SoapRMI C++/Java 1.1: Design and Implementation

Aleksander Slominski (aslom@cs.indiana.edu), *
Madhusudhan Govindaraju (mgovinda@cs.indiana.edu),
Dennis Gannon (gannon@cs.indiana.edu),
Randall Bramley (bramley@cs.indiana.edu)

Abstract

Java RMI provides a simple and elegant paradigm for communication with objects in remote address spaces. RMI is a de facto standard for communication in distributed systems that are written in different languages and optimized to run in disparate environments. XML [10] has emerged as a promising standard for language-independent data representation, and HTTP as a widely-used firewall-friendly network protocol. It is now possible to design and develop a communication system that combines the elegance and strength of Java RMI with the ubiquity of HTTP and platform and language independence of XML. SOAP [12] defines XML based communication and SOAP RPC precisely states the protocol for using XML as the data format and HTTP as the network protocol. This paper presents the design issues in layering a C++ and Java based RMI system on top of SOAP RPC along with an efficient XML Pull Parser that we designed to parse SOAP calls. We explain the various features of the resulting SoapRMI system: dynamic proxies, stub-skeleton generation from XML specification, interoperability, exception handling and different "Naming" services.

Key Words: RMI, Distributed Systems, XML, SOAP, Naming

1 Introduction

Component-based scientific computing applications place ever-growing demands on fast, reliable communication and transfer of data in distributed heterogenous environments. Components are software modules that implement a set of standard behavior [33]. Component architectures, like JavaBeans [24], Microsoft's COM [20], the CORBA component model (CCM) [27], the Common Component Architecture (CCA) model for parallel and distributed scientific computations [4], are frameworks specifying how software components within that framework can communicate with each other. In a truly heterogenous system, it is important that components talk to each other across these architectures. Components

*Department of Computer Science, 150 S Woodlawn Avenue, Bloomington, IN 47405. Ph: 812 855 8305 Fax: 812 855 4829 should employ, a high-level, object-oriented, remote procedure call model for communication that is easily understood and implementable across language and architecture boundaries.

In a distributed system a component may be connected to many other components at a given time, each connection possibly using a different protocol. The choice of protocol depends on dynamically changing factors: size of data, security policies, quality of service and error or exception handling [18]. It is desirable to have a simple common-denominator protocol that all components are guaranteed to support. This protocol can be used by components to negotiate the use of other specialized protocols for exchanging large volumes of data. Moreover this common protocol can also be used to return exceptions and errors. Since components are written in different languages, the protocol must be language independent.

The format of data and the protocol used to exchange it is a determining factor in the degree of interoperability among applications. The lack of a reliable, simple and universally deployed data-exchange format has long limited effective communication between heterogenous systems. The Extended Markup Language (XML) is now accepted as a standard for representing data in a platform independent manner, and HTTP is now a universally supported network protocol for exchanging information over the internet. This ubiquity means moving XML data via HTTP is an attractive way for distributed systems to communicate. The Simple Object Access Protocol (SOAP) does this and has emerged as an open standard that is simple and elegant for information exchange. Thus, SOAP is a natural choice as a common protocol that software components can use as their base protocol.

This paper addresses the following questions:

- Is it possible to design a SOAP based RMI system in Java and C++ that conforms to the Java Remote Method Invocation (RMI) [22] API ?
- How does one design the RMI system that interoperates between C++ and Java RMI systems ?
- What extensions are needed to achieve interoperability with well-known systems like .NET [19] and Apache-SOAP [1] ?

This paper shows how to design an RMI system that uses SOAP as its communication protocol. The paper also discusses the different techniques used for object serialization and remote exception handling between the C++ and Java implementations of SoapRMI. The paper explores possible directions in which one can proceed to interoperate with Apache SOAP [11] and the .NET project.

2 Remote Method Invocation

Java RMI is a framework designed to simplify distributed object oriented computing in Java. An overview of the Java RMI model is shown in Figure 1.

RMI is essentially a client-server model. The **stub** acts as a proxy for the remote object. The **skeleton** is an object that lives in the same JVM as the remote object and handles communication with the stub. The registry is used to manage remote references. The server binds a remote reference to itself on to the registry. To obtain a remote reference to the server, the client contacts a **registry** which may be on a different remote host. The client

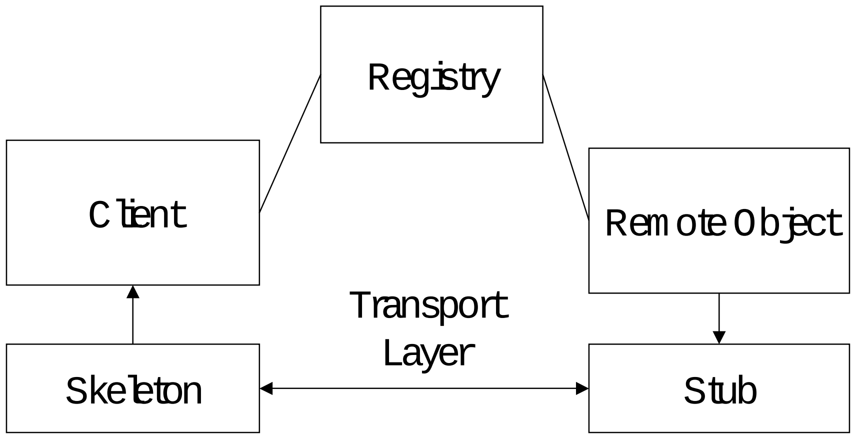

Figure 1: RMI Architecture

can use the remote reference obtained from the registry to invoke methods on the remote object.

