Stream-Components: Component based Stream computation on the Grid

Paul N. Martinaitis 1 Andrew L. Wendelborn 1
1 School of Computer Science
University of Adelaide, South Australia 5005.
Email: paulm@andrew@cs.adelaide.edu.au

Abstract

This paper reports on an investigation into component-based design in object-based distributed systems. We focus on the modeling of stream processing in terms of components (as exemplified in the EU CoreGrid project ProActive), to show how a stream processing system can be built from objects, and components in composition. In particular, we explore mechanisms for dynamic reconfiguration and distributed management of streams in a grid context. In further work, we will examine the relationship of such design to grid workflows and web services.

1 Introduction

This report describes the initial work we have done in designing a set of Components, called Stream Components to facilitate stream-based computation in a Grid environment. Our implementation is based on Java, together with the distributed grid component system ProActive (see Section 2.1.4), a combination particularly well suited to a (potentially heterogeneous) grid environment.

1.1 Stream Model

This work considers the simple stream model shown in figure 1; the salient features of this structure are:

- the stream begins with a 'stream source' which supplies values (data) to the rest of the stream. These values can be generated via a variety of means including: algorithmically (e.g. a random number generator), from a local database, from a local or remote database.
- the rest of the stream consists of a series of transducers (F1, F2, ..., FN) whose purpose is to apply some function or transformation on the stream values as they pass through.
- finally, there is a consumer at the end of the Stream which will accept the processed stream values and use them in some way.

This work is funded from a DEST/DISER International Science Program (ISL) grant for "The Utility Grid Project: Autonomic and Utility-Oriented Global Grids for Powering Emerging e-Research Applications".

Copyright ©2009, Australian Computer Society, Inc. This paper appeared at the 7th Australasian Symposium on Grid Computing and e-Research (AUSGRID 2009), Wellington, New Zealand. Conference in Research and Practice in Information Technology, Vol. 99. P. Roe and W. Kelly, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

Figure 1: A Generic Stream

The evaluation model is very simple. Demand is propagated backwards through the stream by invocations of a `car()` method; the returned values of each invocation are the 'stream values' which flow back to the consumer.

An important characteristic of the stream is that the data-dependencies are explicitly represented by the stream flow; we can thus distribute the stream nodes onto the grid without affecting the semantics. However, to manually deploy a stream computation onto the grid on an individual application basis, involves much effort on the part of the programmer, most of which is unnecessary duplication. This is because typically, aside from minor variations in topology, streams will differ from each other only in the compute 1 code. We can thus identify a 'separation of concerns' between the functional aspects of the stream, that is the compute code unique to a particular application, and the non-functional aspects necessary to maintain stream semantics and distributability on the grid.

To achieve this separation of concerns and make the stream more amenable to deployment and management on the grid, we have developed the Stream Components which wrap the 'stream nodes' as shown in figure 2. The component notation is fully explained in section 2.1; for now, it is sufficient to note that the inter-component bindings are opposite to stream flow direction. There are two types of StreamComponent (SC) - Stream and StreamF which encapsulate a 'Stream Source' and 'Stream Transluter' respectively. The evaluation semantics of the two components are:

- Stream: Invocation of `car()` returns the next value from 'Stream source'.

    *Stream*: Invocation of `car()` returns the next value from 'Stream source'.

*Note:* In this paper, we term the 'stream source' and transducer functions as Compute code.
Figure 3: Example StreamComponent Application

- StreamF: Invocation of `car()` causes the StreamF to obtain the 'upstream value' by calling `car()` on the upstream component. The transducer function is then applied to the 'upstream value' and the result is returned.

To facilitate deployment and management on the grid, the StreamComponents provide additional non-functional properties:

- Location-transparency and distributability: that is to say, the SC can be migrated between hosts with the inter-host communications being automatically and transparently handled.

- A re-configuration mechanism, explained in section 3.3, which allows the compute code of a SC to be changed remotely. This could be used to swap to a better algorithm if the nature of the data changes, for example.

- Inter-component bindings can be altered. This allows SC to be added or deleted from the stream as needed.

- There is the potential to exploit pipelining-parallelism implicit in the stream structures by supporting more eager modes of evaluation.

This paper will describe the incremental development of the basic StreamComponents as a series of experiments.

1.2 Example Application

To appreciate how the various features of the StreamComponents may be used in reality, consider the example Image Processing application shown in figure 3. The ‘Provider’ supplies an image processing stream which is distributed across two hosts (H1 & H2); the system is accessible to the user via four interfaces, which are indicated by diamonds. We have a Stream component which supplies a stream of Images from a local database which are then passed through two StreamF components:

- **User Algorithm**: this is a component which allows the user, via the 'User Alg' interface, to upload their own code to perform some custom processing on the images.

