A RMI Protocol for Aglets

Feng Lu **Kris Bubendorfer**

School of Mathematical and Computing Sciences
Victoria University of Wellington,
P. O. Box 600 Wellington, New Zealand,
Email: Feng.Lu@mcs.vuw.ac.nz, kris@mcs.vuw.ac.nz

Abstract

Aglets is a mobile agent system that allows an agent to move with its code and execution state across the network to interact with other entities. Aglets utilizes Java RMI to support client-server inter-agent communication. However, Java RMI requires static stubs and skeletons to be precompiled and deployed at both the client and server before communication can take place. There are also issues of interoperability, where the single communication protocol is embedded in the Java RMI stubs and skeletons at compile time. Runtime connection to non Java RMI entities is therefore difficult if not impossible. In Aglets, the use of Java RMI runs counter to modern trends for late-binding and highly nomadic code in heterogeneous environments. To address these shortcomings, we have designed and implemented an alternative RMI architecture that is easily retrofitted into the existing Aglets distribution. We have adopted lightweight stubs generated dynamically by the client's supporting runtime environment, and a single generic skeleton deployed on the server's host machine. In addition, we use the SOAP protocol in our prototype to encode remote calls in XML documents to maximize interoperability, albeit at the expense of performance. These are incorporated into a component based communication stack that permits runtime configuration and selection of the protocols transparently to the agents involved. With our RMI architecture we have overcome considerable usability limitations in Aglets, extending Aglets by providing full access and relocation transparencies.

Keywords: Mobile Agents, Aglets, RMI

1 Introduction

A mobile agent is a program that is not bound to the system on which it began execution, but rather travels amongst the hosts in the network with its code and current execution state (Milojicic et al. 2001). The use of this technology represents a modern programming paradigm for the development of distributed applications. Benefits include a reduction in network load, reduced latency, encapsulation of protocols, asynchronous and autonomous operation (particularly disconnected operation in wireless networks), and their ability to adapt dynamically to changes in the environment (Lange et al. 1999). The mobile agent paradigm is not likely to replace traditional non mobile distributed programming technologies, rather we see it as a complementary technology.

Aglets (Wong et al. 1999, Osshima 1998) was one of the first Java-based mobile agent systems. Aglets provides two mechanisms for inter-agent communication. A message passing mechanism supports the peer-to-peer communications, and Java RMI is provided to support client-server communication. Java RMI (remote method invocation) (RMI 1999, Pitt et al. 2001) is an object oriented RPC (remote procedure calls) (Birrell et al. 1984) protocol that allows the user to invoke methods on a remote object as if it were a local object.

Amongst the goals of middleware are interoperability despite heterogeneity and interaction despite distribution (RM-ODP 1995). Aglets falls short of these ideals, with regard to supporting mobile agents, due to the use of Java RMI for client-server communication. Specifically, Java RMI requires static stubs and skeletons to be precompiled and deployed at both the client and server before communication can take place. This is counter to modern trends for late-binding and highly nomadic code in heterogeneous environments. Although Java RMI allows an agent to dynamically load stubs and skeletons from a remote class repository via a URL, this is not a sufficiently flexible mechanism. For instance, runtime requirements for interoperability may demand a different object representation or protocol. One solution to this dilemma is to separate the stub interface specification from the stub protocol and dynamically generate the stub component at runtime.

From these arguments above, it is clear that to fully support nomadic agents, Aglets needs better mechanisms that provide a greater degree of access and relocation transparencies (RM-ODP 1995). Transparencies hide common complexities, such as heterogeneity and physical distribution, from application programmers and transfer the burden to the infrastructure developer.

In this paper, we present a new RMI architecture for Aglets. Our protocol utilizes an architecture based on the component based framework from RM-ODP (RM-ODP 1995) plus dynamic stub generation and generic skeletons from FlexiNet (Hayton 1999) to retrofit access and relocation transparencies (for both the clients, and servers) into Aglets. The dynamic stubs are generated on demand by the client's supporting runtime environment, and forward remote invocations via a communication stack to the appropriate server. On the server, the generic skeleton demultiplexes all incoming remote invocations to the correct target methods using Java reflection.

We support full relocation transparency. When an existing binding fails, the generated stub automatically contacts a location service (Bubendorfer 2001) to obtain the server's current location. The same functionality is provided by the generic skeleton, to

hide client mobility from the server on return from the target server invoked method.

Full access transparency is provided by separating the stub interface specification from the protocol, and utilizing a component based communication stack. Different protocols can be used with the same dynamic stub - depending on the current runtime interoperability requirements.

