Proc UK e-Science All Hands Meeting 2003, © EPSRC Sept 2003, ISBN 1-904425-11-9

Implementations of a Service-Oriented Architecture on top of Jini, JXTA and OGSA

Nathalie Furmento Jeffrey Hau William Lee Steven Newhouse
John Darlington

London e-Science Centre, Imperial College London, South Kensington Campus, London SW7 2AZ, UK
Email: lesc-staff@doc.ic.ac.uk

Abstract

That paper presents the design of an implementation independent Service-Oriented Architecture (SOA) that is the main basis of the ICENI grid middleware. Three implementations of this architecture have been provided on top of Jini, JXTA, and OGSA. The main goal of this paper is to discuss these different implementations and provide an analysis of their advantages and disadvantages.
Keywords: Service-Oriented Architecture, Grid Middleware, Jini, JXTA, OGSA.

1 Introduction

Service-oriented architectures are widely used in the Grid Community. These architectures provide the ability to register, discover, and use services, where the architecture is dynamic in nature. From all the initiatives to define standards for the Grid, a consensus seems to emerge towards the utilisation of such an architecture, as we can for example see with the OGSA initiative of the Global Grid Forum [6].

The ICENI Grid Middleware [7, 8] is based on a service-oriented architecture (SOA) as well as on a component programming model. The goal of that paper is to show how the SOA has been designed to be implementation dependant. That gives us an open model where different low-level libraries can be plugged in.

The following of the paper is organised as follows. Section 2 shows in details the design of the Service-Oriented Architecture, the different implementations are explained and discussed in Section 3, before concluding in Section 4.

2 Design of the ICENI's SOA

A typical Service-Oriented Architecture is presented in Figure 1. One can see three fundamental aspects of such an architecture:

1. **Advertising**. The Service Provider makes the service available on the Service Broker.
2. **Discovery**. The Service Consumer finds a specific Service using the Service Broker.
3. **Interaction**. The Service Consumer and the Service Provider interact.

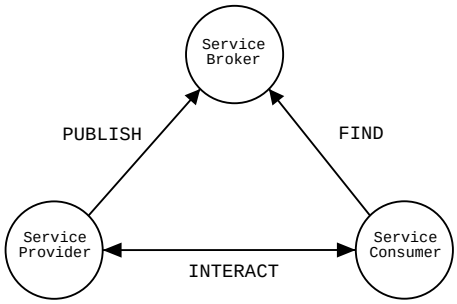


Figure 1: The Service-Oriented Architecture.

In the context of ICENI, a Service Broker is represented by a public Computational Community or *Virtual Organisation*, where end-users – the Service Consumers – can connect through their X.509 certificates to query and access services. Once a service is advertised and discovered, any interaction with it is controlled by the service level agreement (SLA) that is going to define the entities allowed or not to access the service, as well as the interval time the access is allowed or denied.

The lifetime of an ICENI service can be resumed by the three following steps: creation, advertising, discovery. Each of these steps is represented in ICENI by a set of interfaces. We are now going to explain these different steps and demonstrate them through a basic Counter Service.

2.1 Creation

A service is defined by its interface, i.e. the list of methods it provides. For example, a counter service providing basic functionalities to add and subtract a value can be defined as shown in Appendix A. The implementation of the counter is shown in Appendix B.

It is important at that level to notice there is no information on how the service is going to be implemented. We will see in the following sections how this abstract ICENI service is going to be implemented by using for example the Jini library.

The instantiation of the service is done through a call to the `IceniServiceMetaFactory`, that will first instantiate a `IceniServiceFactory` for the used implementation, and asks this factory to return a new instance of the service. At that point, all the necessary classes to implement the abstract ICENI service will be automatically generated. Calling for example the following line of code will create an ICENI service of the type `PrivateCounter`.

```
IceniService xServ = IceniServiceMeta-
Factory.newInstance("PrivateCounter");
```

2.2 Advertising

Once created, a service can be advertised on a specific domain or virtual organisation through the `IceniServiceAdvertizingManager` service. The service is advertised with a SLA that defines the access policy to interact with the service. The same service can be advertised in different organisations with different SLA's. That gives a flexible mechanism to control how different organisations will access the service, by allowing to advertise the service capabilities as required.

The advertising of a service is done through a XML document that defines the SLA's of the service for all the virtual organisations where the service is to be made available. Appendix C shows a SLA XML document that gives access from Monday to Friday noon to all the persons connecting from the virtual organisation *public1* and belonging to the organisation *eScience*.