- **Proprietary Algorithm**: the provider supplies three different proprietary algorithms which the user may choose between by using the 'Select Alg' interface.

Figure 4: A Fractal Component

The user can obtain the Stream values via two interfaces: ‘Post-Process’ and ‘Direct’. The Post-Process interface performs extra processing by passing the values through an additional StreamF component. This example demonstrates several useful aspects of StreamComponents. Firstly, the location-transparency allows the various SC to be distributed across a number of hosts with the communications between H1 and H2 handled automatically. The re-configurability of the components is used in two ways. The **Proprietary Algorithm** component enables the Provider to supply several different algorithms to the user without having to actually disclose the algorithm. Conversely, the **User Algorithm** component allows users to upload their own code for use in the stream whilst maintaining the integrity of the rest of the system. The interfaces to the system may be provided by direct object references or a mechanism such as Web Services; this latter option will be the subject of further investigation.

1.3 Outline for paper

We begin in section 2 by introducing the technologies on which StreamComponents are based, namely the Fractal Component Model and ProActive. Section 3 will then describe, as a series of experiments, the incremental development of the StreamComponents. We start by showing how the Stream model shown above can be implemented using Active objects only in section 3.1. Next, we will describe the implementation of the Stream in Fractive thus yielding the actual StreamComponents. Finally, section 3.3 will demonstrate the addition of the re-configuration mechanism whereby the Compute code of the StreamComponents can be changed at runtime.

In section 4, we will show how a more realistic example, image processing with the ImageMagick library, can be adapted by the user and deployed across a cluster via the StreamComponent system.

2 Fractal and ProActive

2.1 Fractal

The Fractal component model (Bruneton et al., 2004), provides a general (language neutral) model for Component based programming with implementations currently provided in Java, C++ (Object Web Consortium, 2008a), .NET (Object Web Consortium, 2008b) and Smalltalk (Bouraqadi, 2008); in our work, we use the Java implementation, Julia (Bruneton et al., 2006). The main advantages of using a Component-based approach in the context of our current work is that they can be started/stopped, re-configured and re-bound at runtime.

A Fractal component consists of a several parts as shown in figure 4:

- **Server Interfaces**: these provide values or services.

- **Client Interfaces**: these provide values or services.
services to other components via a binding; a Component may have zero or more server interfaces.

- **Client Interfaces**: these consume values or invoke services on a bound server interface; like Server interfaces, a Component can have any number of these. The Client and Server interfaces of a Component are called **Functional interfaces**.

- **Control Interfaces**: these control non-functional aspects of the Component. All Components must have the **LifeCycle Controller** (LC) which facilitates the starting and stopping of the Component, the others are optional. Components providing Client interfaces must also provide a Binding Controller (BC) to allow the Client interface to be bound to another Component which provides the Service. The optional Attribute Controller (AC) allows parameters of the Content to be changed at runtime.

- **Content**: this is the actual user or compute code which gives the Component its functionality.

### 2.1.1 Binding

Binding between Components occurs from a **Client** interface to a **Server** interface; this is because it is the Client which invokes the Server method. Binding can occur at anytime prior to, or during, execution. This allows a component-based system to be dynamically reconfigured by swapping in new components or adding new components as necessary.

### 2.1.2 Attribute Controllers

Attribute controllers (AC) are a means by which parameters of the Content or **Compute** code may be set prior to, or during, runtime. An AC is a standard Java interface written by the creator of the Content class and may be used to set/get the value of any instance variable but has the restriction that it must follow the Java Bean patterns: that is the getter/setter methods must be named, **getX** and **setX** where 'X' is the name of the particular attribute.

### 2.1.3 Fractal Example

To see how the various aspects of the Fractal model fit together, consider the following example of an Image processing system shown in figure 5. There are two main functional interfaces: **ImageIn** (client) and **ImageOut** (server) which consume and supply images respectively. Notice also that we draw bindings as an arrow which goes from the **Client** interface to the **Server** interface; as such, this is generally in the opposite direction to information flow. We have three Components in this system:

- **Image Datastore**: this Component obtains images from a store and supplies them, on request, via **ImageOut**. It also provides a list of available images via the **Index** interface.

