RPC Package for Java

Prasad Deshpande
pmd@cs.wisc.edu

Avinash Sodani
sodani@cs.wisc.edu

Abstract

Java is an object-oriented and interpreted language. Java object code is machine independent and can be run by an interpreter running on any machine. Applets written in Java language can thus be downloaded and executed on any machine. We have provided an RPC package for Java so that Java applications can make remote calls to RPC servers. We have also provided functions so that both RPC clients and servers can be programmed in Java. We have used the underlying Sun RPC to implement RPC in Java.

1 Introduction

This is an age of World Wide Web (WWW). Among the several WWW browsers available taday, **Hotjava** is among the most promising ones. Hotjava, unlike other WWW browsers, can support *dynamic objects* or **applets**. Applets are pieces of code that the browser can download from the remote site and execute [1] on the local machine. This capability of being able to fetch and execute code dynamically, enhances the power of WWW enormously. With this, the web can be used in ways not possible before. There can be elaborate animation on web pages. The web pages can be made more interactive (e.g. games etc.). All these can be put to some good practical use e.g. advertisement, education, online product preview etc.

The applets, which form the core of Hotjava, are written in **Java**. Java is a C++ style, object oriented language. A program written is Java is compiled into a architecture independent object code which is then interpreted by a Java interpreter. The language is so designed that a compiled Java code can have no malicious effect. This feature is mandatory to ensure the innocuousness of the applets downloaded from the remote site.

Despite of all the good features in Java and Hotjava, they lack support from remote procedure calls (rpc). This limits their capability to some extent. For example, Java application or Hotjava applet cannot contact a database server through procedure calls. **In this report we present design and implementation of RPC package for Java** .

The section 2 lists the possible approaches, section 3 explains the design and various design issues, section 4 deals with details of the implementation, and section 5 provides the conclusions. The Java RPC interface and Xdr interface are described in the appendix.

2 Approach

There could be two ways of implementing RPC package for Java. They are,

[1]The word "execute" in context of *applets* means "interpreted".

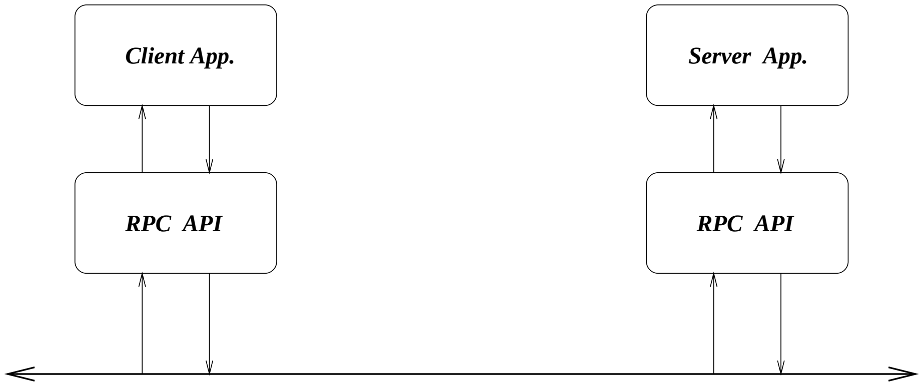

Figure 1: RPC

1. Using Sun RPC package underneath. The Java RPC functions act as wrapper routines which in turn call the RPC library provided by system. This is possible since Java has support for native functions written in C. Thus Java functions call C functions which then use the RPC library.

2. Implementing the entire package in Java. The RPC protocol could be directly implemented in Java. Java provides a socket interface which could be used to implement higher level RPC protocols.

There are some intricacies in the first approach. The interface between Java and C is not very easy to use. One has to be careful about the correspondence between Java and C data types. Besides Java does not support explicit pointers. There are no function pointers, so arguments which are functions (like xdr routines) have to be made into objects and passed. However the first approach avoids unnecessary coding of the entire RPC protocol in Java. Hence we chose the first approach. The only disadvantage is that it depends on the system RPC library. Thus it may not be possible to use this code on machines such running Windows NT which do not yet have Sun RPC. But the solution to this is to port Sun RPC on NT.

3 Design

A high level representation of how RPC works is shown in Figure 1. The client makes the procedure call which goes through the layer labelled "RPC API" (possibly through stubs). The request is send over to the server node. The RPC API at the other end invokes the appropriate procedure and returns the results back to the client.

To provide RPC support for the applications written in Java, we conceptually added another layer of API, called "Java RPC API", as shown in Figure 2. The main idea is, for a Java procedure to be called remotely it needs to be registered with the Java RPC API. The system RPC communicates with the Java RPC API to call the Java function.

