

JPie Interface: An On-The-Fly Job Submission And Communication Configuration Package

Viktor Yarmolenko[†] Paul Cockshott[†] Ewan Borland[†]
v.yarmolenko@dcs.gla.ac.uk wpc@dcs.gla.ac.uk borlaned@dcs.gla.ac.uk

[†]Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, United Kingdom

ABSTRACT

This paper describes a new Java interface loosely modeled on the primitives of the π -calculus [14] to be used as a substratum for Grid based parallel computing. It allows the creation of processes and communications channels between processes. It also allows for the communications network between processes to be dynamically reconfigurable. The aim of the design is to achieve this with the minimum number of primitives and to integrate these primitives into the existing Java class framework.

Categories and Subject Descriptors

H.4.3 [Information Systems Applications]: Communication Applications; D.2.2 [Software Engineering]: Design Tools and Techniques—*modules and interfaces, object-oriented design methods, software libraries, user interfaces*; E.1 [Data]: Data Structures—*distributed data structures*

General Terms

Design, Performance

Keywords

Java, Grid, Dynamic, Distributed, Process Algebra

1. INTRODUCTION

The problem of dynamic job submission and live communications migration was faced by the PGPGrid Project [11], a core aim of which was to parallelize the process of extracting *range data* from the experimental 3D-Matic TV scanner [19] for 3D computer animation. The scanner uses 24 synchronized video cameras (8 groups of 3), each of resolution 640x480 pixels, to image a subject from many directions at once. The data thus acquired is then used to build up a dynamic 3D model with a spatial resolution of around 4mm and a temporal resolution of 0.04 seconds. Following capture, each of eight synchronous pair of monochrome images ($2 \times 300\text{Kb}$) must be processed by a stereo matcher algorithm ($\sim 50\text{sec}$ on P4 2GHz) to produce *range data*. *Range*

data from all 8 groups (9.6 MBytes) defines a point cloud which is now passed to a model-builder process responsible for creating a triangulated VRML model of the whole subject ($\sim 40\text{sec}$ on P4 2GHz). The data rate of the entire model building process exceeds 1GByte/sec. Fortunately, the processes involved are easy to parallelize, although there are substantial challenges involved due to the sheer volume of data being generated and moved between processes.

One of the primary goals is to use Grid technology to acquire the resources needed to meet the challenges outlined above. Existing architectures for parallel computing do not provide a solution without substantial adaptation. Considering this we used π -calculus [14] as a model around which we loosely built a new Java interface. In π -calculus processes read and write data into channels. Channels themselves can be sent through other channels. The interface described in this paper is intended for use by the application described above, but will be useful for other Grid-based parallel computing applications. It allows the dynamic creation of both remote processes and the communications channels between them. It also allows dynamic reconfiguration of the network of channels. We tried to achieve this with the minimum number of primitive types, integrating these into the existing Java class framework. It is at roughly the same abstraction level as JPVM [12] and IceT [15].

We are not alone in using π -calculus as a basis for our research. A group from ICL proposed a Java API [7] which used a $\delta\pi$ -calculus as basis for their mobile distributed computation model. They used TCP and RMI libraries for remote communication. However, their solution requires a higher number of primitives and appears to be more complex than that proposed in this paper. In addition, they do not fully address the properties of channels and pipes as described in [14, 17]. Other attempts to implement π -calculus include PiLib [18] based on Scala language, specifics of which make it possible to use almost the same syntax as in the π -calculus, whereas in projects like Pict [9], Facile [6] and others [8] a new language was developed to map onto π -calculus style channel based communication. In [16], a Java library (JSCP) was developed to provide an extended version of the CSP/occam model for Communicating Processes.