- **Image Process**: this Component takes an image from **ImageIn**, compreses it using some algorithm contained in 'Compressor', and provides the output via **ImageOut**. It also as an attribute controller 'Compression Control' which can be used to set the level of compression.

- **Image Control**: this Component uses the other two Components to obtain compressed images. It obtains the compressed images, **Imagesn**, from the **ImageProcess** component. It also controls the amount of compression via **CompCont** which is bound to the **CompressionControl** AC of **ImageProcess**. Finally, it can obtain an index of available images via **IndexReader** which is bound to **Index** of the **Datastore Component**.

This example shows a number of benefits provided by a Fractal based implementation. Firstly, the design has been greatly simplified by separating the system into several distinct entities (the Components) which have clearly defined roles and also well defined interfaces between them. A benefit of this is that the various Components in the system can be replaced or new Components can be inserted at any time, both prior to, and after execution has commenced. For example if we decided to sharpen the image prior to compression, a 'Sharpen' component can be inserted into the 'processing chain' as shown in the figure. Alternatively, it may be decided that a different compression algorithm would be appropriate; in this case, we could replace the **Compressor** Component with the 'Improved' one.

Notice also that the **Attribute Controller** (AC) allows control of the **Compressor** Component's Compute code by altering the 'compFactor'. As stated above, an AC is a standard Java interface which provides methods to set/get various parameters. The requirements of the interface are that in must extend **AttributeController** and the methods must follow the 'Java Bean' style. An example AC for the image **Compressor** Component is shown in figure 6. The **CompressionController** could be used as follows:

```java
Component Compressor / Pointer to Compressor Component
CompressorController cc;
cc = new CompressorController();
cc.setAttributeController(compressor);
cc.setAttributeLevel(10);
```

### 2.1.4 ProActive + Fractal = Fractive

Our implementation of StreamComponents is based heavily on **Grid Components** which are part of the **ProActive** framework (Baduel et al., 2006; OASIS

---

4AttributeController is a 'blank' interface supplied by Fractal.
Reasearch Team, 2008). ProActive is a 100% Java middleware library for creating Active objects which possess the following properties:

- location transparency - that is method invocations are performed in the same manner whether the object is on the local machine or on a remote host. In the latter case, ProActive automatically and transparently facilitates remote communication with an object via RMI;

- asynchronous method invocations with Futures-based synchronisation - when a method is called on an active object, the caller can continue execution whilst the active object runs the method; the caller will only block if the result of the invocation is needed before it is ready;

- migration - active objects can be easily and transparently migrated between remote hosts.

The ProActive team has created an Active object based implementation of the Fractal component model called Grid Components (Baude et al., 2003) or Fracticle which combines the advantages of Active objects with Fractal components. This model facilitates the use of the Fractal components in a grid environment by providing:

- transparent support for remote inter-component communications thus making the components distributable;

- Grid components that can be migrated readily between remote hosts;

- infrastructure for remote deployment and monitoring. The monitoring is provided by a tool ICD which is included in the ProActive distribution.

3 Experiments with Stream Components

This section describes a series of experiments undertaken to develop the StreamComponents. For developmental purposes, we are concerned with the overall structure and architecture of the streams rather than what values are actually being passed through the stream: due to the Fracticle based design, the mechanisms we present in this section can be used later, without alteration, to construct more useful and elaborate streams.

3.1 ProActive Streams

As outlined in the introduction, there are two main types of StreamComponent: Stream and StreamF. In this section, we will implement and test the basic stream functionality of these two Components in a non-component setting; to do this, we will implement the simple stream model described in section 1.1 using standard ProActive Active Objects.

3.1.1 Stream Interface

All entities producing stream values will implement the Stream interface which consists of a single method:

```java
public value0WB car();
```

When `car()` is called, the next value in the stream will be returned. The next value is obtained differently depending on the entity:

```java
public class IntegerStream implements Stream{
    private int currentValue = 0;
    public value0WB car() {
        int temp = currentValue;
        return new value0WB(new Integer(temp));
    }
}
```

**Figure 7:** IntegerStream

```java
public class StreamF implements Stream{
    private Stream upstream;
    private unaryF myUnary;
    public StreamF(Stream s, unaryF f) {
        upstream = s; myUnary = f;
    }
    public value0WB car() {
        Object temp = upstream.car();
        return new value0WB(myUnary.apply(temp));
    }
}
```

**Figure 8:** StreamF

- In the case of a Stream Object or StreamComponent: these entities act as a ‘Stream Source’ so the next value in the stream must be obtained externally, or generated. For example, external values could come from a datfile or database, a human user, data-sensors, network etc. Values could also be generated algorithmically which is what we will do to produce the IntegerStream for our example.