3 Related Work

Different implementations of RMI differ in their design of object serialization/deserialization, stub and skeleton generation, the Naming service they use, whether or not they use extensions to the existing Java RMI API and the degree of interoperability with RMI systems designed for different languages and varying environments.

JavaRMI [22] developed by SUN is the de-facto standard for RMI systems. JavaRMI supports SUN's native protocol as well as IIOP [21]. However, it is meant to communicate only between JVMs and it does not provide support for interoperability with other languages.

NinjaRMI [34] was developed at the University of California, Berkeley, as part of the Ninja project [26]. The project adds some interesting features to Java RMI but does not address interoperability with RMI systems written in other languages.

NexusRMI [7] developed at Indiana University leverages the Nexus [15] protocol to achieve interoperability between Java and HPC++, a version of C++ with library extensions for parallel computing [16] . Since Nexus is a binary protocol, NexusRMI achieves efficient performance when communicating with HPC++. [6].

To gain significant improvement in performance, Philippsen et al [28] built drop-in replacements in their RMI implementations.

Thiruvathukal et al. [31] used explicit methods to serialize and deserialize each object's internal state. Their implementation of Java RMI provides unique features like dynamically obtaining the interface of remote objects.

In [32] Veldema et. al. implemented an RMI system for their Albatross [3] system. Their implementation is optimized for homogeneous systems and achieves low latency and high bandwidth.

Our earlier implementation of SoapRMI, [18], was developed only in Java and was designed in the classical style with stubs and skeletons to support remote objects and object serialization/deserialization.

4 Parsing Strategies for XML content

The SOAP protocol specifies the data format of each call to be in XML. Thus the wire format of every method call and its return value is XML. Reconstructing an object from its wire representation (called deserialization) involves parsing XML and extracting the values of data members of objects. Hence parsing of XML payloads plays a critical role in the performance of any SOAP based RPC system. There are four current approaches to parsing XML data:

- DOM (Document Object Model) [9] is a tree structure API issued as a W3C recommendation in October 1998. An XML document is represented as a tree whose nodes are elements, text and so on. An XML processor generates the tree and hands a handle to the root node to the application program. The use of DOM is ideally suited for situations when the structure of an XML document needs to be modified and the document in memory needs to be shared with others.
- SAX (Simple API for XML) [13] is an XML processor that parses an XML document and generates events such as "start of element" and "end of element". An application is expected to listen for these events and process them according to its needs. SAX is more efficient than DOM if the document is big and does not fit into memory. SAX is preferred over DOM when the application is interested only in specific elements in a document.
- XPP (XML Pull Parser) [29] is designed for applications that require a small size pull parser. In SoapRMI the structure of the XML document that needs to be deserialized is known in advance. The deserialization routine needs to be able to access the elements in succession and does not need to access elements that it has already read. As a result the *streaming* model of XPP is well-suited for SOAP based RMI systems. It has full support for namespaces and optional support for mixed-content in XML (see section 3.2.2 in [14]. XPP has identical APIs for C++ and Java, and both the C++ and Java versions of SoapRMI use XPP for deserialization.
- Mixed Solutions (Progressive DOM): The XML community has been advocating a mix of DOM and SAX for parsing XML content. For applications that need events just for the part of the document they are parsing, the SAX model is used. However, whenever a partial tree of the document is needed, the XML processor stores a stream of events to build a tree and then hands it over to the application.

5 Architecture of SoapRMI 1.1

The design objectives of SoapRMI 1.1 are as follows:

- To implement an RMI system in C++ and Java that conforms to SUN's Java RMI API.
- Provide access to Directory services based on RMI registry and LDAP [2].