2. THE INTERFACE

The interface, $J\pi$, consists of only four primitive types. These are JPieTap, JPieFunnel, JPiePipe and JPieTask (further on these in 2.1). These relate to the terminology in

π -calculus as follows. A channel in π -calculus is represented as a JPiePipe (Pipe) in $J\pi$ and it contains a JPieTap (Tap) and a JPieFunnel (Funnel) connected together. The Tap and Funnel extend the Java Input- and OutputStream classes respectively. A process in π -calculus is a JPieTask (Task) in $J\pi$ and extends Java's Runnable Interface. It may contain any number of Taps and Funnels as its member variables. All four classes are serializable.

There are two distinctive features of $J\pi$. The first is the ability to send Taps, Funnels, Pipes and Tasks *over* Pipes from one Task to another. The unusual feature of this is that when say a Tap (e.g. created from System.in) is sent to a Task on a remote VM, the Task is still able to read data from the original stream. The second feature is the ability of the remotely spawned Task to recursively spawn another remote Task. The existing member Taps and Funnels of the Task, as well as any received Taps and Funnels, can then be passed into yet another spawned Task. The latter is then able to read data from the original stream.

This concludes the description of the parts of the $J\pi$ implementation that are visible to the user, the class hierarchy of which is detailed in 2.1. In sections 2.2-2.4 we discuss some of the implementation issues of $J\pi$.

2.1 A Class Hierarchy

The overall $J\pi$ class hierarchy is as follows.

1. **abstract class JPieTask** represents the unit of work to be done as a parallel process. It implements the java.lang.Runnable interface, allowing the construction of threads from JPieTasks. It has additional methods that allow channels to be associated with JPieTasks. Before the JPieTask run method is called, JPieTask's transmissible input and output channels, known respectively as JPieFunnels and JPieTaps, must be initialized. A running task can obtain its taps and funnels via the *get* methods supplied.
2. **interface JPie** is a factory interface whose job is to create JPieTasks, JPieTaps, JPieFunnels and JPiePipes (a pipe is a linked tap and funnel). It has a core method, *spawn*, which causes a JPieTask to be run locally or remotely depending on resources. Time and memory parameters which specify anticipated resource usage are supplied. Spawn will fail if adequate resources cannot be found.

JPie also includes methods to create JPieTaps and JPieFunnels that can convert Java streams in the local environment to JPie streams that can be sent to remote machines. The createPipe method also permits interprocess communication streams to be set up between two daughter tasks.

The set of classes required by a remote task are made available to it via a custom class loader which retrieves jar [13] files containing class definitions from the originating process. The set of jar files available to the class loader is specified via a call to the setJars method of the originating JPie instance.

There is also a need to be able to identify sources of data at known locations on the Web. JPie will provide

for this by adding methods that allow for taps and funnels to be published and associated with URIs.

3. **abstract class JPieTap** implements java.io.InputStream, and requires the implementation of a method to get a byte (read). The standard Java blocking protocols are followed.
4. **abstract class JPieFunnel** implements java.io.OutputStream and must provide a write method that writes a single byte to the output channel. It can block on write if the remote process has not performed sufficient reads.
5. **interface JPiePipe** provides a bi-directional inter task communications channel. This has two methods: getTap() and getFunnel(). To set up a data flow graph one creates pipes and passes the input and output channels to tasks which are then forked. It is implemented by using built-in Java classes PipedInputStream and PipedOutputStream; however, a more sophisticated implementation may be arranged. The key issue here is the serialization of the Pipe, Tap and Funnel objects for their transfer between processes.

The overall aim of $J\pi$ is for processes to be able to run across a network of machines without knowing where these machines are, or having to reference the machines explicitly in any algorithm. An algorithm expressed in Java using $J\pi$ should be invariant under alterations in the physical collection of machines that run it.

One of the great strengths of Java is that it allows a binary image to run on a variety of different physical computers, each of which may be running a different operating system and using a distinct instruction set. In order to ensure that a Task will produce the same result regardless of its host, every machine that is to run $J\pi$ Tasks must have the JRE [10] installed and have access to the $J\pi$ library. Data can be transferred to or from a Task over its Taps and Funnels.