- For a StreamF Object or StreamFComponent: these are ‘Stream to Stream transducers’. That is, they obtain a value by invoking `car()` on the ‘up-stream’ component, apply a function to the value and return the result. A consequence of this is that StreamF objects and StreamFComponents must always be connected to ‘up-stream’ components in order to function.

Values are always passed through the stream within value0WB wrapper objects; this is evidenced by the return type of `car()`. An alternative would be to have `car()` return the value directly as an Object. However, having methods which return Object in an Active object or Fracticle Component causes difficulties when attempting to cast it back to its actual type; see (Martinaitis and Wendelborn, 2008) for a more detailed discussion.

3.1.2 Stream and StreamF Objects

In this example we will use a simple IntegerStream as the stream source; the implementation is shown in **Figure 7**. As can be seen, it produces a stream in a co-routine like fashion by returning the next Integer before updating its internal state. The stream function, StreamF, also implements the Stream interface as shown in **Figure 8**. A StreamF constructor takes in a source stream s and a unary-function (unaryF) object f. The unaryF interface provides one method:

```java
Object apply(Object x);
```

which will apply some unary function to x and return the result; an example unaryF which simply

```java
1: public class TimesTen implements unaryF {
2:     public Object apply(Object x){
3:         int temp = (Integer)x.intValue();
4:         return new Integer(temp * 10);
5:     }
}
```

**Figure 9:** Example unaryF - TimesTen
multiplying the argument by ten is shown in figure 9. Returning to figure 8, we see that the `car()` method obtains the next value from the 'upStream', applies the unary-function \( f \) to it and returns the result within a value\_OB object. Figure 12 shows an example stream `composition` consisting of the two objects just described: that is, an IntegerStream joined to a 'TimesTen' StreamF. A 'StreamEater' class is also provided: this provides a simple GUI which, with every push of a button, invokes `car` on the source stream, thus consuming a single stream value, before displaying the result. The demand flow through the stream is shown by the dotted arrows and the returned values are shown by dashed arrows. We see that the StreamEater obtains the next value from the stream by invoking `car()` on its 'upStream' object; in this case StreamF. The StreamF then, in turn, invokes `car()` on its upStream; IntegerStream. IntegerStream then returns the next value (3) before updating its state to 4. StreamF, having now obtained the operand, can apply its unary-function to it (`myFunc`) to yield the value 30. This value is then returned to the StreamEater as the result.

Two versions of the main program are shown which produce this example stream: one creates conventional objects (figure 10) and the other Active objects (figure 11). The conventional version is straightforward: looking at the ProActive version, the relationship between them can be seen. To create an Active object the `newActive` method is used; it has the following signature:

```
Object newActive(String className, Object[] constructorArguments)
```

Thus to create an active IntegerStream object, the first line of figure 11 calls `newActive` with the arguments of the className ("IntegerStream") and an empty object array as the IntegerStream doesn’t take any arguments. TimesTenS is created similarly except that the two constructor arguments (the 'Ints' and 'TimesTen' objects) are passed in via an array.

### 3.1.3 Demonstration

Throughout our examples, we will make use of the IC2D tool which is provided as part of the ProActive distribution. IC2D is a graphical tool which provides a means of monitoring and controlling Active objects, both local and remote, across a cluster of machines. To test the IntsTimesTen stream, we ran it on a cluster of three machines as shown in figure 13 which shows the IC2D display together with the StreamEater. The IC2D display shows the three machines (Orcat, dlp1c3 & dlp1c7) with each one running a single ProActive Node\(^5\) (Node7285..., Node-d13 and Node-d17 respectively); the state of the stream is the same as in figure 12 where the value 30 has just been retrieved. As can be seen, both Active objects are displayed and originally start on the local node `Orcat`. The arrow from StreamF and IntegerStream indicates a method invocation between the two objects; in this instance `car()`: it is important to realize that the arrow directions in IC2D are opposite to the 'stream flow' direction because of this. Note also, that the StreamEater does not appear on the IC2D display because it is an 'ordinary' object.

Migration of Active objects can be initiated from within the program or by 'dragging' the objects to the desired Nodes in IC2D; we did the latter placing each Active object on a different node and retrieving another value from the stream as shown in figure 14. The IC2D display shows the objects residing on their new hosts and due to the location transparency provided by ProActive, the method invocations and return values between the two remotely separated objects are automatically transmitted over the network. The StreamEater is still on Orcat so there is also a 'remote communication' between StreamF and StreamEater which is not shown in the IC2D display.