2 Remote Object Model

A Java RMI stub is a very heavyweight object. It not only acts as a local representative or proxy for the remote object, but also takes on the responsibilities of communication with the server. A method invocation on the stub is marshaled into a request message and transferred to the server. The server skeleton unmarshals the request, invokes the method on the target object and then marshals and sends the result back to the client stub. Finally the client stub unmarshals and returns the result to the client.

Our RMI architecture adopts a different remote object model to Java RMI, taking an approach based on a combination of RM-ODP and FlexiNet. The client stub is a very lightweight object that is not responsible for the 'on the wire' representation of the invocation (Hayton 1999). Each stub is associated with a protocol stack which provides the communication mechanisms. An invocation on the stub is transferred to the client protocol stack, as shown in Figure 1. It is the protocol stack that marshals the method invocation and communicates with the server. On the server side, the protocol stack unmarshals the request, and then forwards the request to the generic skeleton, which invokes the appropriate method on the target object.

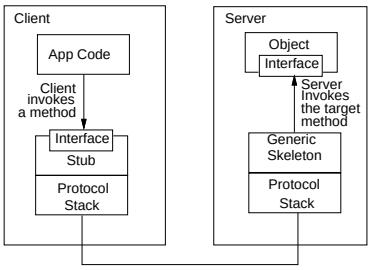

Figure 1: *RMI call to a remote object*

3 RMI Architecture

As discussed in section 2, the stub interface is separated from the protocol stack. This lightweight stub is generated dynamically to represent the remote object within the client application name space and provides a dynamic rebinding mechanism to automatically rebind itself to mobile server object. The protocol stack is a multi-tier set of components, as shown in Figure 2. The layer components of the client side implement the interface *CallDown* to pass client side calls down the stack, and the layer components of the server side implement the interface *Callup* to pass server side calls up the stack. A call object contains the reference to the remote target object, the required method signature and the arguments of the method. The call object is created by the stub and transferred to the top of the client's protocol stack. Each layer performs operations on the call object, and by the bottom of the client side stack, the object reference has been resolved and the call object has been serialized. The following sections 3.1 through 3.5 detail the functionality of each layer.

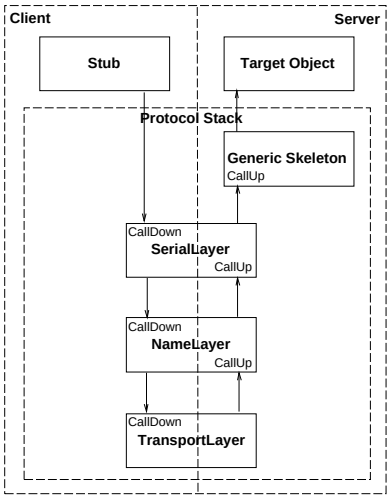

Figure 2: *The RMI Architecture*

3.1 The Stub

The bytecode generator from FlexiNet is used to construct bytecode for stubs on demand. To simplify the generation of the stub bytecode, a template is used to abstract the parts common to all stubs. For example, each stub requires the name of the remote target and a reference to the top of the protocol stack. Since the stub is the proxy for a remote object, it must also provide the appropriate methods as specified by the remote object's interface. These are added to the stub instance using Java reflection to analyze each method signature from the remote object's interface. The stub is generated directly in the client's name space. Generating a lightweight stub from a locally held interface can take less time than downloading a heavyweight stub across the network.

When a stub's method is invoked, the parameters of the method are passed as an array of object. If a parameter is of primitive type, it is converted into the corresponding Wrapper Object type. Then a `call` object is created and associated with the name and signature of the corresponding method as well as the arguments. The `call` object represents an invocation and each method is associated with a different `call` object. When the method `invoke` of the `call` object is called, the `call` object is transferred to the top of the communication stack. The result from the server is stored in the `call` object after the invocation completes.

Remote Invocation introduces two additional exceptions that may be raised during invocation. The first is an exception caused by failure of the network or target machine. The second is an `ObjectNotFoundException` exception that indicates the server Agent was not found on the target machine. We do not know if the remote object was destroyed or simply moved to another location. Thus, the stub tries to rebind to the remote object. This rebinding process is described in Section 4.

3.2 SerialLayer

Serialization is the process of converting an object into a serial, or byte array, form. We serialize an object into a byte stream before sending it to another host and reconstruct the new instance from the byte stream at the destination. Rather than utilizing the built-in Java Serialization technology for the prototype, we use the Apache implementation of the Simple Object Access Protocol (SOAP) (SOAP1.1 2000) to provide a high degree of interoperability and access

transparency. The Apache SOAP implementation supports the encoding of most Java primitive types and their corresponding Wrapper classes, plus some other standard classes, such as String, HashTable, Map and JavaBean object in textual XML documents. There is no general purpose object encoder in Apache SOAP. If users want to transfer their own (non bean) objects as parameters, they have to create special encoders.