2.2 Resource Locator Model

$J\pi$ can make use of various resources from a pool, but requires a mechanism to identify, check for availability and access these resources. This is where the idea of a Service Locator comes in. The model we have proposed has three components: the Initiator (Initiator), the Service Locator (Locator) and the Server Entity (ServEnt), referred to collectively as the ILS model.

A typical interaction of the components in the ILS model would be as follows: A ServEnt advertises its hardware characteristics and availability for $J\pi$ Tasks by informing the Locator, which maintains a list of this information for several ServEnts. An Initiator wishing to launch a $J\pi$ Task requests suitable resources from the Locator. The Locator returns a list of resources, from which the Initiator then confirms the first resource's availability directly with the ServEnt and, if it's available, submits the $J\pi$ Task. On Task completion the ServEnt passes the job metrics back to the Initiator.

- **Locator** maintains an updateable list of resources with their associated static and dynamic information. The Locator stores this information in a database such that

it can be maintained, accessed and searched quickly and efficiently. The Initiator and ServEnt can respectively request and update this information via two Webservices [4] made available by the Locator. The Locator is hosted within a Tomcat [1] server, and utilizes the Axis [2] framework for constructing SOAP [3] messages and Webservices.

- **ServEnt** acts as the access level for the ILS model to utilize the underlying resource. When the ServEnt is installed on a machine, the user must supply the architectural information (based on XRSI [4]) that is in turn passed on to the Locator. The ServEnt also can accept $J\pi$ Tasks from an Initiator, and will pass metrics of the task run to the Initiator when it has completed.
- **Initiator** instigates $J\pi$ Tasks. It requests resources from the Locator and receives the *addresses* of the resources available and appropriate to the job. The Initiator negotiates with the resources in turn via their ServEnts to discover if they can accept the job. If declined by one resource, then the next ServEnt is negotiated with. Once the negotiations have completed, the Initiator submits the Task to the ServEnt.

There are several technologies already in existence that offer similar functionality to the interactions described above, and a survey of these was undertaken as part of [11]. However, none of the technologies matched the requirements exactly, and those that came closest were relatively heavy-weight solutions, which would have required significant investment in terms of time and effort in adapting them to our requirements. Due to the inherent difficulties of such an approach, and given the time constraints, it was decided to utilize the Webservices paradigm. This allowed an iterative approach to developing the functionality, and the resulting implementation should also be straightforwardly extensible to the emerging WSRF [5] standards.

2.3 Job Submission

The abstraction of π -calculus is not concerned with the physical location of its processes. However, in a Grid environment typically the resource handles must be known *before* the job is started. $J\pi$ does not contain lists of such handles. Indeed, in the dynamic resource discovery paradigm static lists are virtually useless. In $J\pi$ this will be implemented using the ILS model described in 2.2. The Locator is essentially a set of Services. When an Initiator queries a Locator, the former obtains a list of handles of random remote resources, which are adequate for the Task to be spawned. Once the handle is received the spawning of the Task is performed by the Initiator communicating directly with the remote resource (i.e. ServEnt). If the negotiation between the Initiator and ServEnt is successful the Task is executed, otherwise Initiator tries to connect to another ServEnt on the list. The negotiation process is essential as it ensures that all criteria are met for the successful job completion. Currently, these criteria enlist the normalized expected execution time of the Task, the time Initiator is prepared to wait until the Task is finished (e.g. for slower CPU) and memory required for the Task. The negotiation protocol may be extended to include other parameters, such as network bandwidth, the charge rate for CPU cycles, *etc.*

After instantiating $J\pi$ and creating Taps and Funnels from the standard Java streams, Taps and Funnels are passed into the user-extended Task. The Task is serializable as are Funnel and Tap, thus they are ready to be sent out to the remote resource. When the JPie.spawn(...) method is called the serialized Task must be sent to the ServEnt (the handle for which the Initiator obtained from the Locator). When the ServEnt agrees to run the Task it creates a new instance of the VM and any data is then exchanged between two VMs (i.e. InitiatorVM and a new VM controlled by the ServEnt) via Taps and Funnels that have been previously instantiated. Note, that every individual user-extended JPieTask is always executed on a new VM. This is done to prevent abnormal interactions between Tasks, possible VM memory overflows, security breaches, *etc.* It is up to the ServEnt, managed by a user, as to how many separate Tasks (i.e. VMs) are permitted to run simultaneously.

