JStyx v.0.1.0-SNAPSHOT

Project Documentation

Reading e-Science Centre

06 September 2005

Table of Contents

1 JStyx

2 Styx Grid Services

3 JStyx References

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 the JStyx libraries

Currently the only way to download the JStyx libraries is as a platform-independent installer package in JAR format. The latest version is 0.1.0-SNAPSHOT and you can download it from here .

Installing the software

If you have downloaded the JAR installer (currently the only option) you can start the installer by running:

```
java -jar jstyx-0.1.0-SNAPSHOT-installer.jar
```

This has been tested on Windows and Unix. You will need a Java runtime environment version 1.4.2 or above (http://java.sun.com) in order to run the installer (and, indeed, to run the software).

Uninstalling the software

If you installed using the JAR installer, you can uninstall the software by changing to the "Uninstall" directory of the distribution and running

```
java -jar uninstaller.jar
```

If you are running Windows, there is probably a shortcut to this uninstaller in the JStyx program group in the Start Menu.

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?

Specific points

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
- Firewall-friendly
- Data streaming
- Built on robust framework (MINA)
- Platform independence
- Secure (in future!)
- Comparison with Web Services...

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.

Specific points

Specific points

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.

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 hierachy 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 BufferedReaders and BufferedWriters 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 `setContents()`:

```
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 Tutorial

What are Styx Grid Services?

Please see this paper for an explanation of what SGSs are and how they are used. More details to follow.

Getting Started

Download the software

Install the software

My First Styx Grid Service

2.1.1 Tutorial 1

2.1.2 Tutorial 2

3.1 JavaDocs

3.2 Source code