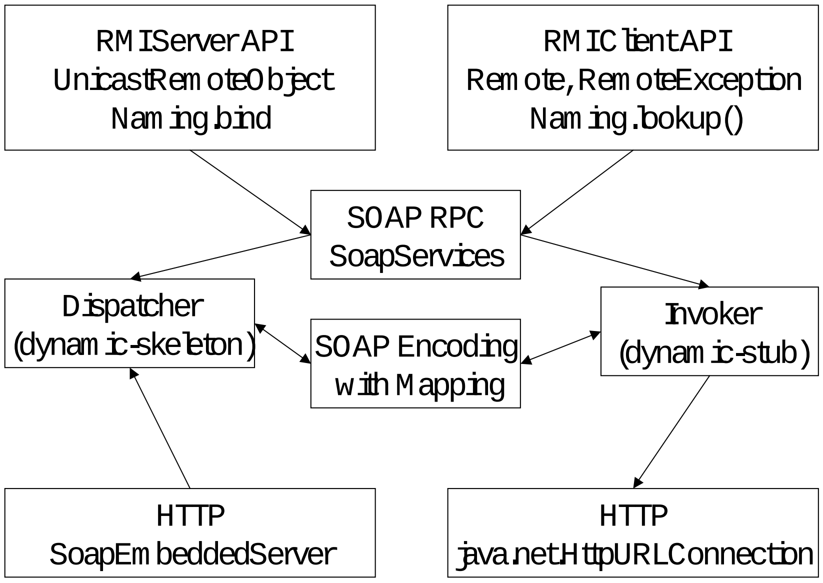

Figure 2: SoapRMI 1.1 Java Architecture

- Architect a SoapRMI server to function as a web-service if needed.
- Automate the generation of stubs and skeletons from XML-Schema based specification of interfaces.
- Enable multiple object serialization mechanisms in SoapRMI-Java and recursive serialization in SoapRMI-C++.
- Use the dynamic proxy classes provided as part of Java 1.3 to reuse the same stub and skeleton for every remote object.
- Support exceptions across C++ and Java platforms in an seamless manner.
- Interoperate with existing SOAP based frameworks like .NET and Apache SOAP.

Figure 2 shows the architecture of SoapRMI 1.1 Java framework. It is designed in a modular fashion to enable the possibility of plugging in different implementations of various modules at a later date. The *UnicastRemoteObject* is used by objects to export themselves as remote objects. Every client and server has access to a *soap-services* module. This module encapsulates all the services like SOAP serialization and deserialization that any object in the RMI system may need. A serialized object is dispatched using a *dynamic-stub* to the HTTP *connection* layer. Since we use HTTP as the network protocol, each call has to be prepended with an HTTP header. On the receiving end, an HTTP *embedded-server* daemon waits for requests. Every request is directed to the *dynamic-skeleton*. The same dynamic stub and dynamic skeleton is used for every call of every remote object. Each SoapRMI call and its parameters are associated with namespaces. The *soap-encoding/Mapping* layer maps the namespaces to known interfaces and types in the system.

Figure 3 shows the architecture of SoapRMI 1.1 C++ framework. It is designed in the traditional RMI style with a stub and skeleton for every remote object. The stubs use a templated factory for serialization and deserialization of parameters. A *communication* layer abstraction is interposed between the stub and the HTTP layer so that each call can be

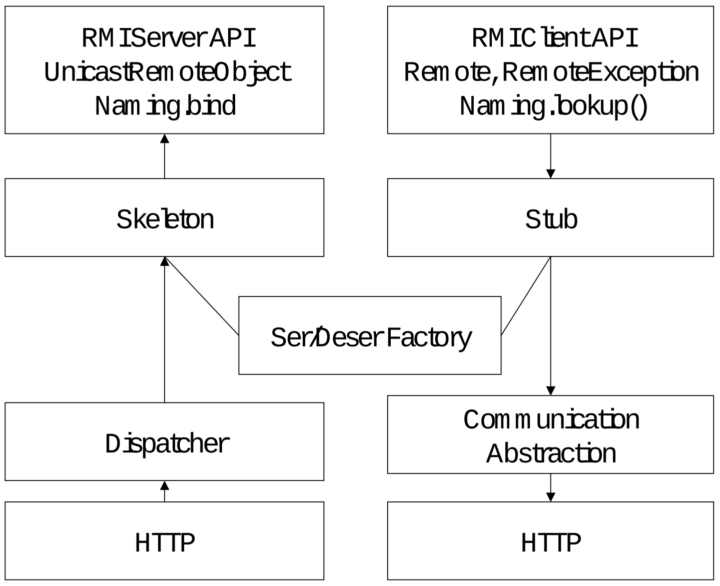

Figure 3: SoapRMI 1.1 C++ Architecture

wrapped in a SOAP envelope. The SOAP envelope is the top element of the XML document representing the message. The HTTP layer adds the relevant headers before sending the SOAP call on the wire. On the receiving end, a *dispatcher* object spawns a new thread for every request that it receives from the HTTP port and dispatches it to the skeleton object corresponding to the remote object to which the call is directed.

5.1 RMI Registry

Traditional RMI systems use a registry as the meeting point for exchanging remote references between clients and servers. A registry is essentially a table with a list of references bound to well known names. A registry is expected to provide the basic features like binding a reference to a well known name and looking up a reference based on that well known name. SoapRMI provides a registry service in the traditional style. A remote reference is stored as a *port*, explained further in Section 8. So, a *bind* request or the return of a *lookup* request will result in conversion of the remote reference to a SOAP compliant XML representation format on the wire.

