JISGA: A Jini-Based Web Service-Oriented Grid Architecture

Yan Huang

Department of Computer Science
Cardiff University
PO Box 916
Cardiff CF24 3XF
United Kingdom
{Yan.Huang}@cs.cardiff.ac.uk

Abstract. This paper describes how to extend a Jini system into an OGSA-compliant infrastructure for Grid computing by introducing Web service techniques into the Jini system. The Jini-based Service-oriented Grid Architecture (JISGA) is presented, the functionality of the system is described, and the main services and components of the system are discussed. The paper shows how to present a Jini service as a Web service, and how an application composed of interacting services can be described using the service workflow description language, SWFL. The conversion of a composite application specified in an SWFL document into an executable Java code, and it subsequent execution, are also discussed. This latter capability, enabled by the *SWFL2Java* tool, permits the blocking and non-blocking submission of jobs to the Grid, and their processing in both sequential and parallel execution modes.

1 Introduction

This paper describes how Jini services can be integrated with Web services within a common Service-Oriented Architecture (SOA) for Grid computing. This common SOA permits the transparent interaction of Jini services and Web services, thereby extending the usefulness and applicability of both approaches. The main benefits of this work are: (1) Jini services can be accessed from outside of a Jini community; (2) Jini services can be invoked in the same way as any other Web service; and, (3) Jini services can be integrated with other Web services.

Web services have arisen as an essential component of the infrastructure of e-Business, and enable business-to-business transactions via the Internet. In general, these B2B transactions take place directly between computer programs, rather than between computer programs and users.

A Web service has five essential attributes [5]: it can be *described* using a standard service description language, usually Web Service Description Language (WSDL) [1]; it can be *published* to a registry of services, usually a UDDI (Universal Description, Discovery and Integration) registry; it can be *discovered* by searching the service registry; it can be *invoked*, usually remotely, through a declared API; and, it can be *composed* with other Web services.

A WSDL document describes one or more Web services, each of which is made up of multiple messages, port types, and bindings. A message gives an abstract definition of the input and output data of a Web service. A port type is used to describe the functionality of a Web service in terms of a set of operations. A binding associates a concrete protocol and data format with a port type. The Simple Object Access Protocol (SOAP) is a widely used protocol for Web service messaging, and uses an XML data encoding and HTTP transfer protocol. The binding may also specify the security mechanism for a port's communications.

The Jini networking system aims to support the rapid configuration of devices and software within a distributed computing environment [2]. These devices and software are made available to remote clients as Jini services. Jini is one of the network technologies that are suitable for building the middleware for Grid computing. A central theme of Grid computing is the sharing of resources within a virtual organization through direct transactions between computer programs [4]. This has led to the emerging concept of a Grid service and to the Open Grid Services Architecture (OGSA) which is currently being developed on the basis of Web service concepts and technologies [3]. In this approach, Grid services are regarded as specialized extensions of Web services, and support new types of problem-solving application which are composed of services. In this paper, a Jini-based Grid will be developed into a OGSA-compliant system. This is done by combining Web service and Jini service SOAs.

Jini's SOA is very similar to that used by Web services. By using Jini lookup services, Jini services can be published, discovered, and invoked and so possess three of the five essential attributes of a Web service enumerated at the start of this section. However, the other two of the five essential attributes, service description and composition, are not part of the basic Jini system. In addition, Web services use XML-based messaging which is not required (nor prohibited) by the Jini communication model.

The simplest way to combine the Web service and Jini service SOAs is by registering services using both the service registry and lookup service mechanisms. The service registry works in the world external to the Jini community, and the lookup service works within the Jini community. Thus, correspondingly, there are two sorts of publish operations, two sorts of find or discovery operations, and two sorts of bind or invoke operations.

Henceforth, a request for a Jini service through the non-Jini mechanism will be called an *indirect service request*, and a request for a service through a Jini lookup service will be called a *direct service request.* An indirect service request finds the service requested by sending a query based on the service description to the service registry and a task description based on a service flow language is formed, also based on the service description. An agent service called the *Workflow Engine* accepts the indirect service request, translates the request into a service request understandable by Jini, and makes sure the request is carried out and the results sent back to the requester, as shown in Fig. 1. These capabilities are the basis of the JIni-based Service-oriented Grid Architecture (JISGA) described in this paper.

