verbose" paramType="switch" flag="v"/>
    <param name="inputfile" paramType="flaggedOption" required="yes" flag="i"/>
    <param name="outputfile" paramType="flaggedOption" required="yes" flag="o"/>
    <param name="stringToFind" paramType="unflaggedOption" required="yes"/>
    <param name="stringToReplace" paramType="unflaggedOption" required="yes"/>
  </params>

  <inputs>
    <input type="fileFromParam" name="inputfile"/>
  </inputs>

  <outputs>
    <output type="fileFromParam" name="outputfile"/>
  </outputs>

</gridservice>
```

We've already described the parameters [above](#). All we have done here is to state that the executable expects one input file, whose name will be given by the value of the parameter called "inputfile". Furthermore, we state that the executable writes a single output file, whose name is given by the value of the parameter called "outputfile".

2.4 The SGS namespace

The SGS server namespace

All resources on any kind of Styx server are represented as a set of files in a hierarchical structure (see the [JStyx tutorial](#)). This file hierarchy is known as a *namespace*. Styx Grid Services are no different. When an executable is exposed as a Styx Grid Service, a namespace is created. Clients interact with the SGS by reading from and writing to files in this namespace.

This page describes the namespace of a general Styx Grid Service. This information is intended for developers who want to know what is going on behind the scenes in the SGS system, perhaps in order to develop custom clients or add new features. Knowledge of the SGS namespace is not at all necessary for most users.

Namespace description

The diagram below shows the namespace of a general Styx Grid Services server. The hyperlinks will take you to a detailed explanation of the purpose of each file and directory. Note that the structure of the namespace mirrors closely the structure of the XML configuration file, which is described [here](#).



Styx Grid Service files

The files in this section are "global" to a particular Styx Grid Service: they do not belong to a specific instance of the Service.

The clone file

The clone file is used to create a new SGS instance. When a client reads from this file, a new SGS instance is created and the URL to the root of this instance is returned to the client. Note that the instance might be created on a different server for load-balancing reasons, which is why a full URL is returned (however, in the current release, instances are always created on the same server). Writes to this file are not allowed.

The config file

When a client reads from this file, an XML string will be returned, representing the configuration of the Styx Grid Service in question. This XML string is very close to the [XML configuration](#) of the Styx Grid Service, except that the <docs> tag does not appear, nor does the "command" attribute of the root <gridservice> tag. Neither of these things are relevant to the client.

The docs directory

This directory contains all the documentation files that the server has chosen to expose for the benefit of the client. These files can be in any format and they are here to provide more information to the client about the use of the SGS in question.

The `description` file in this directory is present in all Styx Grid Services. It is a short description of the purpose of the SGS, and is gleaned from the "description" attribute of the [XML configuration file](#). The `readme.txt` file shown in the above diagram is simply an example of a documentation file and is not present in all SGSs. Note that the `docs/` directory can contain any number of files and subdirectories.

The instances directory

The `instances/` directory contains a set of directories, one for each instance of this particular Styx Grid Service. The name of the directory is the unique ID of the instance. The diagram above shows two instances for the `mySGS` service, with IDs 0 and 1.

Styx Grid Service instance files

The files in this section belong to a particular instance of a Styx Grid Service.

The ctl file

This is a very important file: it is used to control the Styx Grid Service ("ctl" is short for "control"). When all [parameters](#) have been set and all [input files](#) have been uploaded, the client starts the service running by

writing the string "start" into this file. The service can be stopped at any time by writing the string "stop" into this file. The instance can be destroyed and its resources freed by writing the string "destroy" into this file.

The arguments file

The `args` file is used to read and write the full set of command-line arguments that will be passed to the underlying executable. Reading this file is very useful for debugging purposes.

The parameters directory

The `params/` directory contains a set of files, one for each parameter that can be set by clients. There is one parameter file for each parameter in the [configuration file](#), except for parameters that represent output files. Clients set parameters by writing values into these files. The parameters are translated into command-line arguments to be passed to the underlying executable. With each write, the server will check the validity of the input, returning an error if the parameter value was not valid. These files can be read to get the parameter values.

The inputs directory

The `inputs/` directory contains files to which clients can write input data. If the "stdin" file is present then this means that the underlying executable is expecting data to be passed to its standard input. Clients can write data to this file while the service is running in order to stream data to the standard input.

The other files in this directory represent input files that will be read by the underlying executable. These files **must** be uploaded before the service is started. Some input files have fixed names: these are always present in the namespace. Other input files are specified by the value of a certain parameter (see the [configuration instructions](#)). In these cases, the corresponding file will not appear in the namespace until the value of that parameter is set. Note that several filenames can be written to the parameter (separated by spaces), in which case a file will appear in the namespace for each filename that is written.

The outputs directory

The `outputs/` directory contains a file for each output file or stream that is produced by the underlying executable. The files called "stdout" and "stderr" are read to obtain data from the standard output and standard error of the executable. Other files represent output files that are produced by the executable.

The service data directory

The `serviceData/` directory contains files, one for each element of service data that the service exposes. "Service data" is data about the state of the service instance.

In the above diagram there are three elements of service data shown. The first two service data elements (SDEs) are present in all Styx Grid Services. The "status" SDE can be read to find out the status ("created", "running", "finished", "aborted" or "error") of an SGS instance. If a client reads from the "exitCode" file, no data will be returned (i.e. the read will block) until the underlying executable has finished running, at which point the exit code from the executable will be returned. The "customSDE"

element is a user-specified service data element (see [here](#) for details of how this is done). User-specified SDEs might be used, for example, to provide a way to monitor the progress of an executable in a finer-grained way than is possible with the "status" SDE.

Files that represent service data elements exhibit blocking behaviour. When a client reads from the file from the first time, the data are returned immediately (except in the case of the `exitCode` file as described above). If the same client reads from the file again *without closing the file*, the read request will block until the underlying data change, at which point the data are returned to the client. This is how clients can receive asynchronous messages without the need for callbacks.

The steering directory

In some cases, programs permit some parameters to be adjusted while the executable is running. This is known as computational steering. The `steering/` directory contains the files that are used to do this, but this is considered an advanced feature and will not be discussed here (yet).

The time directory

The `time/` directory contains files that are pertinent to the lifecycle of the SGS instance. The `currentTime/` file can be read to give the current time according to the SGS server. The `creationTime` file can be read to give the time at which the instance was created. The `terminationTime` file contains the time at which the service instance will be automatically terminated. When the instance is first created it is set to never terminate (the `terminationTime` file will be empty). Clients can set a lifetime for the instance by writing a termination time into this file. In order to terminate the instance immediately, clients should write the string "destroy" into the `ctl` file (it is not legal to write a time in the past into the `terminationTime` file).

All times are read and written in the `xsd:dateTime` format, for example "2006-01-23T17:34:56+00:00".

3.1 **JavaDocs**

3.2 **Source code**

3.3 **Sf.net site**

<http://sourceforge.net/projects/jstyx/>