Figure 4 shows the different ways in which SoapRMI can bind remote references. Apart from the traditional RMI style registries for both of the Java and C++ implementations, SoapRMI can also bind to an LDAP [2] service using the JNDI [23] interface. SoapRMI-C++ uses an indirection to a Java program to bind to LDAP. It is even possible to store remote references in the local filesystem and reuse the reference at a later time by deserializing it from the file. This is possible because SoapRMI remote references can be represented as an XML string in ASCII format.

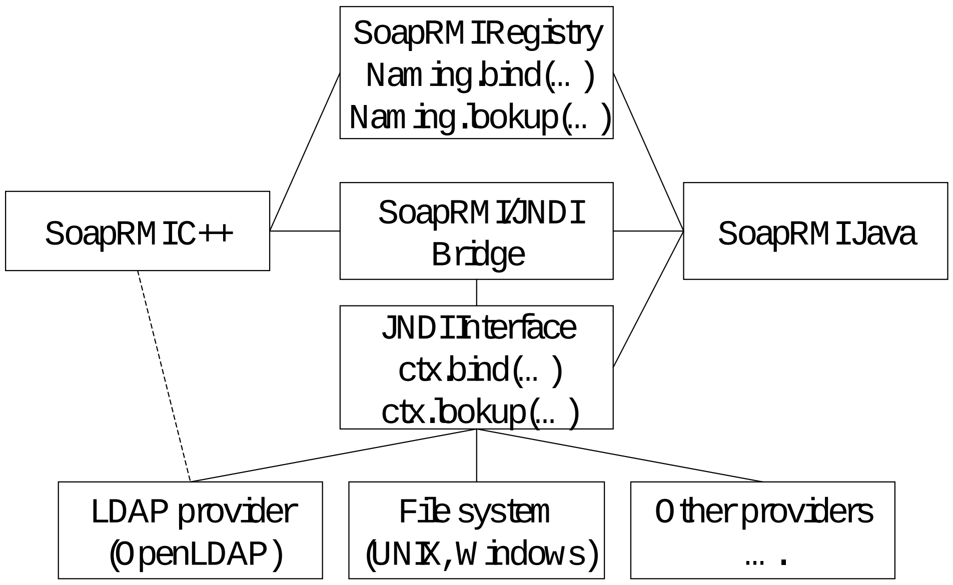

Figure 4: SoapRMI 1.1 Naming Architecture

5.2 Web Service

A WebService is any service that handles requests at a well known location such as a URL. Such services are designed to run persistently. Since these services reside at well known locations, clients are expected to directly connect to them without the help of a registry. SoapRMI is also designed to function as a WebService. A SoapRMI Client (C++ or Java) can be provided with the location of the SoapRMI-WebService at startup or any other desired time . The runtime machinery can then create a remote reference that points to the WebService and direct the client's requests accordingly. The design of an RMI system as a WebService obviates the need for registries. Web services of these kind are now common as can be seen in the .NET [19] and Apache [1] projects.

6 Object Serialization

Unlike Java, C++ does not have a standard for object serialization. There are currently three common ways of implementing object serialization in C++:

- Java Model: In Java, default serialization is provided for all non-transient fields. User objects can override the default serialization by providing readObject() and writeObject() methods for the object. We can emulate this behavior in C++, by ensuring that each serializable object implements two methods: one for serialization (void writeObject(Buffer& buffer)) and another for deserialization (TestObject* readObject(Buffer& buffer)). These methods need to be accessible from the runtime library.
- HPC++ Model: Every serializable object can declare a global method to be its *friend*. The runtime library can then use this global method to access an objects's internal state to serialize or deserialize it. This technique is effectively used in HPC++.

- Template Factory Model: A method that is templated on the type of each serializable object in the system and accesses the internal state of the object using getX() and setX() methods. This model of enforcing accessor methods is used in Java Beans [24]. The runtime library uses a factory method to obtain the serialization or deserialization routine based on the type of the object.

SoapRMI-C++ uses the Template Factory Model for serialization and deserialization of objects. The methods in the factory are all generated by a specialized stub-compiler. The stub-compiler receives the object specification in XML-Schema [8] and generates C++ code to serialize and deserialize objects of the type specified by the schema. Note that Java's reflection mechanism allows the library to introspect the JVM to determine the types and field names of data members in an object. However, such information is not available in C++ and hence SoapRMI-C++ uses the Template Factory Model.

```
class TestObject  {
 private:
 public:
  long llong;
  short sshort;
  int iint;
  float ffloat;
  char cchar;
  double ddouble;
  string sstring;
  // constructors and destructor
  // overloaded operators
  // accessor methods
};
```

Figure 5: A C++ Class for "TestObject".

An example of a templated serialization method is shown in Figure 6. The serialization routine is for an object defined as in Figure 5. Every object calls a serialization method for all its data members. This recursive-serialization process is sometimes unsafe as it is possible to get into an infinite loop when serializing linked-lists that have cycles in them. We plan to add mechanisms to detect such loops and thereby avoid the problem of infinite-loops in serialization.

SoapRMI-Java supports both recursive and multi-ref forms of serialization. The multi-ref technique serializes a given object only once. An object is serialized normally the first time, then on successive encounters only a reference (unique-id) to the object is serialized.