When the Task is loaded from the initiating VM to the executing VM, the class definitions for the Task are loaded through $J\pi$'s customized class loader. The Task's Taps and Funnels are also deserialized, however the class definitions are loaded locally. Note, the deserialization mechanism for Taps and Funnels ensures that the data is read/written from and to the respective streams on the initiating VM. Once all the fields are deserialized the Task is executed. After the completion of the Task a notification of the job done is carried out between all the relevant parties. In 2.5 we present a simple example where this is described in more detail.

When a Task is recursively spawned N times any Taps or Funnels that are passed from the initiating machine to the N -th Task through its predecessors will be *connected* to the relevant streams on the initiating machine directly. Also, when the initiating machine concurrently spawns a chain of tasks A and B , where task A_N creates a new Tap and passes it to B_N , this Tap will *connect* to its original stream directly through the ObServ of A_N .

2.4 Communication

As was mentioned earlier, Taps and Funnels of $J\pi$ can be sent down through other Taps and Funnels. This is a potential issue, because local java stream types must be able to present data written in another spatially different VM. Therefore, we need to be able to link java objects using globally referenced unique IDs.

In $J\pi$ this is implemented using a static INDEX and a daemon acting as an Object Server (ObServ). INDEX holds references to all Taps and Funnels that have been sent down Pipes to remote Tasks and is not available to the user. ObServ is another hidden component of $J\pi$. Like the INDEX it is started whenever $J\pi$ is present on a VM. ObServ accepts requests from remote VMs on one side and accesses the INDEX on the other. ObServ is uniquely identified on the Internet. There can be only one ObServ for each instance of $J\pi$. Also, there can be only one instance of $J\pi$ for a single VM.

$J\pi$ objects are moved across the distributed resources and reconstructed on the other end using, for example, a standard java.io.Serializable interface. Tap and Funnel extend java.io.InputStream and java.io.OutputStream respectively

and cannot be serialized conventionally. Therefore, an illusion of the standard serialization is performed, in which all the member variables of a Funnel or a Tap are serialized except the transient underlying stream variable. When a Funnel or a Tap is serialized a unique *key* is written to the INDEX. The INDEX associates the underlying stream with the relevant *key*. During the deserialization of $J\pi$ objects on a remote resource, this *key* is used to *connect* the local stream of the Tap or Funnel with the stream on the original VM. We omit a description of this *reconnection* and only mention that a *key* contains all the information necessary to find and *connect* the deserialized object with its original live stream via the ObServ of the originating VM. Registering with the INDEX and then *connecting* is done by overriding methods:

```
private void writeObject(java.io.ObjectOutputStream out)
private void readObject(java.io.ObjectInputStream in)
```

A Pipe consists of a Funnel and a Tap which are already serializable, thus the implementation of the Pipe falls back to simply implementing the `java.io.Serializable` interface where all its fields get serialized by the default Java mechanism.

2.5 A simple example

Consider the case when a single Task is spawned (for Java code see Appendix A). The Task simply reads characters from its Tap, processes them and sends a new sequence back via its Funnel. The Java code for the Task itself is presented in Appendix B. The diagram in Figure 1 shows a somewhat simplified version of the $J\pi$ implementation of the spawn.

INDEX is created and ObServ starts automatically with $J\pi$. After this all the Taps and Funnels are created. The entries for the relevant objects are written in INDEX before the Task T is spawned (step (1) in Figure 1). This happens behind the curtains without user intervention.