At this stage, we have demonstrated a simple mechanism for expressing stream-based computations and shown that this model can be successfully deployed and executed as Active objects. Although we have used Integers in this example, any `serializable` objects can be passed through the stream via the val...

---

\(^5\)A ProActive node is an entity, identified by a unique URL, which runs on each host involved in the computation. Active objects can be transferred between Nodes via the migration mechanism.
3.2 Fractive Streams = StreamComponents

In the previous Section, we have shown how to use ProActive to construct a stream with properties of incremental evaluation, runtime migration of stream transducer functions, and monitoring. We will now describe how the objects: Stream and StreamF can be implemented as Fractive components thus building a framework for exploring dynamic reconfigurability.

3.2.1 Stream Component

A Stream component, shown in the left of figure 15, encapsulates a Stream object as its compute code which will produce the stream values. Any object which implements the Stream interface can be a compute object including IntegerStream from the last section. The server interface of the component is also implemented as Stream and is named 'StreamSource' as it supplies values to downstream components via the $car()$ method; in fact an invocation of $car()$ on the component interface ‘StreamSource’ is simply redirected to a call on the internal compute object $myStream$.

As well as the standard lifecycle controller(LC), there is an AttributeController provided called StreamCompAttributes which has the following methods:

```java
public Stream getStream();
public void setStream(Stream newStream);
```

This controller is used to set (or reset) the compute code ($myStream$) to a different object; this can be done at anytime including after execution of the Stream has begun. Thus by using an AttributeController to set $myStream$, we have allowed the actual process generating the stream data to be dynamically configured and changed at runtime: this could be used, for example, to switch between different data sets - see section 3.3 for a detailed description of our reconfiguration mechanism. Following on from our previous example, we could generate a Stream of Integers by setting the $myStream$ attribute to IntegerStream from section 3.1.

3.2.2 StreamFComponent

A StreamFComponent, shown in the right of figure 15, has the same purpose as the StreamF objects described in the previous section: namely, to apply a unary function to an incoming Stream of data. This Component has two functional interfaces:

- StreamSource: a server interface to obtain next value of the Stream.
- StreamSink: a client interface which obtains stream values from an 'upstream' source.

The StreamSource component interface is again implemented as Stream and therefore provides the method ‘$car()$’. Internally, the StreamF component operates in the same way as the StreamF object. There is a reference to the 'upStream' Stream component which has the same role here as in the StreamF object except that instead of being set by the constructor, it is now set when the client interface StreamSink is bound to the server interface of an 'upstream' component; this is done by the binding controller (BC). Figure 16 shows the example stream used in the previous section re-implemented using StreamComponents. It can be seen that from a component perspective, we view the binding between the StreamF and Stream components as from the client to server interfaces but it is actually implemented by setting the internal 'upStream' reference of StreamF to the Stream component (shown by the dashed arrow).

An AttributeController, StreamFCompAttribute is provided to allow the unary function used to be set and provides the following methods:

```java
public int unaryF(int x);
public void setUnaryF(int unaryF);
```

This interface is analogous to the StreamFAttribute controller of the Stream component. The $car()$ method of the StreamF component is essentially identical to that of the StreamF object shown in figure 8.

3.2.3 Stream Component Factory

To make it easier and faster to work with StreamComponents, we have written a StreamComponentFactory (or SCF) class to create these Components as
public class IntsTimesTenFact {
    public static void main(String[] args) {
        // StreamComponentsFactory SCF = new StreamComponentsFactory();
        IntegerStream Ints = new IntegerStream();
        StringStream SS = new StringStream();
        IntsTimesTen SSInts = new IntsTimesTen();
        String timesTen = "Times Ten";
        int bound = 10;
        int boundTimesTen = bound * 10;
        IntsTimesTenStream stream = new IntsTimesTenStream(IntsTimesTen.class);
        String streamID = String.valueOf(boundTimesTen);
        StreamComponents SCF = stream.getComponents();
        SCF.bind(IntsTimesTen, timesTen, timesTen);
        // Start Components
        SCF.startComponents();
        // Attach StreamEater and start GUI
        StreamIntsInts = new StreamInts(IntsTimesTenStream.class);
        // ComponentGUI compGUI = new ComponentGUI(componentsList);
    }
}

Figure 18: IntsTimesTen - StreamComponents version

well as provide set of utility methods. By using the SCF, most of the component-specific code is hidden from the user. The most important methods provided by the SCF are shown in figure 17.