Polymorphism can be defined as the ability to pass a derived type wherever a base type is expected. Since Java supports polymorphism for every method, this features naturally maps to SoapRMI-Java. In C++ however, it is a bit tricky to support this feature. SoapRMI-C++ uses the C++ RTTI (Run Time Type Inference) feature to detect the type of parameter

```
void xSoapPack<TestObject>(SerEnv& serEnv, Buffer& buffer,
                           TestObject& testObject, string tagName) {
  buffer += "<''+tagName+'' xsi:type='ns1:samples.pingpong.TestObject'
                xmlns:ns1=urn:soaprmi-v11:temp-java-port-type'>\n";
  xSoapPack<long>  (serEnv, buffer, testObject.get_llong(),  "longValue");
  xSoapPack<short> (serEnv, buffer, testObject.get_sshort(), "shortValue");
  xSoapPack<int>   (serEnv, buffer, testObject.get_iint(),   "intValue");
  xSoapPack<float> (serEnv, buffer, testObject.get_ffloat(), "floatValue");
  xSoapPack<char>  (serEnv, buffer, testObject.get_cchar(),  "charValue");
  xSoapPack<double>(serEnv, buffer, testObject.get_ddouble() "doubleValue");
  xSoapPack<string>(serEnv, buffer, testObject.get_sstring() "stringValue");
  buffer += "</" + tagName + " >\n";
}
```

Figure 6: Templated C++ Serialization method for TestObject.

being passed in a remote call. Based on the type an appropriate factory method is called to handle the serialization and deserialization of the object.

The SOAP specification does not state how an object instance should be instantiated on the receiving side to process the call. SoapRMI provides an RMI layer on top of SOAP, but still ensures that data representation on the wire for every call is compliant with the SOAP 1.1 specification.

7 Classical Stub-Skeleton Design in RMI Systems

In classical RMI the system is designed as a client-server model. The client talks to the remote object through a proxy called **stub**. The stub itself talks to an active object on the server side called the **skeleton**. The job of the skeleton is to receive a request from the stub and direct it to the remote object. The skeleton also receives the response from the remote object and packages it back to the client. The stubs and skeletons are generated by a *stub-compiler* are hidden from the user.

SoapRMI-C++ uses the classical stub-skeleton design for communication between clients and servers. The description of remote object interfaces needs to be provided using XML-Schemas, which define powerful constraints on document structure. They also provide support for namespaces and standard primitive data types. Typically an Interface Definition Language (IDL) [17] is used to describe the remote object interface. However, XML can be used instead of IDL as it allows greater flexibility in the content of a specification than a static CORBA-style IDL does [5].

7.1 Dynamic Proxy

Java 1.3 has added support for dynamic proxy classes as part of the Java Reflection API. A dynamic proxy class implements a list of interfaces that are specified at runtime. Any method belonging to one of the interfaces provided in the list can be invoked on the proxy. The proxy can be used in user programs as if it actually implemented these interfaces. A dynamic proxy class is a type safe proxy object for a list of interfaces [30].

This feature provides a powerful paradigm in designing RMI systems in Java. In the classical model of Java RMI (see section 2) a new stub and skeleton needs to be generated for every remote object. With dynamic proxy classes, it is now possible to reuse the same stub and skeleton for every remote object.

```
public Object invoke (Object proxy, Method m, Object[] args)
  throws Throwable {
    String methodName = m.getName();
    Class classname = m.getDeclaringClass();
    // obtain all the parameters (args) in the call
    // make network connection with remote endpoint
    // specified in the port
    // open output stream
    // serialize every parameter into the stream
    // close the output stream
    // open input stream
    // wait for reply to be filled into intput stream
}
```

Figure 7: Generalized Stub method using Dynamic Proxy.

For every invocation of a method on a remote object, the method *invoke(...)* in Figure 7 is called in SoapRMI-Java. This method belongs to the *java.lang.reflect.InvocationHandler* interface. The first parameter, *proxy*, is a reference to the proxy object that is invoking the remote method. This reference makes it possible to use the same invocation handler with multiple proxy instances. The second parameter of type *Method* corresponds to the interface in which the method invoked on the proxy was declared. The third parameter is an array of values for the arguments passed in the method invocation. The information contained in these parameters is enough for SoapRMI-Java to serialize a call into the output stream.

8 Remote Reference

A remote reference is a reference that points to an object instance living in a remote process. In C++ world it is commonly called a *global pointer*. A global pointer is a generalization of the C pointer type. A remote reference or a global pointer is used to make method invocations on remote objects but the syntax of the call is identical to that of a local method call, hiding its remote/local nature.