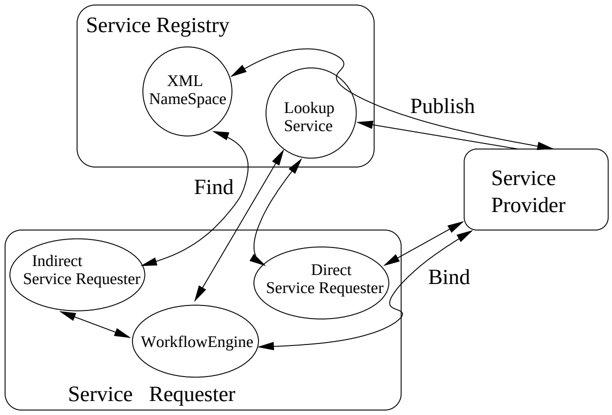


Fig. 1. A dual Service-Oriented Architecture.

2 Deploying a Jini Service as a Web Service

There are a number of differences between a Web service and a Jini service. A Jini service uses a service proxy that contains all the code needed to connect to the service. The service proxy itself is a Java object, and a copy of the proxy is uploaded to a Jini Lookup Service when the service is published. The user of the Jini service has to be a Jini client that depends on Jini Lookup mechanisms to search for the service, download the service proxy object, and access the service via its proxy object. Thus, a Jini service is only accessible in a Jini environment. However, the user of a Web service is not constrained to a particular environment. Since a Web service registers itself to a UDDI registry server and has an XML-based Web service description document to provide all the information necessary for accessing it, the user of a Web service needs to know how to search for Web services through the UDDI registry and how to access the service according to the service's XML-based description document (a WSDL[1] document). Normally, the client interacts with the Web service through SOAP[2] – an XML-based lightweight protocol for exchanging information in a decentralized, distributed environment.

The techniques and tools that we use to deploy Jini services as Web services include Apache Axis 1.0 [12], Java Web Service Develop Pack (JWSDP) 1.0.01

[1] WSDL = Web Services Description Language
[2] SOAP = Simple Object Access Protocol

[13], and *UDDI4J* [14]. Apache Axis is essentially a SOAP engine – a framework for constructing SOAP processors such as clients, servers, gateways, etc. It also includes as a stand-alone web server that plugs into servlet engines, such as Tomcat, and WSDL support tools, such as *WSDL2Java* and *Java2WSDL*. *WSDL2Java* generates Java classes from WSDL and *Java2WSDL* generates WSDL documents from Java classes. The JWSDP package contains the Tomcat servlet engine and a UDDI registry for setting up a Web service environment. *UDDI4J* is a very useful tool for the programmer in interacting with a UDDI server. In the Web service environment, a Web application server is needed to host Web services and a UDDI registry server is needed to register Web services.

Services in JISGA are activatable by using the Jini activation framework. Activation provides a way of creating remote objects that do not have to be active all the time, so a remote object is inactive until it is needed. A Jini activation system with which the remote objects are registered is in charge of activating the objects by reconstituting them and running them when a remote method call is made on them. Activation is very beneficial in saving both computing and network resources [2].

Here the Jini services we are using in the system are activatable back-end services which are running in activatable back-end processes and provide off-machine resources. Presenting a Jini service as a Web service allows the service to retain the benefits of activation and lifecycle management from the Jini world. In our implementation, the same activatable back-end process is used for the service, and an additional process is used to trigger the back-end service. The additional process is called the *Web Service Trigger* (WS Trigger) which runs a class that shares the same interface with the Jini service. *WS Trigger* works as a normal Web service, but instead of doing the work by itself it passes the service request to the real service, i.e., the back-end Jini service. Since the Jini service and its *WS Trigger* are running in different Java virtual machines, Java RMI is used for communication between them, and an RMI naming server is needed for the RMI communication. As an example, suppose we have a Jini activatable back-end service, *Vector*, that implements the following interface:

```
...
public interface VectorInterface extends Remote{
  public double dot(double[] A, double[] B)
               throws RemoteException;
  ...
}
```

Instead of registering this service only to Jini lookup services, the *Vector* service is also registered to an RMI naming server with the name *"VectorWS"* to allow RMI access by its *WS Trigger*. The *WS Trigger* class named *VectorWS* is given as follows:

```
...
public class VectorWS {
   public double[] dot(double[] A, double[] B) {
```

```
            try {
                VectorInterface backendRef = (VectorInterface)
                  Naming.lookup("rmi://demetia.cs.cf.ac.uk/VectorWS");
                return backendRef.multiply(A, B);
            } catch (Exception e) {
                return null;
            }
        }
        ...
    }
```

In the above code, *VectorWS* obtains a reference to the Jini back-end service from the Java naming server and then invokes the service by RMI. Then the *VectorWS* class is installed as a Web service by deploying it to an AXIS web application server, generating its WSDL document by using *Java2WSDL*, and registering its WSDL to a Java WSDP UDDI server with the *UDDI4J* tool.

Figure 2 illustrates the scenario in which the *Vector* service works as both a Jini and a Web service.