When the `JPie.spawn(...)` method is called the remote resource is discovered first (step (2)) through the Locator. Once the handles to ServEnts are obtained the Initiator engages in negotiation (step (3)) by sending memory (m), time (t) and other relevant requirements and constraints. If negotiation is successful the ServEnt starts a new VM and Initiator sends T to it (steps (4) and (5) respectively).

T is received (step (5)) and deserialized (step (6)) (T') using class definitions obtained from Initiator by the custom class loader and all the member variables are deserialized with it. Among them are Taps and Funnels which *connect* to the original streams on Initiator (step (7),(8)).

Now T' is executed (Figure 1). It asks for a name by sending a request to the Funnel (Appendix B). This request appears on the user's screen since the Funnel was created from system out (Appendix A). The user then enters his name which is read by the Task and greetings are sent back to the Funnel. The message appears on the user's screen. Then the Task terminates.

In the end of execution of the Task T' (step (9) in Figure 1), ServEnt notifies Locator about the completion, in order for Locator to update its *available resources*-list (step (11)). Also, ServEnt notifies Initiator (step (10)).

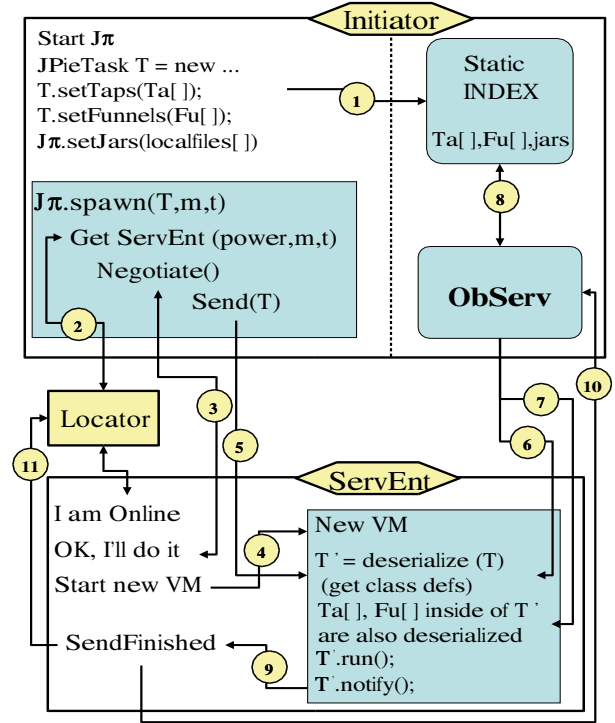


Figure 1: A diagram of the relationships within the Initiator and between the components of the ILS

3. CONCLUSIONS

JPie is a set of Java classes which implement the essential semantics of the parallelism model proposed in Milner's π -calculus. It can be implemented on top of both Grid protocols and lighter weight protocols. It allows the architecture of parallel systems to be concisely specified in the π -calculus and then implemented as a distributed processing system in Java. As such it allows the architect of a parallel system to abstract from its Grid implementation, focusing instead on its essential data flow properties.

In such a paradigm it is easy to imagine Taps and Funnels sent between a number of Tasks during the run of a parallel job. Such a setup enables not only the data to be dependent on the results of the computation but also the routing of the data between Tasks.

4. ACKNOWLEDGMENTS

We acknowledge the funding of the UK National E-Science Center in this project.

5. ADDITIONAL AUTHORS

Paul Graham (paul@epcc.ed.ac.uk, EPCC, University of Edinburgh, UK, Lewis Mackenzie[†] (lewis@dcs.gla.ac.uk)

6. REFERENCES

- [1] Apache tomcat. <http://jakarta.apache.org/tomcat/>.
- [2] Axis. <http://ws.apache.org/axis/>.
- [3] Simple object access protocol. In *W3C Notes*. <http://www.w3.org/TR/soap/>.