Figure 8 shows an example of how a remote reference can be described in XML. In SoapRMI, a remote object is considered as a *port*. A port consists of a *name*, a list of *port types* and a list of *endpoints*. The name of the port is the well-known name by which the port can be identified in local repositories or user programs. The two sub-elements of a port

```
<Port>
  <name>MyHelloWorld</name>
  <ports>
    <portType>
      <uri>urn:soaprmi-v11:events</uri>
      <name>hello-world-examples</name>
    </portType>
    ...
  </ports>
  <endpoints>
    <Endpoint>
      <location>http://exodus:8888/server/MyServlet</location>
    </Endpoint>
    ...
  </endpoints>
</Port>
```

Figure 8: Example of a Remote Reference.

type, uri and name, uniquely identify the interface that the remote object implements. The receiver of the call is expected to have a mapping from this port type to one of its known interfaces to be able to invoke methods on the remote reference.

A list of endpoints enables the service, which the remote reference represents, to be available via multiple connections. It is quite natural for a web service to serve requests on many different ports to increase the number of incoming calls it can serve.

When a user passes a remote reference as a parameter (for callbacks) the RMI machinery detects this and automatically converts it to a XML port representation on the wire as in Figure 8. On the receiving end the RMI machinery detects it and converts it back to a remote reference before passing it to a user. The representation of a remote reference as a port is language independent and thus makes it convenient for receiving processes (Java or C++) to map it into their own internal format.

9 Exception Modeling

Both C++ and Java have built-in support for exception handling. To a user, a method invocation on a remote reference should seem like just another local method invocation. For this to be true a user of SoapRMI should be able to catch exceptions that are thrown whenever a remote method is invoked, even when the local and remote objects are in different languages.

The two general kinds of exceptions that must be handled are:

- System Exceptions: The standard runtime exceptions as have been defined by the Java RMI API, for example RemoteException, AlreadyBoundException and MarshallException.

```
<SOAP-ENV:Fault>
<faultcode>SOAP-ENV:Server</faultcode>
<faultstring>no method named getCount</faultstring>
<detail>soaprmi.ServerException: no method named getCount
  at soaprmi.soaprpc.SoapDynamicSkeleton.dispatch
  at soaprmi.soaprpc.SoapDispatcherImpl.dispatch
  at soaprmi.soaprpc.SoapEmbeddedServerConnection.doPost
  at soaprmi.soaprpc.SoapEmbeddedServerConnection.process
  at soaprmi.soaprpc.SoapEmbeddedServerConnection.run
  at java.lang.Thread.run
</detail>
</SOAP-ENV:Fault>
```

Figure 9: Example of a Remote Exception on the wire.

- User Exceptions: The exceptions that result from executing user code in the implementation of a remote method.

Figure 9 shows an example of an exception marshalled onto the wire as a SOAP-Fault element.

In Java, it is natural to expect a stack-trace along with an exception. This feature however is not present in C++. This difference in C++ and Java exception models leads to different mechanisms in handling exceptions depending on the originating source context.

Exceptions are not serializable objects nor do they conform to the Java Beans definition. So, the SoapRMI-Java runtime library has specialized routines to serialize and deserialize remote exceptions.

It is relatively easy to handle exceptions that are thrown and caught in similar execution environments. An exception thrown in a JVM is marshalled with the entire stack trace as part of the *detail* element. If the exception is thrown by a C++ process it is marshalled with a message generated by the runtime library as no stack trace is available. The *faultstring* element always contains the message that is provided with the exception that is thrown. The receiving end needs to throw the exception back to the user who made the remote call that resulted in the exception. It is important that the type of exception thrown to the user is the same as the type thrown by the remote object. This is because a user may not catch all the exceptions thrown by the remote method, and might be interested in just a few exceptions. In SoapRMI, there is an *a priori* agreement that the first string in the *detail* element corresponds to the actual type of the exception.

In case the runtime library receives an unknown type of exception, a base class exception (which every remote exception extends) or a specialized *UnknownException* is thrown.

10 Interoperability

An important design goal of our project is to determine the maximal subset of the Java RMI API that can be interoperably supported by both the C++ and Java versions of SoapRMI.