3.3 NameLayer

A server object has two identities. A published external name string describes the service provided by the object. An internal reference is used to identify the location and interface of the service Agent. The resolution of an external name provides a client with an internal reference for the service. The reference consists of three parts: *protocol*, *address* and *id*. The *protocol* states which application protocol is used. The *address* represents the last known address of the server Agent, including IP address and port number. The *id* is a globally unique number used to identify the server object. It will never change after it is generated. The reference can be represented, parsed and stored, in stringified format.

3.4 TransportLayer

The TransportLayer is used to send the client message to the server and receive the server's response. On the server, the TransportLayer receives the request message, constructs a call object, and forwards it up to the NameLayer. As the prototype utilizes SOAP for object serialization, we also utilize HTTP as the matching TransportLayer protocol.

3.5 Generic Skeleton

The Generic Skeleton performs the local method invocation on the targeted object. Reflection is used to identify the correct target object and method from the signature encoded by the client stub. The resulting returned value or exception from the target method is then stored in the call object, which is passed back down the stack.

3.6 Client Side Call Processing

In this section, we will detail the process of a call being made down the stack.

- A method is invoked on a client stub. A call object is created, consisting of, the internal reference to the remote object, the signature of the method and any method arguments.
- The call object is passed to the top of the protocol stack, the SerialLayer. The SerialLayer serializes the contents of the call object into a XML document. When the invocation returns, it will also deserialize the result. The modified call object is then passed to the NameLayer.
- The NameLayer resolves the reference of the target object and extracts the target machine address.
- The TransportLayer wraps the XML document into a HTTP Request message, opens a TCP connection to the target and then sends the HTTP Request message. The TransportLayer then waits for the response[1]. On receipt of the server response, the TransportLayer extracts the XML document from the HTTP Response Message and stores it in the original call object. The call object is then passed back up through the stack. If the connection timeouts, then the call object is passed back up with the appropriate exception.

[1]Any call to migrate the Agent at this point must wait for the call to complete or timeout.

3.7 Server Side Call Processing

A call up the stack is somewhat more interesting, as we now must deal with the fact that the server may have moved, invalidating the reference held by the client. In this section, we will go through the process of an incomming call up the stack.

- The TransportLayer initializes the listening port and then waits for requests. When it receives the HTTP Request Message, it creates a call object containing the entire message.
- The NameLayer extracts the target id from the request message header. Then, it checks if the target Agent exists on this host. This is important, as the mobile agent may have since moved to a different host, or ceased execution. Each host runs an extended Aglets Tahiti server that automatically maintains a register of all Mobile Agents executing within it. If the target Agent is not in the register, the call up the stack terminates, and an *ObjectNotFound* exception is returned to the client. The client stub will then attempt to relocate the Agent on a different host, as described in section 4.
- The SerialLayer extracts the XML document from the HTTP Request message, and then parses the XML document to get the method's signature and any objects passed in as arguments. When the invocation returns, it serializes the result into the returning XML document.
- Finally the Generic Skeleton uses Java reflection to find and invoke the required method on the target object. The result is then stored in the call object which is passed back down the stack. If the result is an exception, then the exception is stored in the call object and the exception flag in the call object is set.

4 Dynamic Rebinding

Aglets needs to have relocation transparency to hide the mobility of Agents within the network. When a server object moves to another host, all references held on this object by its clients become out-of-date. From this point, all subsequent invocations on this object will result in an *ObjectNotFound* exeption generated by the host encoded in the out-of-date reference.

Relocation transparency requires that resolving the new object location and rebinding occur transparently to the client. This duty falls to the stub, which contains a reference to a *Whitepages* location repository (relocator in RM-ODP terminology). This service is then queried by the client stub to resolve the new location. The whitepages service is automatically maintained on object migration by extensions to the Tahiti Aglet Server. This location service is a simplified version (less scalable) of the NOMAD global location service (Bubendorfer et al. 2003), which is sufficient for our purposes in the prototype. The new reference is then used by the stub to rebind to the object, and the invocation is then reissued.