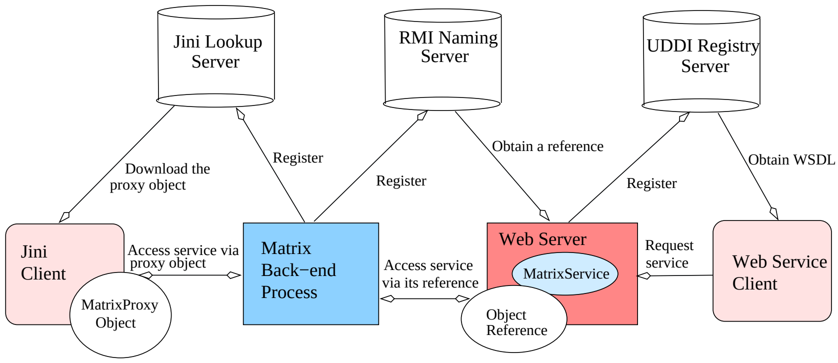


Fig. 2. The *Vector* service works as both a Jini and a Web service.

3 The Architecture of JISGA

JISGA is a Jini-based service-oriented architecture (SOA) for Grid computing. In addition to the basic functionalities of a general Jini system, it has the following features:

- It gives users easy access to services by supporting an XML-based job description language (where a job, in general, is a composition of services).
- It automatically processes the job for the user by dynamically creating and executing the job processes.

– It allows both blocking and non-blocking job submission.
– It allows both sequential and parallel job processing.

Here the job that is submitted to JISGA is not a traditional software application written in some programming language or an executable code generated by a compiler, but a service-based application described by an XML document. The XML description document is based on Service Workflow Language (SWFL) [10] and describes a job by specifying the workflow structure connecting the services of which it is composed. The automatic processing of such a job involves creating the Java code for the job, compiling the code, loading the class, executing the code, and sending the results to the job submitter.

As mentioned in Section 2, although JISGA is a Jini system, all its basic services are presented as Web services as well. The Jini lookup service and UDDI registry server coexist as the central mechanisms for publishing and discovering services. The Jini lookup service provides the link between the Jini services and users in JISGA and UDDI registry server provides the link between Jini services and non-Jini users. Several assistant services or components are used in JIGSA:

– The Visual Service Composition Environment (VSCE) provides a graphical tool for generating composite service-based applications through a "drag-and-drop" interface. This generates an application described by an SWFL document.
– The *Workflow Engine* provides an execution environment for composite applications described in SWFL. It allows both blocking and non-blocking job submission, and sequential and parallel job processing. A *JobProcessor* service mainly processes parallel jobs, which includes partitioning parallel jobs into sequential sub-jobs and then processing each sequential sub-job.
– JavaSpaces is a basic service provided by Jini, and its tuple-space is used in JISGA as a shared memory mechanism. JavaSpaces stores the job queues of the JISGA system, and also serves as an intermediary in communicating between distributed processes.
– A *JSHousekeeper* service is used to deal with certain tasks relating to the use of JavaSpaces, such as removing objects that have not been recently used, and updating the state of certain objects.
– A *ServiceManager* service is used to publish and remove services. The *ServiceBrowser* service is a support tool that allows a user to graphically and dynamically monitor the services within a JISGA system. Other supporting tools include service maintenance tools for service providers.

Figure 3 depicts the architecture of the JISGA system.

4 Service Workflow Language

In many cases it is desirable to create a job or application by composing multiple services. Such applications can be represented by a graph of interacting services that must be specified in a job description. Thus, not only a service description

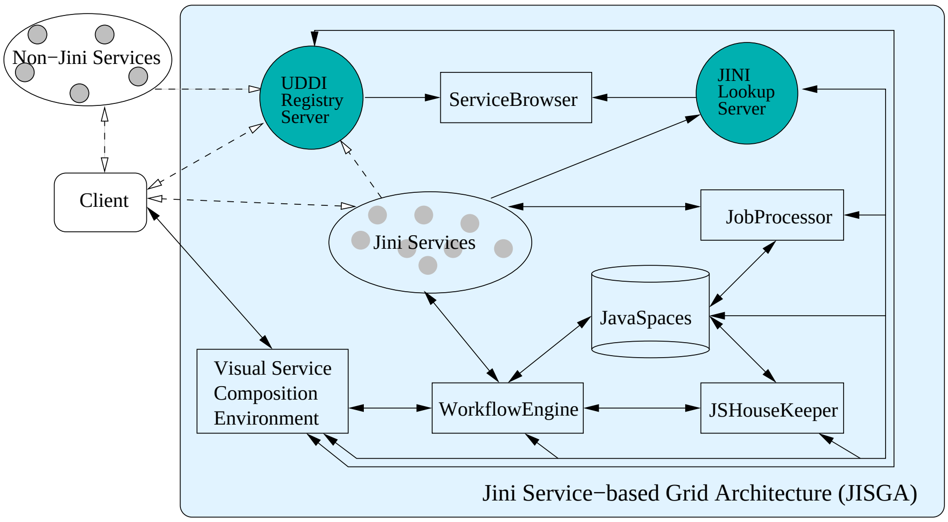


Fig. 3. JISGA: A Jini-based SOA for Grid computing

language, but also a job description language, are needed to standardize the description of both services and composite jobs[3].