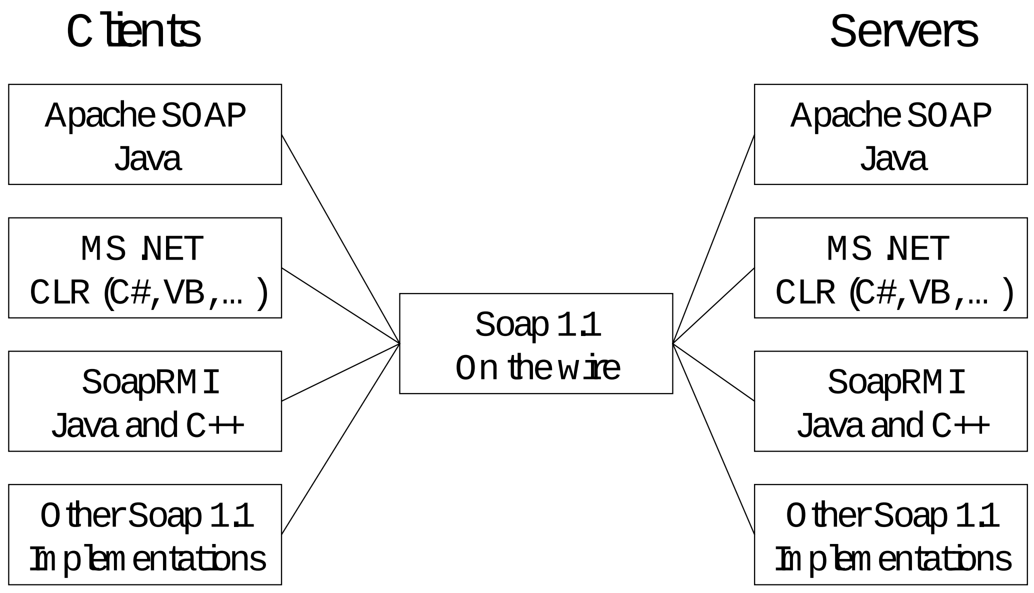

Figure 10: SoapRMI can interoperate with Microsoft .NET and Apache SOAP

We wanted to find a powerful and useful subset that would also be efficient for high performance scientific computing. Furthermore, the interoperability should extend to other SOAP frameworks like Apache SOAP and .NET.

To achieve interoperability, SoapRMI-Java and SoapRMI-C++ have an *a priori* agreement on the namespaces used and the interfaces they correspond to. Since the SOAP protocol specifies the wire representation to be in ASCII, no endian (Big Endian or Little Endian) conversion is necessary while sending or receiving a call.

Figure 11 shows the representation of an exception on the .NET framework. Note that the packaging of the exception is different from the way it is marshalled by SoapRMI. In particular the *detail* element is a sub-element called *exceptionType*. We are considering transforming SoapRMI wire exception format to be compatible with the .NET framework.

Aapche SOAP implementation is compliant with SOAP 1.1 specification. Since SoapRMI 1.1 calls on the wire also conform to the same specification, it is possible to interoperate between the two systems. Both Apache SOAP and SoapRMI-C++ support recursive object serialization. SoapRMI-Java switches to recursive serialiation from mult-ref serialization mode to interact with Apache SOAP.

11 Further Work

SOAP is a simple protocol and lends itself to robust implementation. While SoapRMI provides a protocol based on almost universally available communications protocol (HTTP) and data representation (XML), it is ASCII-based and hence has to send more data on the wire than a protocol with binary data format. We will perform experiments to determine the exact performance penalty imposed by using SoapRMI C++, helping scientific component framework developers to know when a SoapRMI call should be used to convey actual application

```
<SOAP-ENV:Fault id="ref-1">
<faultcode id="ref-2">SOAP-ENV:Server</faultcode>
<faultstring id="ref-3">Exception thrown on Server</faultstring>
<detail xsi:type="a1:ServerFault">
<exceptionType id="ref-4">System.Exception, mscorlib</exceptionType>
<message id="ref-5">Missing SOAPAction header.</message>
<stackTrace id="ref-6"> at System.Runtime.Remoting.Channels.
             HTTP.HTTPChannel.GetChannelHeaders
             (HTTPChannelRequest request)
           at System.Runtime.Remoting.Channels.
             HTTP.HTTPChannel.ProcessRequestResponse_
             SOAP(HTTPChannelRequest request,
             HTTPChannelResponse response, Boolean&#38;fIsOneWay)</stackTrace>
</detail>
</SOAP-ENV:Fault>
```

Figure 11: Example of a Remote Exception in .NET.

data, and when it should be used to negotiate a higher-performance communications protocol for that data. In any case, SoapRMI provides an ultimate fall-back for communications because it depends on minimal standards which are widely supported. Earlier performance tests for Java-only version of SoapRMI [18] helped us delineate the bottlenecks inherent in the SOAP protocol from the ones that could be removed with better implementation, leading to the current version.

We are currently integrating SoapRMI as one of the communication layers in the Common Component Architecture Toolkit (CCAT) [5], which will allow us to test it in realistic scientific application settings. We also plan to further explore the design and implementation issues in integrating SSL [25] into the transport layer to facilitate secure exchange of data using SOAP.

12 Conclusions

The design issues underlying SoapRMI 1.1 were explored, and it has been implemented in both C++ and Java in a way conformant with SUN's Java RMI API. The different types of XML processors that are available today were described, including XPP, a small and efficient pull parser that we designed and developed for applications that need to deserialize XML into object instances.

SoapRMI has extensive support for remote exception handling. This feature allows a user program in C++ and Java to catch exceptions that are thrown by the server on a remote process. SoapRMI-C++ has a templated model to do recursive serialization of objects into SOAP while SoapRMI-Java supports multi-ref serialization in addition to the recursive serialization style.

SoapRMI interoperates not only between our C++ and Java versions, but also with

Apache SOAP and the .NET framework.

References

[1] Apache SOAP, visited 02-05-01. http://xml.apache.org.

[2] Open LDAP, visited 02-25-01. http://www.openldap.org.

[3] Albatross project, 1998. http://www.cs.vu.nl/albatross.