2.3 Discovery

By connecting to a virtual organisation, a service consumer can query a service and interact with it once discovered. The query can be very simple as defining a service type, or defines a service to match using service data. The different steps to discover services are shown in Figure 2.

2.4 Invocation

Any interaction with the service is controlled by an external entity, that will first authenticate the service consumer through its X.509 certificate and authorise it against the policy of the service it wishes to access.

3 Implementation of the ICENI's SOA

Figure 3 shows the ICENI architecture. One can see the three implementations of the different service API's. In the following of this section, we will present for each of them how the different aspects of the ICENI's SOA have been implemented, and we will also give the advantages and disadvantages of this specific implementation.

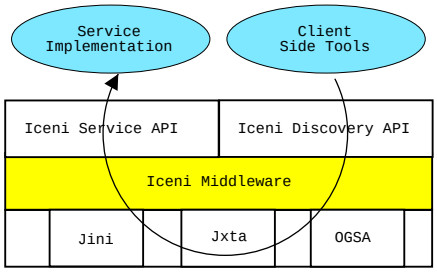


Figure 3: The ICENI Middleware Architecture.

3.1 Implementation Using Jini

The first version of the ICENI Grid Middleware had been directly implemented on top of the Jini API [3]. The rewriting of ICENI was mainly achieved to allow services to be independent of the underlying remote communication library. As we said before, the interface and implementation of an ICENI service are completely unaware of that remote communication library. When using Jini, the different classes will be automatically generated for a service named `MyService`.

1. Instantiate a discovery manager.

```
IceniServiceDiscoveryManager xDiscovery = IceniServiceDiscoveryManagerFactory.-
newInstance();
xDiscovery.setLocation("location");
```

2. Instantiate a discovery query. Here we use the instantiation mechanism based on the interface of the service to listen to.

```
IceniServiceDiscoveryQuery xQueryCounter = IceniServiceDiscoveryQueryFactory.-
newInstance(PrivateCounterService.class);
```

3. Register a listener. For every new service matching the query, the `servicePublished()` method will be called with the service as a parameter. Similarly, the `serviceUnpublished()` method will be called for each service disappearing from the lookup service.

```
xDiscovery.registerListener(xQueryCounter, new IceniServiceDiscoveryListener() {
  public void servicePublished(IceniService pService) {
   // code to execute when a new service is available
  } // end servicePublished
  public void serviceUnpublished(IceniServiceId pServiceId) {
   // code to execute when a service is no longer available
  } // end serviceUnpublished
});
```

Figure 2: Discovery and Interaction with ICENI Services.

- **MyServiceJiniNoAbstract.java** extends the implementation of the service `MyService` to provide an implementation for all the basic ICENI/Jini mechanisms.
- **MyServiceJiniStub.java** is the main Jini interface extending the interface `java.rmi.Remote`. It will act as a proxy for `MyService` service, and defines exactly the same methods.
- **MyServiceJiniStubImpl.java** is the implementation of the interface **MyServiceJiniStub**. It uses a reference to **MyServiceJiniNoAbstract** to execute all the method calls on the service.
- **MyServiceJini.java** implements the interface `MyService` by using a reference to **MyServiceJiniStub** to redirect an ICENI service's method call as a Jini service's method call.

Figure 4a) shows an interaction diagram of these different classes and interfaces.

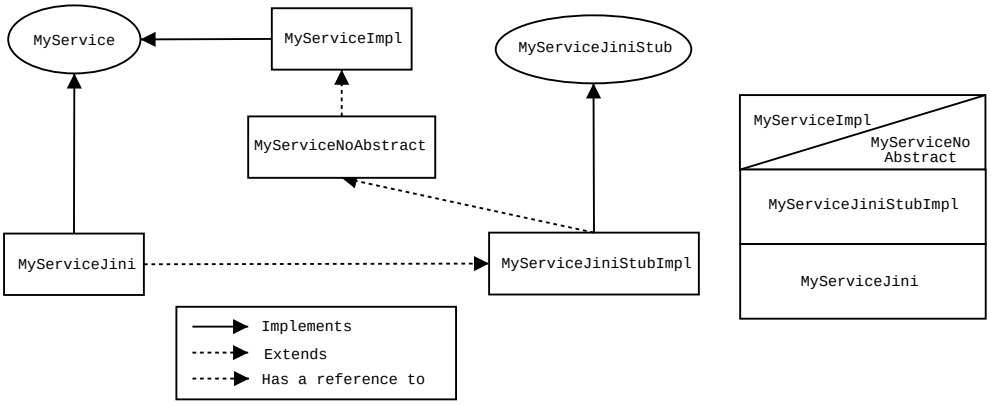
Creation. That step creates an object of the class `MyServiceJini` and initialises it with the corresponding stub, i.e. an instance of the class `MyServiceJiniStubImpl`. We obtain an object as shown in Figure 4b).