In May 2001, IBM's Web Services Flow Language (WSFL) [6] and Microsoft's draft of XLANG [7] were released as languages for describing the composition of Web services. Although the intended uses of WSFL and XLANG are broadly the same, they have completely different structures: WSFL directly represents the control and data flow of an application in terms of its workflow graph; XLANG is closely based on the use of Java-like language constructs to describe a job. In August 2002, a combined version of WSFL and XLANG, called Business Process Execution Language for Web Services (BPEL4WS) [11], was published that largely inherited the programming-oriented approach of XLANG, rather than the graph-based approach of WSFL. BPEL4WS is effective for representing service-based composite applications for which the order of execution of the services is pre-defined. However, it does not have the same flexibility as WSFL, in which the only constraints on order of execution of services are implicit in the workflow graph. Having a pre-defined service execution order, as in XLANG and BPEL4WS, is not suitable for representing a distributed application, where the ability to dynamically partition and schedule services at runtime is important in order to exploit potential parallelism and to make best use of the available distributed resources. WFSL, however, does allow this capability, and hence provides a flexible and effective basis for representing a Grid application.

We have developed Service Workflow Language (SWFL) which extends WSFL in three important ways:

[3] Here a job is a composition of interacting services.

1. SWFL improves the representation of conditional and loop control constructs. Currently WSFL can handle *if-then-else*, *switch*, and *do-while* constructs and permits only one service within each conditional clause or loop body. SWFL also handles *while* and *for* loops, and permits sequences of services within conditional clauses and loop bodies.
2. SWFL permits more general data mappings than WSFL. SWFL can describe data mappings for arrays and compound objects, and also permits a activity to receive input data from more than one preceding activity.
3. SWFL provides an assignment activity which allows the assignment of one variable to another.

Further details of how SWFL provides the above capabilities are given elsewhere [10], and the complete XML schema for SWFL is given in [8]..

5 Visual Service Composition Environment

VSCE provides a visual environment for users to generate service-composite applications through a simple "drag-and-drop" interface. Icons representing services may be dragged from a virtual service repository and dropped onto a canvas, where they can be connected with other services through data and control links. VSCE is described in more detail in [**?**]. The implementation of VSCE is based on the graphical representation of workflow inherent in SWFL. Since SWFL describes composite applications in terms of their workflow graph structure, it is easy to transform an application between its graph structure and the corresponding SWFL document.

Figure 4 lists all the mappings between graphical symbols and concepts in SWFL. A normal activity is one that is supported in WSFL and is represented as a square box. It can have zero or more data input ports and at most one data output port, which correspond to the input parameters and return value of an object method in Java. In the control and loop processes, each control or loop activity is represented as a diamond box with a number of input data ports and the same number of output data ports, one input control port and two or more output control ports. Assignment activities have the same graphical symbol as normal activities. In addition, a small square box attached to the top of an activity box stands for an input port, and a small circle attached to the bottom of an activity box stands for an output port. Control ports are shown in black and data ports in white. A control link is represented by a solid line with a black arrow and a data link is represented by an dashed line with a white arrow.

As a simple example, Fig. 5 shows an application's graphical workflow model created using VSCE.

In the example, *A*, *B*, *C*, *D*, and *E* are activities. Based on the mapping rules, the graphical flow model corresponds to the following SWFL description:

```
<FlowModel ...>
```

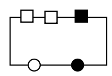
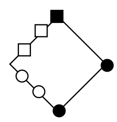
Graphical Symbol	SWFL Concept
□	Data input port
○	Data output port
■	Control input port
●	Control output port
- - ▷	Data link
——▶	Control link
	Normal and assign activity
	Control activity

Fig. 4. Mapping between graphical symbols and SWFL concepts.