5 Interoperability

CORBA (OMG 1999), DCOM (Horstmann et al. 1997) and Java RMI, use the remote object model to allow objects in one machine to access objects in other machines. However, these protocols have limited interoperability, although some protocol bridges have been developed. Also, the standard implementations of these protocols use randomly generated port to communicate. This means that they are extremely difficult to pass through firewalls, which typically block all ports except for several well known ports, such as 80 and 25. Tunneling has been suggested to enable the firewalls to be sidestepped, but the configuration is complex. However, the fundamental problem is that it is difficult to design a universal RMI protocol — SOAP is one possibility. However, SOAP is recognized as providing considerably worse performance than other RMI protocols (Govindaraju 2000).

5.1 SOAP

SOAP provides a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML(Extensible Markup Language) (SOAP1.1 2000). SOAP utilizes XML to define a set of platform independent encoding rules to represent data that are easy to generate and parse. SOAP is a text-based protocol, in contrast to current RMI protocols, which are binary-based protocols. Text-based protocols such as SOAP are especially easy to debug because all of the data transferred is in a human readable format. In addition, SOAP can use a variety of existing Internet Protocols, such as HTTP, SMTP. This makes it very easy for distributed applications to communicate with each other in a network environment with firewalls. The most important thing is that SOAP is widely supported by the main industry consortiums, such as Microsoft and IBM.

5.2 SOAP & Serialization

SOAP defines a set of encoding rules that describe how to represent the different types of object in an XML document. A set of XML schemas are used to define the different types and describe their structures. With the XML schema and the corresponding data values, an XML document can be constructed. On the other end, an XML document may be destructed to a set of data values and corresponding XML schema.

5.3 SOAP & RPC

An RPC may be represented as an object consisting of the address of the target, a method signature and the arguments of the method. Therefore, any RPC call can be encapsulated into a SOAP XML message. Using HTTP as the carrier for SOAP messages provides the advantage of being able to use the formalism and decentralized flexibility of SOAP with the rich feature set of HTTP (SOAP1.1 2000).

Consider the following example. The *EchoImp* class implements the interface *Echo* and acts as a server object. It is located in the server *tahi.mcs.vuw.ac.nz* on prot *9000*. The global id *12345687* is assigned to this object.

```
public class EchoImp implements Echo {
  public String echo(String message, int id)
     throws IncorrectIDException{
    return message + " " + id;
  }
}
```

When the client tries to invoke the method on the server object *EchoImp*, a HTTP Request message is generated as a wrapper of the SOAP message. The SOAP Action field indicates the target object global ID *12345687*. The SOAP Envelope specifies the overall structure of the message. A method call is modeled as a compound data type with a sequence of parameters. The parameters of this method call are: arg0(message) *This is a test* and arg1(id) *123*.

```
POST / HTTP/1.0
Host: tahi.mcs.vuw.ac.nz:9000
Content-Type: text/xml; charset=utf-8
Content-Length: 469
SOAPAction: "12345687"

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:echo xmlns:ns1="12345687"
SOAP-ENV:encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/">
<arg0 xsi:type="xsd:string">This is a test</arg0>
<arg1 xsi:type="xsd:int">123</arg1>
</ns1:echo>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The server wraps the result into a HTTP Response Message. It is similar to the usual HTTP Response except the fact that there is a suffix "Response" added to the element name representing the method name.

```
Content-Type: text/xml; charset=utf-8
Content-Length: 482

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:echoResponse xmlns:ns1="12345687"
SOAP-ENV:encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:string">
Your message = This is a test and id = 123
</return>
</ns1:echoResponse>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