[4] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation*, August 1999.

[5] Randall Bramley, Kenneth Chiu, Shridhar Diwan, Dennis Gannon, Madhusudhan Govindaraju, Nirmal Mukhi, Benjamin Temko, and Madhuri Yechuri. A component based services architecture for building distributed applications. In *Proceedings of Ninth IEEE International Symposium on High Performance Distributed Computing Conference, Pittsburgh*, August 1-4 2000.

[6] Fabian Breg, Shridhar Diwan, Juan Villacis, Jayashree Balasubramanian, Esra Akman, and Dennis Gannon. Java RMI performance and object model interoperability: Experiments with Java/HPC++. *Concurrency and Experience*, 1998. Presented at 1998 ACM Workshop on Java for High-Performance Network Computing.

[7] Fabian Breg and Dennis Gannon. Compiler support for an rmi implementation using nexusjava. Technical Report 500, Extreme Computing Lab, Indiana University, Bloomington, Indiana, 1997.

[8] World Wide Web Consortium. XML-Schemas, visited 2-17-00. http://www.w3.org:80/TR/NOTE-xml-schema-req.

[9] World Wide Web consortium. Document Object Model, visited 7-15-99. http://www.w3c.org/DOM.

[10] World Wide Web Consortium. XML, visited 7-20-99. http://www.xml.org.

[11] J. M. Duftler, S. Weerawarana, and F. Curbera. SOAP For Java, visited 7-01-00. http://www.alphaworks.ibm.com/tech/soap4f.

[12] D. Box et al. Simple Object Access Protocol 1.1. Technical report, W3C, 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[13] D. Megginson et al. SAX 2.0: The Simple API for XML, visited 07-01-00. www.megginson.com/SAX/.

[14] Tim Bray et al. Extensible Markup Language (XML) 1.0 (Second Edition 6 October 2000), visited 03-01-01. www.w3.org/TR/2000/REC-xml-20001006.

[15] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multi-threading and Communication. *J. Parallel and Distributed Computing*, 37:70–82, 1996.

[16] D. Gannon, P. Beckman, E. Johnson, and T. Green. *Compilation Issues on Distributed Memory Systems*, chapter 3 HPC++ and the HPC++Lib Toolkit. Springer-Verlag, 1997.

[17] ISO/IEC 14750 — ITU-T Rec. X.920. Corba IDL Standard, visited 03-01-01. www.omg.org/corba/standards.htm.

[18] Madhusudhan Govindaraju, Aleksander Slominski, Venkatesh Choppella, Randall Bramley, Dennis Gannon. Requirements for and Evaluation of RMI Protocols for Scientific Computing. In *Proceedings of SuperComputing 2000, Dallas TX, 2000*, November 2000.

[19] Microsoft. .NET framework, visited 02-10-01. http://www.microsoft.com/net/.

[20] Microsoft. COM, visited 4-1-2000. http://www.microsoft.com/com.

[21] SUN Microsystems. RMI over IIOP, visited 02-15-01. http://java.sun.com/products/rmi-iiop/.

[22] Sun Microsystems. Java Remote Method Invocation, visited 07-01-00. http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html.

[23] SUN Microsystems. JNDI, visited 3-7-2000. http://java.sun.com/products/jndi/.

[24] SUN Microsystems. Java Beans, visited 4-15-00. http://java.sun.com/beans/.

[25] Network Working Group. The transport layer security protocol, visited 03-01-01. ftp://ftp.isi.edu/in-notes/rfc2246.txt.

[26] Ninja project, 1998. http://ninja.cs.berkeley.edu/.

[27] OMG. Corba Component Model, visited 1-11-2000. http://www.omg.org/cgi-bin/doc?orbos/97-06-12.

[28] Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *to appear in Concurrency: Practice and Experience*, 2000.

[29] A. Slominski. Design of a Pull and Push Parser System for Streaming XML. Technical Report TR-550, Indiana University, April 2001.

[30] SUN Microsystems, Inc. Dynamic proxy classes, visited 03-01-01. java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html.

[31] George K. Thiruvathukal, Lovely S. Thomas, and Andy T. Korczynski. Reflective remote method invocation. *Concurrency: Practice and Experience, 10(11-13):911-926*, 1998.

[32] R. Veldema, R. van Nieuwpoort, J. Maassen, H.E. Bal, and A. Plaat. Efficient Remote Method Invocation. Technical Report IR-450, Vrije Universiteit, Amsterdam, sep 1998.

[33] Juan Villacis, Madhusudhan Govindaraju, David Stern, Andrew Whitaker, Fabian Breg, Prafulla Deuskar, Benjamin Temko, Dennis Gannon, and Randall Bramley. CAT: A high performance, distributed component architecture toolkit for the grid. In *Proceedings of Eighth IEEE International Symposium on High Performance Distributed Computing Conference, Redondo Beach, California*, August 3-6 1999.

[34] M. Welsh. Ninjarmi, 1998. http://www.cs.berkeley.edu/~mdw/proj/ninja/.