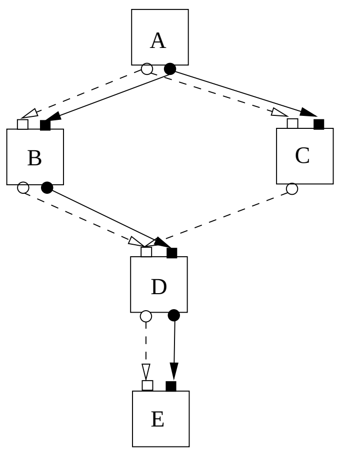


Fig. 5. An example of a graphical workflow model.

```
  <ControlLink name=AtoB source=A target=B/>
  <ControlLink name=AtoC source=A target=C/>
  <ControlLink name=BtoD source=B target=D/>
  <ControlLink name=DtoE source=D target=E/>

  <DataLink name=AtoB source=A target=B>..</DataLink>
  <DataLink name=AtoC source=A target=C>..</DataLink>
  <DataLink name=BtoD source=B target=D>..</DataLink>
  <DataLink name=CtoD source=C target=D>..</DataLink>
  <DataLink name=DtoE source=D target=E>..</DataLink>
</FlowModel>
```

6 SWFL2Java

A job composed of interacting services can be represented by the SWFL job description language [10]. The main motivations for developing SWFL were to describe Java-oriented conditional and loop constructs, to permit sequences of more than one service within conditional clauses and loop bodies, and to overcome limitations inherent in WSFL's data mapping approach. In addition, we have developed: (1) a tool called `SWFL2Java` that converts the description of a composite job in SWFL into executable Java code; and, (2) a Workflow Engine that provides an execution environment to run the composite job.

The details of `SWFL2Java` are discussed elsewhere [8], however, an overview of the implementation will be given here. In `SWFL2Java` an SWFL document is not translated directly into a Java program but is stored in an intermediate form as a Java *FlowModel* object. This is made up of two Java *Graph* objects: *DataGraph* and *ControlGraph*. The former stores the data flow structure of the flow model, and the latter stores its control flow structure. One reason for storing the job description in this intermediate form is to be able to interact readily with Java-based tools, such as the Visual Service Composition Environment (VSCE), for the visual composition of Web services [**?**]. In VSCE, a composite job is represented as a graph in which activities (representing services) correspond to nodes, and data links and control links correspond to different kinds of directed edge. The graph can be stored as a *FlowModel* object and converted to and from SWFL, as well as into a Java program. Another reason for using the *FlowModel* form is to reduce the overhead when the same job is used many times and scheduled on different resources. In such cases it is easier to generate the Java code from the intermediate form rather than starting from the original SWFL. Figure 6 illustrates the role of the *FlowModel* object.

Given a graphical *FlowModel* of arbitrary structure to be transformed into a Java Jini-based distributed program, the following three issues have to be addressed: how to find all the services; how to decide the order of execution based on the flow model; and, how to automatically name all the variables, class names, and methods.

Normally the first thing a Jini-based distributed program does is to discover the services that are going to be used in the program. Thus, to automatically

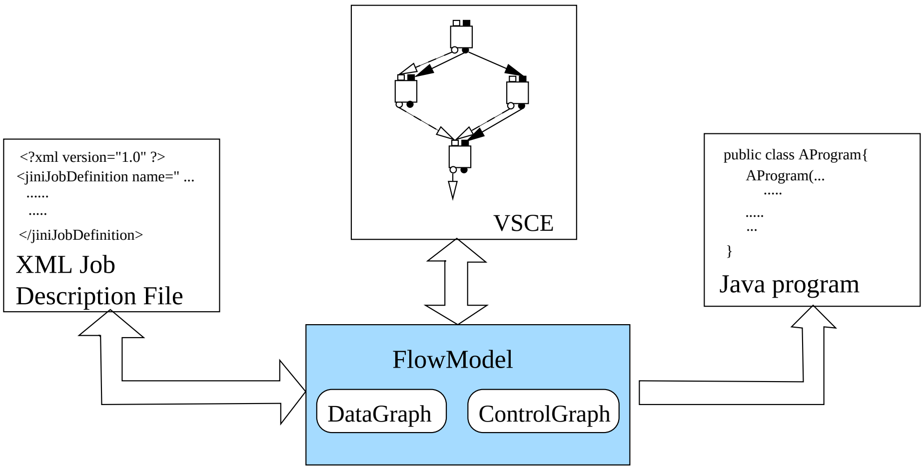


Fig. 6. Using a *FlowModel* as an intermediate form in `SWFL2Java`

create a Java Jini-based program from a graphical flow model, service discovery is the first issue to be addressed. Service discovery is performed by an assistant class called *ServiceFinder*. The *ServiceFinder* thread takes a list of services and discovers each using a *ServiceDiscoveryManager*. Once a service is discovered, its *ServiceItem* proxy is downloaded and stored.

After all the service proxies needed in the program have been found and stored, the main program is generated. The problem here is determining the order in which tasks are to be processed by services. This order is deduced from the control graph using the Task Processing Sequential Order Generation algorithm described in [8].