If the result is an exception, the content of the body is a SOAP:Fault structure which describes the exception.

```
Content-Type: text/xml; charset=utf-8
Content-Length: 480

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<SOAP-ENV:Fault>
<faultcode>
nz.ac.vuw.nmi.util.ObjectNotFoundException
</faultcode>
<faultstring>
nz.ac.vuw.nmi.util.ObjectNotFoundException
</faultstring>
<faultactor>17345687</faultactor>
</SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

6 Implementation in Aglets

The implementation of the RMI protocol in Aglets required alterations to two standard Aglet components. Firstly, the Tahiti server, that creates and maintains the Aglet's runtime environment, was modified to initialise the RMI protocol stack. Secondly, the `Aglet` class, that defines the methods for controlling an Aglet's life cycle and behavior, was extended by the subclass `RmiAglet`. The new `RmiAglet` class permits Aglets to utilise the new RMI communications stack. The default Aglet Dispatch method was modified to add migration failure recovery by reestablishing the original *Whitepages* entry. In addition three new methods were required to interact with the *Whitepages*. The method `lookup` queries the *Whitepages* to find a service. The method `export` is used to export a service name into *Whitepages* when a server object wants to advertise its service. The method `unexport` removes a service from the *Whitepages* when a server object is disposed of or migrates[2] to another host.

7 Performance Analysis

We conducted experiments to compare our performance to that of Java RMI. In these experiments, we implemented four pairs of client/server programs that communicated using our RMI (using SOAP components), Java RMI, Java TCP Sockets and Native TCP Sockets. All applications are written in Java, except for the application using Native TCP socket which uses the Java Native Interface (JNI). All programs were run on the same two machines, a NetBSD client and a Sun Solaris server, on a 100 Mb/s switched Ethernet network. We recorded the interval from the time when a client sent the package to the time when a client received the response.

Size(byte)	RMI(ms)	Java RMI	Java TCP	Native TCP
10	265.05	0.70	0.40	0.22
100	266.63	0.78	0.52	0.25
1000	277.97	1.56	1.66	0.47
2000	279.78	2.62	2.85	0.61
3000	388.96	2.94	3.77	0.86
4000	278.56	3.7	4.78	1.02
5000	280.37	4.01	6.64	0.92
6000	386.18	4.06	7.32	1.03
7000	386.66	4.78	8.43	1.08
8000	379.76	5.28	9.32	1.32
9000	300.2	6.02 8	9.87	1.43
10000	388.15	6.54	10.75	1.36

Table 1: Performance

As shown in Table 1, the performance of our RMI is significantly worse than Java RMI, between 60-100 times slower in fact. This is result is consistent with previous research (Govindaraju 2000) utilizing SOAP underneath an RMI mechanism. From this it is clear that the performance is a SOAP issue. Clearly, for SOAP to have any credibility as a protocol for interoperability, vast improvements must be made to the implementations currently available.

However, these results should not be taken out of context. They provide a proof-of-concept that the RMI architecture presented in this paper can be retrofitted into Aglets. Indeed, one of the points of our architecture is that it is component based, and therefore we can substitute alternative protocol components both at compile and runtime. Alternative components should be able to achieve better performance, albeit at the cost of interoperability.

[2]Once on the new host, the server Aglet automatically re-exports its service name.

8 Conclusions

We have presented a RMI protocol that can be retrofitted into Aglets via a combination of dynamic stub generation, and server side extensions to the Aglets Tahiti server. The stub supports a dynamic rebinding to automatically relocate and bind itself to a mobile server Agent providing relocation transparency. The protocol stack is designed as a multiple-tier set of interchangable components. SOAP is integrated into the protocol stack as the SerialLayer because SOAP's interoperability makes it valuable in heterogeneous environment. Though our prototype's performance is considerably worse than Java RMI, this is a result of selecting SOAP as a serialization component in the prototype's protocol stack. This none-the-less provides a proof-of-concept that the RMI architecture presented in this paper can be retrofitted into Aglets with clear advantages in access and relocation transparency. In this we overcome a considerable usability limitation in Aglets.

References

A. D. Birrell & B. J. Nelson (1984), Implementing Remote Procedure Calls, ACM Transactions on Computer Systems.

Danny B. Lange & Mitsuru Oshima (1999), Seven Good Reasons for Mobile Agents, Communication of ACM, volume 42.

Dejan Milojicic, Frederick Douglis & Richard Wheeler (2001), Mobility - Processes, Computers and Agents, Addition Longman.

D. Wong, N. Paciorek & D. Moore (1999), Java-based Mobile Agents, Communication of ACM, volume 42.

Esmond Pitt & Kathleen McNiff (2001), The Remote Method Invocation Guide, Pearson Education.

International Standards Organisation (1995), Open Distributed Processing Reference Model - Part 3: Architecture.

Kris Bubendorfer (2001), NOMAD: Towards an Architecture for Mobility in Large Scale Distributed Systems, Ph.D. Thesis, Victoria University of Wellington.

Kris Bubendorfer & John Hine (2003), NOMAD: Application Participation in a Global Location Service, Lecture Notes in Computer Science, number 2574, pages 294-306.

Madhusudhan Govindaraju (2000), Requirements for and Evaluation of RMI Protocols for Scientific Computing, IEEE.

Markus Horstmann & Mary Kirtland (1997), DCOM Architecture, Technical Report, Microsoft.

Mitsuru Osshima (1998), Aglets Specification 1.1 Draft, IBM Japan.

Object Management Group (1999), The Common Object Request Broker:Architecture and Specification, 2.3.1 edition.

Richard Hayton (1999), FlexiNet Architecture, Technical Report, ANSA.

SOAP 1.1 Specification (2000), World Wide Web Consortium, http://www.w3.org/TR/SOAP/.

Sun Microsystems (1999), Java Remote Method Invocation Specification.