Advertising. The object `MyServiceJiniStubImpl` — hold by the ICENI service created in the previous step — extends indirectly the interface `java.rmi.Remote`, it can therefore be made available in a Jini lookup service.

Discovery. The object returned from the Jini lookup service is a `MyServiceJiniStubImpl`. It is going to be wrapped in an instance of the class `MyServiceJini` before being returned to the listener. We obtain here a similar object to the one obtained when creating the service.

Invocation. Any method call will be done on an instance of the class `MyServiceJini` and will finally be redirected on an instance of the class `MyServiceImpl` as one can see in Figure 4.

Advantages / Disadvantages. The functionalities provided by the SOA of ICENI and the Jini library are basically the same. It was therefore very easy to implement the SOA on top of Jini without however tying up ICENI to Jini and get an implementation dependant SOA. Moreover, the Jini



a) Class and interface diagram. b) Instantiation diagram.

Figure 4: Jini Implementation of an Iceni Service.

implementation is very scalable, some experiments have been done inside the group to test the performance of Jini when increasing largely the number of Jini services. These experiments showed a good result in the performance when discovering and accessing the Jini services. The potential problems when using Jini lie in security and in the connection of services across firewalls.

3.2 Implementation Using JXTA

Project JXTA[4] provides a set of XML based protocols for establishing a virtual network overlay on top of current existing Internet and non-IP based networks. This standard set of common protocols defines the minimum network semantics for peers to join and form JXTA peergroups - a virtual network. Project JXTA enables application programmers to design network topology to best match their requirement. This ease of dynamically creating and transforming overlay network topology allows the deployment of virtual organisation.

The fundamental concept of the ICENI JXTA implementation is the ICENI peergroup. The ICENI peergroup provides a virtual ICENI space that all ICENI JXTA services join. The peergroup contains the core ICENI services - `IceniServiceDiscoveryManager` and `IceniServiceAdvertizingManager`. These two services allow any services in the ICENI group to advertise its presence or to discover other services using ICENI ServiceData embodied in JXTA advertisements. Figure 5 presents an overview on how ICENI services behave when implemented on top of JXTA.

Creation. The creation of an ICENI service is just a matter of opening two separate JXTA pipes. Pipes are the standard communication channels in JXTA, they allow peers to receive and send messages. One of the two required pipes is a listening pipe that will listen for the control message broadcast to the whole ICENI peergroup. The other is the service's private ServicePipe. ServicePipes provides the communication channel for invocation messages. Depending on service functionality and requirement, these pipes could have varied properties such as encryption, single/dual-direction, propagation, streaming, ...

Advertising. Once join the ICENI peergroup, a service can advertise its presence by publishing its `ServiceData`. This is a two step process:

1. Create a new `IceniServiceAdvertisement`. This is a custom advertisement that contains the ICENI service identifier `ICENIServiceID` and the service data `ServiceData`. Service ID can be automatically generated during advertisement creation and `ServiceData` will be converted into XML format and embedded into the advertisement.

2. Publish the advertisement by using the `IceniServiceAdvertizingManager` service from the ICENI peergroup.

Discovery. Peers in the ICENI peergroup can discover available services by using the `IceniServiceDiscoveryManager` service.

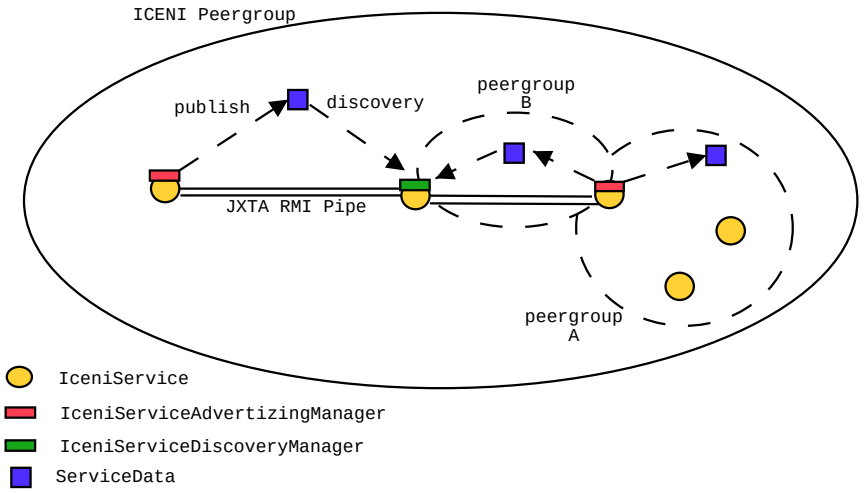


Figure 5: JXTA Implementation of the SOA.