In automatically generating a Java program several naming issues need to be addressed. In the automatically produced program, the class name takes the form *FlowModel_XXX* where *XXX* is the name of the flow model. The class has a constructor and an *execute()* method that runs the job specified in the flow model. The parameters of the *execute()* method are determined by the input, or *flowSource*, of the flow model. The parameter names are the same as the corresponding *part* names of the *flowSource* message. The return variable of an activity takes the form *XXX_return* in which *XXX* is the name of the activity.

6.1 An Example of the Use of SWFL2Java

An example will now be given of the conversion by `SWFL2Java`of a workflow graph generated with VSCE into a service-based Java program. In this example the elements of a vector, B, are assigned values within a *for* loop by a service that computes some function of the loop control variable, $in0$. On exiting the *for* loop the scalar product of B with itself is evaluated by another service. The workflow diagram for these operations is shown in Fig. 7.

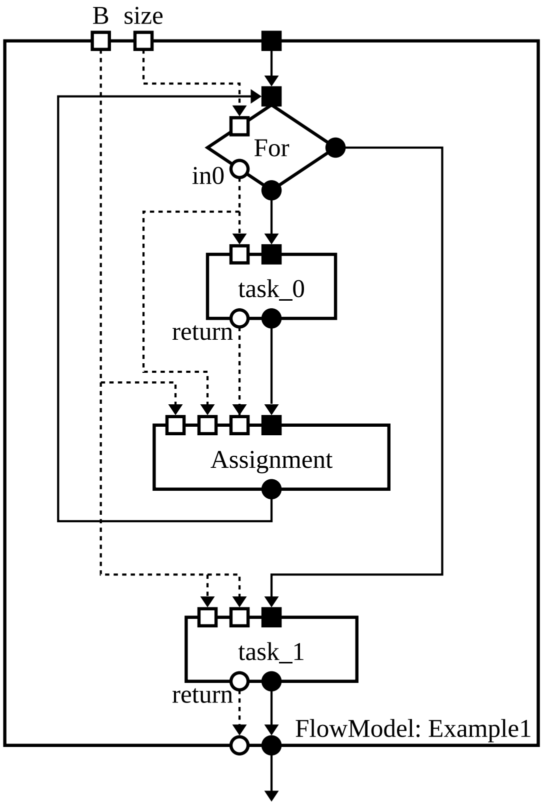


Fig. 7. An example workflow.

In Fig. 7, one *for* activity, one assignment activity, and two normal activities (task_0 and task_1) are depicted. The inputs to the workflow are the vector, B, and the integer, $size$. The task_0 and assignment activities are both within the *for* loop, as may be seen from the fact that control flows out of the assignment activity and back into the *for* activity. On completion of the *for* loop control flows out of the loop to the task_1 activity. The output of the workflow is the output of task_1. The task_0 activity evaluates a function of the loop control variable $in0$, and the assignment activity then assigns this value to $B[in0]$. On exit for the *for*loop task_1 evaluates the scalar product of B with itself.

The Java program, corresponding to the workflow shown in Fig. 7, that is automatically created by `SWFL2Java` is as follows.

```
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceTemplate;
import java.util.HashMap;
import SWFL2Java.XML2Graph.ServiceFinder;

public class FlowModel_Example1{
  private HashMap taskTemplateMap = null;
  private ServiceFinder serviceFinder = null;

  public FlowModel_Example1(){
```

```
    this.taskTemplateMap = new HashMap();
    this.setTaskTemplateMap();
    this.serviceFinder = new ServiceFinder(this.taskTemplateMap);
    Thread thread = new Thread(this.serviceFinder);
    thread.start();
    try{
      thread.join(2*60*1000);
    } catch (java.lang.InterruptedException e){
        System.out.println("Failed to find services.");
        System.exit(-1);
    }
  }

  private void setTaskTemplateMap(){
    this.taskTemplateMap.put("task_1", new ServiceTemplate(null,
         new Class[]{services.Vector.VectorInterface.class}, null));
    this.taskTemplateMap.put("task_0", new ServiceTemplate(null,
         new Class[]{services.Math.MathInterface.class}, null));
  }

  public double[] execute(int size, double[] B)
                    throws java.rmi.RemoteException{
    for(int in0=0; in0<size; in0++){
      services.Math.MathInterface task_0 = (services.Math.MathInterface)
                (this.serviceFinder.getServiceItem("task_0")[0].service);
      double task_0_return = task_0.function(in0);
      B[in0] = task_0_return;
    }
    services.Vector.VectorInterface task_1 = (services.Vector.VectorInterface)
                (this.serviceFinder.getServiceItem("task_1")[0].service);
    double task_1_return = task_1.dot(B,B);
    return task_1_return;
  }
}
```