There are four creation methods provided for each Component. The first just takes the name of the Component; the ‘name’ is an arbitrary label supplied to identify the component in the StreamComponents GUI and other environments. The others allow combinations of a compute class and/or a non-local node for remote deployment: the compute class is a stream generator for Stream Components or a stream transducer for a StreamF Components.

As an example, to create the 'Integer stream' discussed in the previous example, we can now use:

    IntegerStream Ints = new IntegerStream();
    ComponentInts = SCF.createNewStreamComponent("Ints", Ints)

where the streamGenerator object ‘Ints’ is defined in figure 7. If we instead create Ints with:

    ... SCF.createNewStreamComponent("Ints")

then the streamGenerator can be set later using the AttributeController of the Component.

There are two methods to indicate if an SC is a Stream or StreamF and also a polymorphic bind method which allows the easy joining of two Stream Components: of course, the method can only bind valid combinations of arguments namely:

* Stream -> StreamF
* StreamF -> StreamF

Note that the arguments to the bind method are in 'stream order'; that is values flow from the left argument to the right argument; this is opposite to the 'binding direction' drawn on component diagrams where we regard binding as occurring from client to server.

For changing the execution state of Components, that is to Start/Stop them, two methods are provided: one to act on a single Component and one to act upon a group of them represented as an ArrayList.

Figure 19: IntsTimesTenStream - Distributed

3.2.4 Demonstration

The example stream used previously, IntsTimesTen, was rewritten to use the StreamComponents as shown in figure 18. On lines 2 & 3, we instantiate the two compute objects IntegerStream and TimesTen which we then use to create the two StreamComponents on lines 4 & 5; notice that in this case, we also supply a remote node address (rmi://lhp17 ... etc.) which results in the components being immediately deployed on the remote nodes rather than starting on the local host as was the case previously. We bind the two components together, in the order of streamflow, on line 6 using the bind method. The components are then added to an arraylist so that they can be started (line 10) and also passed to the StreamComponent GUI (line 14); the StreamComponent GUI was developed to assist in the monitoring and control of StreamComponents and will be described below. Finally, the StreamEater is attached to the head of the stream (lines 11 - 13).

The example program is shown running in figure 19 at the same stage as the object version of figure 14. The IC2D display is now showing the Fractive components rather than the objects: thus the Stream is now labeled StreamCompImpl rather than IntegerStream and StreamF is now StreamFCompImpl.

The value 40 has just been retrieved from the stream; thus there is an arrow between the components indicating the invocation of car() on the Stream component by StreamF; again notice that this is opposite to the 'stream flow' direction.

Below the IC2D display is the StreamComponent GUI mentioned previously. We can see that for each component, there is a horizontal panel indicating the components name, execution status (STARTED / STOPPED) and the class of the compute object currently being used: in this example IntegerStream and TimesTen for the Stream and StreamF components respectively. Each panel also has two buttons to facilitate reconfiguration of the compute code: ‘Change Class’ and ‘Load Class’. We will describe the function of these two buttons in section 3.3.

At this stage, we have shown how the Stream and StreamF objects from section 3.1 can be re-expressed as Fractive components. We have seen that the Fractive components are, like Active objects, distributable and location-transparent. The main advantage of using the Fractive components is that of re-configurability and it is this aspect which we shall describe next.
3.3 Reconfiguration

3.3.1 Static Reconfiguration

As described previously, each Stream component has an attribute controller which provides methods to allow the compute code of the component to be changed. They are:

```java
public void setStream(Stream newStream);
public void setUnaryF(unaryF);`

for the Stream and StreamF components respectively. Clearly, each of these methods requires an actual instantiated object to be passed at runtime to effect the change. Using an attribute-controller directly is somewhat messy as the controller has to be obtained from the component before being used as shown in the ‘Compression Controller’ example in section 2.1.3. To make this process easier, we provide a utility method in the StreamComponentFactory:

```java
public static void changeStreamFunction(Component C,
unaryF newFunction)
so we can achieve the same effect with simply:

`C.changeStreamFunction( FunctionComp,
newTimesHundred());`

An analogous method is provided for the Stream component. The main limitation of this mechanism is that the new compute code has to be instantiated statically when the driver code is compiled which is very limiting. We want the configuration mechanism to be more dynamic by allowing compute code to be instantiated at anytime, as dictated by the user of the stream. To facilitate this, we have provided a dynamic configuration mechanism which we will describe next.