Search can be conducted using service ID or service data elements.

Invocation. Invocation behaviour of ICENI JXTA services depends on the specific protocol each service is running. There are currently some projects working on providing different service architecture over JXTA such as JXTA-rmi [11] and JXTA-soap [12]. These projects wrap the original invocation messages (such as SOAP) into JXTA pipe messages and transport them through JXTA pipes to enable peers to invocate services using well-known service invocation API.

Advantages / Disadvantages. JXTA provides an architecture that gives middleware programmers the flexibility and ease of creating virtual organisations. It also provides an easy interface for publishing and discovering data in a peer to peer manner. The potential problems with using JXTA as an architecture for building Grid middleware lies in security and performance. JXTA's P2P nature makes it harder to secure than traditional platforms. Also it will be difficult for the requirements of high performance grid application to be met by JXTA's current XML based messaging protocols.

3.3 Implementation Using OGSA

The Open Grid Services Infrastructure is an effort to build on the wide adoption of web services as an inter-operable foundation for distributed computing. The Grid Services Specification [13] describes a set of core port types using WSDL that are essential for the Grid setting. In ICENI, important notions of an `IceniService` are mapped to the relevant constructs in the `GridService` port type, such as meta-data as service data, and lease as termination time. Our implementation is based on the Globus Toolkit 3.0 [1] core distribution. It is the Java reference implementation of the Grid Services Specification. It allows Java objects to be deployed as OGSI services. The hosting environment acts as a SOAP processing engine that can be executed as an embedded HTTP server or operate as a Java Servlet inside a servlet engine. We have enhanced the implementation with an Application Programming Interface (API) for runtime dynamic deployment of service without the use of deployment descriptor. It serves as the kernel for the ICENI OGSA implementation.

Creation. To transparently transform an `IceniService` object into an OGSI-compliant service, the runtime system reflectively interrogate the class information of the service object and generate adapted classes that can be deployed through the deployment API. Adaptation is performed using the ASM byte-code generation library [5]. The adapted class is loaded from the byte stream into the running virtual machine using a specialised `ClassLoader`. The adapted object represents a service object that conforms to the requirement of GT3, such as an extension to the `GridServiceBase` interface. The adapted class acts solely as the delegate hosted by GT3 and directs invocation to the service object.

Advertising and Discovery. OGSI currently does not mandate a particular form of advertising

and discovery mechanisms. We have chosen to use an instance of the `ServiceGroup` port type as a representation of a community. A `ServiceGroup` service is set up at a well-known location. When an `IceniService` is created, the Grid Service Handle of the OGSI-service representing this service object is published to the known `ServiceGroup`. Future implementations can experiment with using UDDI directory for long-lived services, such as Factory or Virtual Organisation Registry. For transient services, the Globus Toolkit 3.0 Index Service [2] can cater for the dynamic of temporal validity of service and its meta-data. Also, it provides a rich query mechanism for locating service instances based on their service data and port types.

Invocation. When a client locates an `IceniService` from the `IceniServiceDiscoveryManager`, the OGSA implementation will return a Java Reflection Proxy implementing the interfaces expected by the client. The proxy traps all invocations on the object. The invocation handler uses the JAX-RPC [10] API to marshal the parameters into SOAP message parts based on the WSDL description of the service.

Advantages / Disadvantages. The OGSI-compliant implementation allows ICENI services and clients to communicate through an open transport and messaging layers instead of the proprietary RMI protocol used by Jini. Also, non-ICENI clients can interact with ICENI services as if they are OGSI-compliant services. The extensible nature of OGSI permits different transport and messaging protocols to be interchanged. Our current implementation uses the web service security standards for encrypting message as well as ensuring authenticity of the caller. One disadvantage of the current invocation model is that ICENI clients can only transparently invoke OGSI services that originate from an ICENI Service. This is due to the fact that the Java interface of the OGSI service is pre-established before the conversation. For ICENI client to invoke an external OGSI service, stubs need to be generated at compile-time, or the Dynamic Invocation Interface of the JAX-RPC API could be used instead.