In JISGA, the SWFL document describing the application that has been generated by VSCE from the graphical workflow model is submitted to a *WorkflowEngine* service as described in the next section. The *WorkflowEngine* service then uses the `SWFL2Java` tool to convert the SWFL document into Java code, and then compiles, loads, executes it. In the above code, the service templates for the services to be used as put into a *HashMap*. The interfaces for these services have previously been found within VSCE from the services' WSDL, and are used in generating the service templates in the *setTaskTemplateMap()* method. A *ServiceFinder* service is then used to locate the services to be used. The *execute()* method contains a *for* loop in iteration $in0$ of which the *function()* operation of the task_0 service instance is invoked with input $in0$. This corresponds to the activity labelled "task_0" in Fig. 7. Element $in0$ of the vector B is then assigned the value returned by *function()*. This corresponds to the activity labelled "Assignment" in Fig. 7. On exiting from the *for* loop the *dot()* operation of the task_1 service instance is invoked. This corresponds to the activity labelled "task_1" in Fig. 7. The value returned by the *dot()* operation is returned by the *execute()* method. The *WorkflowEngine* service would then return this value to whoever submitted the SWFL document.

7 Implementation Aspects of JISGA

The role of the *WorkflowEngine* service is to receive an SWFL document describing an application that has been generated by VSCE from a graphical workflow model, and ensure that the service-oriented tasks encoded in the SWFL document are performed, and any results are returned to the job submitter. The *WorkflowEngine* service supports both blocking and non-blocking job submission, and sequential and parallel job processing. In a blocking, sequential job submission the *WorkflowEngine* processes the job by generating the Java harness code with the `SWFL2Java` tool, and then executes it and sends the result to the job submitter. Non-blocking and parallel jobs are handled somewhat differently, as described below.

In non-blocking and parallel jobs, JavaSpaces plays a key role in the job processing. JavaSpaces provides object storage services through a tuple-space abstraction, and supports the sharing of objects between distributed applications. In JISGA, JavaSpaces is used in the following ways:

1. As a shared memory, allowing message passing between different services and processes.
2. To store the *result* objects of non-blocking jobs so that job submitters can retrieve their job results later.
3. To store *message* objects used in communication between sub-job processes, thereby allowing job parallel processing.
4. In maintaining the two job queues of the JISGA system so that jobs and sub-jobs can be processed in a FIFO (First-In-First-Out) order.

JISGA makes use of two job queues. One is the Parallel Job Queue, containing parallel jobs to be processed. The other is the Sub-Job Queue, containing sub-jobs generated from the partitioning of parallel jobs. The *JSHousekeeper* performs housekeeping tasks for JavaSpaces serving in a JISGA system. Its duty is to collect *result* objects, update job queues, and maintain data messages. It is also a service, and provides an interface for job submitters to retrieve their job results.

The *WorkflowEngine* and *JobProcessor* services both deal with the processing of jobs. The *WorkflowEngine* service allows clients to submit their jobs, which are described in an XML-based workflow language such as SWFL. Based on the required functionality of JISGA, the *WorkflowEngine* service provides four methods for the four different ways of submitting a job: blocking submission of a sequential job, blocking submission of a parallel job, non-blocking submission of a sequential job, and non-blocking submission of a parallel job. If a job is submitted to a *WorkflowEngine* service as a sequential job, then the

WorkflowEngine will process the job itself. However, if the job requires parallel processing, the *WorkflowEngine* service will leave the job for the *JobProcessor* service to process by adding the job to the Parallel Job Queue. A *JobProcessor* is a service that works internally in a JISGA system. Its main duty is to process parallel jobs, which includes partitioning a parallel job into sub-jobs, adding these sub-jobs to the Sub-Job Queue, and processing sub-jobs in the Sub-Job Queue. There are two threads running synchronously in a *JobProcessor*. One of the threads takes one job at a time from the Parallel Job Queue and then partitions the job into sub-jobs which are then added to the Sub-Job Queue. Another thread takes one sub-job a time from the Sub-Job Queue and processes it. A JISGA system can have as many *WorkflowEngine* and *JobProcessor* services as deemed necessary – it is up to the JISGA administrator to decide on the number of *WorkflowEngine* and *JobProcessor* services. However, more than one *JobProcessor* service is needed to allow the JISGA system to support parallel job processing.

A JISGA system can be pictured as a virtual computer with multiple processors that is composed of multiple distributed machines connected by a network. JavaSpaces works as the memory and *JSHousekeeper* works as a kind of memory management unit. In this virtual machine each *JobProcessor* works like a CPU. *WorkflowEngine* services provide the interactive interface for the clients. Figure 8 depicts such a virtual computer.