3.3.2 Dynamic Reconfiguration

In the static mechanism, the new compute object (Stream or unaryF) had to be instantiated statically by statements compiled into the driver code. With the Dynamic Instantiation mechanism, this can be done any time by using the following method6:

```java
public static void changeStreamFunction(Component C,
String SClass)
Like the static version, this method takes in the component to be reconfigured ‘C’; this time, however, the compute code is passed as a class-name rather than an object. Provided that the classfile corresponding to SClass is present in the classpath of the localhost, this method will instantiate the corresponding object and swap it into the component as the new compute code. So our example would now become:

`C/changeStreamFunction( FunctionComp, “TimesHundred” );`

The main advantage of this mechanism, is that the decision as to which compute code to use can now be decided at runtime by a user or by automated software which may be overseeing and controlling the execution of the stream. Additionally, provided that the compute class is compiled just prior to the reconfiguration, it can be written/changed at any time up until that point; this would obviously not be possible with the static reconfiguration mechanism. In our StreamComponent GUI, the ‘Change Class’ button brings up a dialog so that the name of the class to be used can be entered; it is then passed to the changeStreamFunction method. We also provide a ‘Load Class’ button on the GUI so that the classfile can be selected directly from the filesystem.

6There is also an analogous method for the Stream component.

Figure 20: Reconfiguration of IntsTimesTen

3.3.3 Demonstration

To demonstrate the reconfiguration mechanism we return to the IntsTimesTen stream at the same stage as shown in figure 19; that is just after the value 40 has been retrieved from the stream. Looking at figure 20, we see the stream in this state just after the ‘Load Class’ button of the StreamF has been pressed. This has brought up a file chooser where we are about to select the new compute class, ‘TimesHundred’. It’s important to note that the GUI is running on the local host Oracle whereas the StreamF component being reconfigured is still on ‘hpcl13’ as seen in the ICID display. Hence when the reconfiguration occurs, the dynamic classloading mechanism will load the classfile from Oracle’s filesystem, instantiate the compute object and then Fractive will automatically serialize the object and transmit it to the remote StreamF component being reconfigured. Since the next value of the Ints stream is 5 and we now have ‘TimesHundred’ as the StreamF compute, the next time ‘Next Value’ is pushed, it will display 500.

4 Applications

In the last section, we showed a series of experiments which outlined the development of StreamComponents using simple examples, in order to focus on the mechanism and nature of the components. We will now present a more realistic example, showing how a user of StreamComponents can use a pre-written system, namely ImageMagick (ImageMagick Library, 2008) as the compute code for the stream. ImageMagick is a C++ based image manipulation library which provides the ability to load, manipulate, process and write image files in a wide variety of formats. There are interfaces available to enable the library to be used from a wide variety of languages; we use the Java implementation JMagick (JMagick, 2008). JMagick (JM) is essentially a wrapper library which provides an object-oriented interface to the ImageMagick(IM) system by using the Java Native Method Interface to dispatch to the actual IM C++ methods when performing the image loading and processing. This retains the speed of the IM implementation but, as a consequence, results in the image-data being stored in the native C++ representation: since these will be our stream values, we will have to take this fact into account when designing our compute classes.

For this example, we want to create the Image stream shown in figure 21. It consists of a Stream component at the end with a SingleImageStream as its compute object. The SingleImageStream object will read in an image file from the local filesystem and then supply the loaded image as a stream.
value. There will then be a number of StreamF components composed together each performing a specific ImageMagick function (IMFunc) and finally a StreamEater will be attached to the head of the stream; the StreamEater in this case is a simple 'picture viewer' which will display the image obtained from the stream when 'Next' is pushed. We will now describe how the compute classes can be written.

4.1 SingleImageStream

The code required to use JM to load an image, rotate it 90 degrees and then 'blur' it is as follows:

```java
// Load Image
MagickImage image = new MagickImage( new ImageInfo("TestImage.jpg") );

// Process Image
MagickImage rotated = image.rotateImage(90.0);
MagickImage blurred = rotated.blurImage(0.0, 2);
```

So it can be seen that the library is very straightforward to use and also that the image is represented as a MagickImage object. It would seem natural, therefore, to use MagickImage objects as our stream values; the `car()` method of SingleImageStream could then be simply:

```java
public valueOf<car>()
{  
MagickImage image = new MagickImage(  
new ImageInfo("TestImage.jpg") );  
return new valueOf<image>();  
}
```

However as stated above, there is a complication in that the MagickImage class stores a pointer to the C++ datastructure which represents the image for use with the native C++ methods; clearly, this class is not serializable and therefore cannot be used as a stream value. Fortunately, JM provides a set of methods to convert the MagickImage representation to/from Blobs which are in fact an array of bytes. Since a byte[] is serializable, SingleImageStream can be implemented as above as elaborated above in 4.1.

4.2 IM Functions

The implementation of IMFunction compute classes are fairly straightforward. For example to implement the IMRotate90 `unaryF`:

```java
public Object apply(Object x) {
byte[] nextImageBlob = (byte[]) x;
MagickImage nextImage = BlobToImage(nextImageBlob);
MagickImage transformedImage = nextImage.rotateImage(90, 0);
return ImageToBlob(transformedImage);
}
```

We convert the argument `x` from a Blob (byte[]) back to an MagickImage, then apply the `rotateImage` function. The resultant image is then converted back to a Blob before being returned. This is the definition for the class `IMRotate`; the other IM functions are implemented in exactly the same way except that the corresponding function is used in place of `rotateImage`.

4.3 Demonstration

To demonstrate the generic nature of StreamComponents, we are going to reconfigure the running `IntsTimesTen` stream shown in figure 20; remember that this stream has just been reconfigured with a compute of `TimesHundred` and that the value 500 has just been obtained. To effect the reconfiguration, we use the 'Load Class' button as before: we selected the `SingleImageStream` for the Stream component compute and `StreamIdentity` for the StreamF. StreamIdentity is a unaryF which returns its input unaltered. The sample image supplied by the `SingleImageStream` is the University logo as we can see in Figure 22 which shows the reconfigured stream after the 'Next Value' button has been pressed. Looking at the Java 2D display, it can be seen that the components are still distributed in the same way. When 'Next Value' was pressed, demand propagated back to the Stream component on D17 which then loaded the logo from the filesystem and transferred it to the StreamF on D13. The StreamF applied the StreamIdentity function, hence the image was unaltered when the value was passed to the StreamEater on Oracle.

We then reconfigured the StreamF to use the IMRotate compute class and obtained the next value as shown in figure 23; here the image has been rotated 90 degrees to the right as expected. In this example, we have shown how the same stream can be used with two very different types of values provided that the compute objects used have been written to handle the particular values. As a final example, we will demonstrate the composition of four StreamF components together to produce a more complex stream as shown in figure 24. As before, there is a SingleImageStream residing on D17 but there are now also four StreamF components
in the stream distributed over all three nodes. Looking at the IC2D display, the arrows correspond to the bindings between the components (which is opposite stream flow direction) whilst the GUI shows the components in order from top to bottom. Note also that the names given to the components in IC2D do not match those in the GUI.

To test the composition, we configured the first three StreamF components to perform a rotation and the fourth to perform IMBlur. Thus the net result should be a blurry logo rotated 270 degrees to the right; looking at the figure, this is indeed the case.

5 Summary and future directions

In this paper, we have presented a simple model of stream evaluation and shown how we used this as the basis for designing StreamComponents (SC) to facilitate stream based computation on the Grid. We have based our implementation on Fractive components which are in turn implemented using ProActive; this enables our SC to be location transparent and migratable. We have taken advantage of the reconfigurability of Fractive components to develop a mechanism by which the compute code of the components can be developed, compiled, and then inserted into the SC at anytime. Being components, the topology (that is inter-component bindings) can also be modified at runtime. We have also demonstrated that the stream structure is totally separate from the values it carries. In particular we have shown that any code libraries can be used as compute code (including those written in other languages) provided that the values they produce are representable in a serializable form.

There are few re-configurable component-based systems which are targeted specifically at computation on the grid; however, one example is Higher Order Components or HOC (Dunnweber and Gorlatch, 2004). HOC provides a means of constructing parallel skeletons (common parallel patterns) which are parameterized by compute code: this is similar in concept to SC where we provide separation between stream topology and the compute code. A major difference between the systems is that the structure of a HOC program, that is the choice and implementation of the skeletons, is fixed at compile time. By contrast, SC allows reconfiguration of both the compute code and the structure of the computation; that is, the stream topology.

We are currently investigating the incorporation of Web Services into the system to allow SC to be connected by both the Fractive 'remote methods' shown in this paper as well as via web-service connections using SOAP. At present, the SC are tightly coupled to one another with only one value flowing through the stream at a time. To take advantage of potential 'pipelining parallelism' which is available in a long chain of StreamF components, we intend to investigate more eager modes of stream evaluation.

References


ImageMagick Library (2008), Project Homepage: http://www.imagemagick.org/.