- [4] Webservices. In *W3C Notes*.
<http://www.w3.org/TR/ws-arch>.
- [5] Ws-resource framework: Faq.
<http://www.globus.org/wsrf/faq.asp>.
- [6] S. P. A. Giacalone, P. Mishra. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.
- [7] D. L. A. Phillips, S. Eisenbach. From process algebra to java code technical. In *Technical Report*. Imperial College London, 2002.
- [8] F. Achermann. Forms, agents and channels. defining composition abstraction with style. In *PhD Thesis*. Universitat Bern, Institut fur Informatik und Angewandte Mathematik, March 2002.
- [9] D. T. B. Pierce. Pict: A programming language based on the picalculus. In *Language and Interaction: Essays in Honour of Robin Milner, by G. Plotkin, C. Stirling, M. Tofte*, pages 455–494. MIT Press, 2000.
- [10] D. J. Berg and J. S. Fritzinger. *Advanced Techniques for Java Developers*. John Wiley & Sons, Inc, 1997.
- [11] P. Cockshott, V. Yarmolenko, and L. Mackenzie. PGP Project. In *Proceedings to 10th Global Grid Forum*, March 2004.
- [12] A. Ferrari. Jpvm: Network parallel computing in java. *ACM Workshop on Java for High Performance Network Computing*, 1998.
- [13] e. Luke Cassady-Dorion. *Industrial Strength Java*. New Riders Publishing, Indianapolis, Indiana, USA, 1997.
- [14] R. Milner. The polyadic π -calculus: A tutorial. The University of Edinburgh, 1991.
- [15] V. S. P. Gray. Icet: Distributed computing and java. *Concurrency, Practice and Experience*, 9(11):291–301, November 1997.
- [16] J. F. P. H. Welch, J. R. Aldous. CSP networking for java (JCSP.net). In *ICCS 2002*. Springer-Verlag, April 2002.
- [17] D. W. R. Milner, J. Parrow. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.
- [18] M. O. V. Cremet. Pilib: A hosted language for picalculus style concurrency. École Polytechnique Fédérale de Lausanne, Switzerland.
- [19] J.-C. N. W. P. Cockshott, S. Hoff. Experimental 3d tv studio. *IEE Proceedings - Vision Image and Signal Processing*, 2003.

APPENDIX

A. AN EXAMPLE OF REMOTE SPAWN

Below is a simple Java code which spawns a remote Task using $J\pi$. For simplicity, example has only one Tap (created from System.in) and one Funnel (created from System.out). The user-extended Task (MyTask, see Appendix (B)) reads the data from key-press and outputs the processed data back to the screen.

```
JPie j = JPieFactory.getJPie(JPieFactory.GRID);
j.setJars(array_of_filenames);

JPieTap []t={j.createTap (System.in)};
JPieFunnel []f={j.createFunnel (System.out)};

MyTask job = new MyTask ();
job.setTaps(t);
job.setFunnels(f);

if (!j.spawn (job, 12, 57, 0.3E8))
    System.out.println ("couldn't spawn the job");
```

The user gets an impression that MyTask is executed locally on the same VM.

B. A USER-EXTENDED TASK, MYTASK

Below is the Java code of a class MyTask referred to in Appendix A. The Task simply reads the data from the Tap (connected to System.in in case of Appendix A), presumably user's name and outputs a personalized greeting to Funnel (connected to System.out in case of Appendix A).

```
public class MyTask
    extends uk.ac.gla.dcs.JPie.JPieTask
{

    public void run()
    {
        byte name[] = new byte[100];
        int nr_read = 0;

        try
        {
            getFunnels()[0].write(
                "What is your name?".getBytes());
            getFunnels()[0].flush();
            nr_read = getTaps()[0].read(name);
            getFunnels()[0].write(
                "Hello ".getBytes());
            getFunnels()[0].write(name,0,nr_read);
            getFunnels()[0].flush();
        }
        catch (IOException e)
        {
            System.out.print("I'm Sorry."+
                " I didn't catch your name.");
        }
    }
}
```