3.4 Discussion

The three implementations we have presented all provide the basic functionalities needed by the ICENI Service-Oriented Architecture. However, one of the main concerns in grid middleware is that security should be present at any level of the infrastructure. We need to provide basic security for remote calls such as mutual authentication, authorisation and integrity. We also need to know that the code downloaded across a network can be trusted. The SOA of ICENI provides an authentication and authorisation model that allows to check the access to its services, but that model needs to be extended into a full security model in order to be used in any production Grid.

We believe that these concerns can be dealt with by using Jini 2.0 [9]. That new version of the Jini Network Technology provides a comprehensive security model which one of the main goals is to support pluggable invocation layer behaviour and pluggable transport provider. We could therefore use OGSI instead of RMI as a remote communication layer, and benefit of the encryption and authentication features of the web service security standard.

We are also thinking of developing a protocol layer that will allow our three implementations to inter-operate and be able of getting a virtual organisation composed for example of ICENI/Jini services and ICENI/JXTA services. That will allow us the following configuration: use Jini inside a local organisation, and use JXTA to cross boundaries between networks potentially configured with firewalls.

4 Conclusion

We have shown in that paper the design of a Service-Oriented Architecture for a Grid Middleware that is implementation-dependant. This Service-Oriented Architecture has been successfully implemented on top of Jini, the ICENI services also being exposed as Grid Services through an OGSA Gateway [7]. We are currently prototyping the JXTA and the OGSA implementations of the SOA.

These three implementations do not answer all the requirements our SOA has, we are planning to work on a new implementation that will provide a full security model by using characteristics of our existing implementations.

References

[1] The Globus Toolkit 3.0. `http://www-unix.globus.org/toolkit/download.html`.

[2] GT3 Index Service Overview. http://www.globus.org/ogsa/releases/final/docs/infosvcs/indexsvc_overview.html.

[3] Jini Network Technology. http://wwws.sun.com/software/jini/.

[4] Project JXTA. http://www.jxta.org/.

[5] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A Code Manipulation Tool to Implement Adaptable Systems. In *Adaptable and Extensible Component Systems*, Grenoble, France, November 2002.

[6] Global Grid Forum. http://www.gridforum.org/.

[7] N. Furmento, W. Lee, A. Mayer, S. Newhouse, and J. Darlington. ICENI: An Open Grid Service Architecture Implemented with Jini. In *SuperComputing 2002, Baltimore*, Baltimore, USA, November 2002.

[8] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Journal of Parallel Computing*, 28(12):1753–1772, 2002.

[9] Jini Network Technology. Jini technoloy starter kit overview v2.0. http://developer.java.sun.com/developer/products/jini/arch2_0.html.

[10] Sun Microsystems. Java API for XML-Based RPC 1.1 Specification. http://java.sun.com/xml/jaxrpc/index.html.

[11] JXTA Project. Jxta-rmi Project. http://jxta-rmi.jxta.org/servlets/ProjectHome.

[12] JXTA Project. Jxta-soap Project. http://soap.jxta.org/servlets/ProjectHome.

[13] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, D. Snelling, and P. Vanderbilt. Open Grid Service Infrastructure (OGSI) v.1.0 Specification, Feburary 2003.

A Interface for a Counter Service

```
import org.icenigrid.runtime.resources.PrivateResourceService;
import org.icenigrid.services.basic.IceniServiceException;

public interface PrivateCounterService extends PrivateResourceService {
    public int addValue(int pValue) throws IceniServiceException;
    public int subtractValue(int pValue) throws IceniServiceException;
    public int getValue() throws IceniServiceException;
} // end interface PrivateCounterService
```

B Implementation for a Counter Service

```
import org.icenigrid.management.resources.PrivateResourceImpl;
import org.icenigrid.services.basic.IceniServiceException;

public abstract class PrivateCounter extends PrivateResourceImpl
                                     implements PrivateCounterService {
    protected int _counter = 0;
    public int addValue(int pValue) throws IceniServiceException {
      _counter += pValue;
      return _counter;
    }
    public int subtractValue(int pValue) throws IceniServiceException {
      return addValue(-pValue);
    }
    public int getValue() throws IceniServiceException {
      return _counter;
    }
} // end class PrivateCounter
```

C Service Level Agreement for a Counter Service

```
<publicDomain name="public1">
 <policy:accessPolicy>
   <policy:accessRequestCache>50</policy:accessRequestCache>
   <policy:allow startDay="monday" stopDay="friday" stopHour="12" stopMn="00">
    <policy:entity type="organisation" name="eScience"/>
   </policy:allow>
 </policy:accessPolicy>
</publicDomain>
```