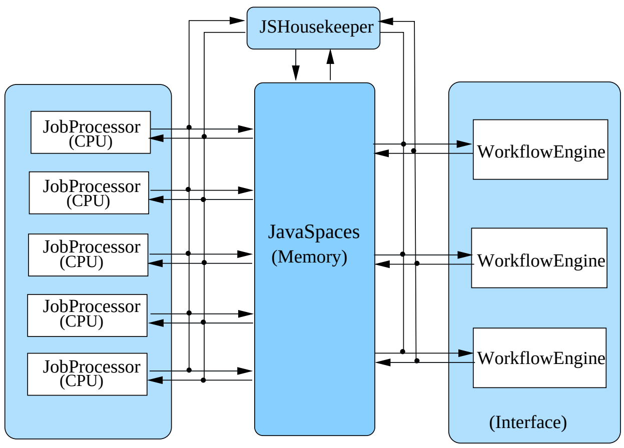


Fig. 8. JISGA works as a virtual computer with multiple processors.

8 Conclusions

This paper has presented JISGA, a Jini-based Web service oriented system for Grid computing. Several technical issues involved in developing a Jini system into an OGSA-compliant system based on Web service concepts have been focused on. These include the deployment of a Jini service as a Web service, the visual composition of service-based applications based on their workflow, the conversion of a workflow diagram into an SWFL document, tools and techniques for converting this document into an executable code, and an execution environment for running the resulting distributed application. This execution environment is based on the `SWFL2Jave` tool, and the *WorkflowEngine* and *JobProcessor* services.

Deploying a Jini service as a Web service is done by adding a Web service trigger to each Jini service. The Web service trigger works as a Web service on behalf of the Jini service by passing any service request to the back-end Jini service. RMI is used for the communication between a Web service trigger and its Jini service.

SWFL, an extension to WSFL, is an XML-based description language for describing composite service-based Grid applications. Given an SWFL job description, the `SWFL2Java` tool can generate a representation of the corresponding data and control link structure in the form of a Java *FlowModel* object. From this the corresponding Java program can be automatically generated.

The paper also presents aspects of the JISGA implementation. A workflow engine provides an execution environment for applications described by SWFL documents. By using `SWFL2Java`, it can dynamically generate code based on an SWFL document and run it. The *WorkflowEngine* service accepts both blocking and non-blocking jobs and provides for sequential and parallel job processing. In parallel job processing, JavaSpaces is used as a mechanism for message passing between distributed processes. It is also used to store job queues for the system. *JobProcessor* services are used to partition parallel jobs into sub-jobs and subsequently process them.

References

1. Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana, "Web Services Description Language (WSDL) 1.1," available as a W3C note at http://www.w3c.org/TR/wsdl/, March 2001.
2. W. K. Edwards and T. Rodden, "Jini Example By Example," published by Prentice Hall, 2001.
3. Ian Foster, Carl Kesselman, Jeffrey Nick, and Steven Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," January, 2002. This is available online at http://www.globus.org/research/papers/ogsa.pdf.
4. Ian Foster, Carl Kesselman, and Steven Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," The International Journal of High Performance Computing Applications, Vol. 15, No. 3, pages 200-222, Fall 2001.

5. K. Gottschalk, S. Graham, H. Kreger, and J. Snell, “Introduction to Web Services Architecture,” IBM Systems Journal, Vol. 41, No. 2, 2002.
6. F. Leymann, “Web Services Flow Lauguage (WSFL 1.0),” available at http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, May 2001.
7. S. Thatte, “XLANG: Web Services for Business Process Design.” Available at http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001.
8. Y. Huang, “The Role of Jini in a Service-Oriented Architecture for Grid Computing,” PhD thesis, Department of Computer Science, Cardiff University, April 2003.
9. Y. Huang, “GSiB: PSE Infrastructure for Dynamic Service-Oriented Grid Application” in proceedings of the Workshop on Complex Problem-Solving Environments for Grid Computing Workshop, held at ICCS 2003 in Melbourne, Australia, 1 June 2003.
10. Y. Huang and D. Walker, “Extensions to Web Service Techniques for Integrating Jini into a Service-Oriented Architecture for the Grid,” in proceedings of the International Conference on Computational Science 2003, held in Melbourne, Australia, 2-4 June 2003.
11. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, S. Weerawarana “Business Process Execution Language for Web Services”, available at http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, August 2002.
12. “AXIS,” available at http://xml.apache.org/axis/, 2002
13. “Java Web Services Developer Pack,” available at http://java.sun.com/webservices/webservicespack.html, 2002
14. “UDDI4J,” avalable at http://www-124.ibm.com/developerworks/oss/uddi4j/, 2001