This is a very simplified view of the whole thing. In reality, the interaction between the Java layer and system layer is not so straight forward. Before we describe the system in more detail, there are couple of things that needs to be known about the Java RPC API that we provide. The various functions provided are listed in the appendix. Apart from them we provide two *abstract* Java classes : *servfunc* and *func*. A Java function that need to be called remotely has to be named "func" and should be a part of a Java class

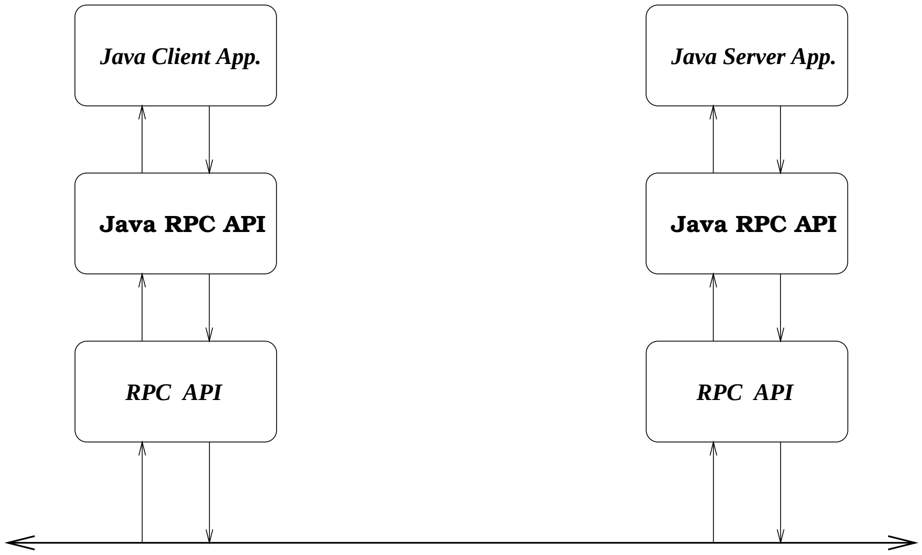

Figure 2: Support for RPC in Java

that *extends* [2] the abstract class servfunc. All the filter routines should be named "xdr_func" and should be part of Java class that extends the abstract class func.

Now, we proceed to explain the Java RPC API design in more detail. First, we open up the Java RPC API layer to reveal more detail. This exposed layer is shown in Figure 3. It shows two extra blocks on each side : *func pool* and *xdr func pool*. The purpose of these blocks will become clear as proceed with the explanation.

Registering Service : For resistering a function, Java application calls a function provided by Java RPC API (e.g. registerrpc). This call serves two purposes :

- It registers the Java function with the Java RPC API. This involves storing the pointers[3] to the Java function and the filter routines in a Java RPC API datastructure.
- It allocates a C function from the func pool (say pfunc()) and registers that function with the system RPC. In a way, this C function is a proxy for the Java function (hence the letter 'p' in the name 'pfunc()'). Corresponding to each filter routine, a C function (say pfilter()) is allocated from the xdr func pool. These C functions are proxy for the corresponding Java filter functions and are used by the underlying RPC system.

Remote procedure call : The sequence of events for a remote procedure call are as follows (Figure 4). A Java client makes a remote function call, which calls a function provided by the Java RPC API (event 1, e.g. callrpc). This allocates C functions for the filters from the xdr func pool (pfilter()). The call is then passed to the underlying system RPC. The System RPC calls the proxy function pfilter() (event 3) to marshall the arguments. This function, in turn, calls the appropriate Java function which performs the required marshalling of arguments (event 4).

On the server side, the system RPC calls the proxy filter function for the procedure being called (event 7). This in turn calls the appropriate Java function which does the unmarshalling of the arguments.

[2] "extending" an abstract class means defining all its member functions

[3] It is actually a pointer to the class containing the function

The system RPC then calls the proxy function (event 9), which, in turn, calls the appropriate Java function (event 10). Finally, the proxy filter for sending back the results is called (event 11). This filter calls the Java filter (event 12) which performs the marshalling of the result before sending it back to the client.

Similar sequence of steps are taken at the client for getting the result back to the Java application.

This design requires lot of calls from Java to C and vice versa. It is the crossing of the C-Java interface where the most of the complexity lies. As Java is a very restrictive language (essential feature needed to make Java programs harmless) mapping between C types and Java types is not straight forward. Some of the noteworthy implementation issues are discussed in the next section.

4 Implementation

In this section we discuss some of the interesting issues that came up during the implementation. We provide two classes as part of the RPC package. One is called **rpcInt** and the other **xdrInt**. The rpcInt class provides an API for client and server RPC calls. The xdrInt class provides the API for the basic XDR routines.

These Java routines are implemented as native functions which are implemented in C. These C routines in turn call functions in the Sun RPC library.

4.1 Correspondence between C and Java Types

The arguments passed to the Java routines are Java types which get exposed in C when these routines are native functions. These get mapped to the corresponding C types. The basic types in Java are boolean, int, float, char, short, ushort, double. Except for float and double all these map to long in C. Float and double map to float in C.

Objects and arrays in Java correspond to handles in C. A handle to an object is a pointer to structure having information both on the methods and data parts of the object. This handle can be used to access the fields within the object. Handles are also needed to make calls to methods on objects from C.

4.2 Interaction between Java and C for implementing RPC

Java doesn't support function pointers. Thus to pass functions as arguments we need to make objects out of it. For some of the C RPC routines it is necessary to pass function pointers. For example consider **callrpc**. This function requires us to pass pointers to the xdr routines. These routines act as filters for writing out the arguments of the remote procedure and reading in the results. They are needed to convert data to and from the external machine independent representation.

Consider the corresponding **callrpc** function provided by our package. The prototype is :

```
Java class : RpcInterface:
static public native int callrpc(String host, int prognum, int
    versnum, int procnum, func Inproc, Object in,func Outproc, Object out)
```

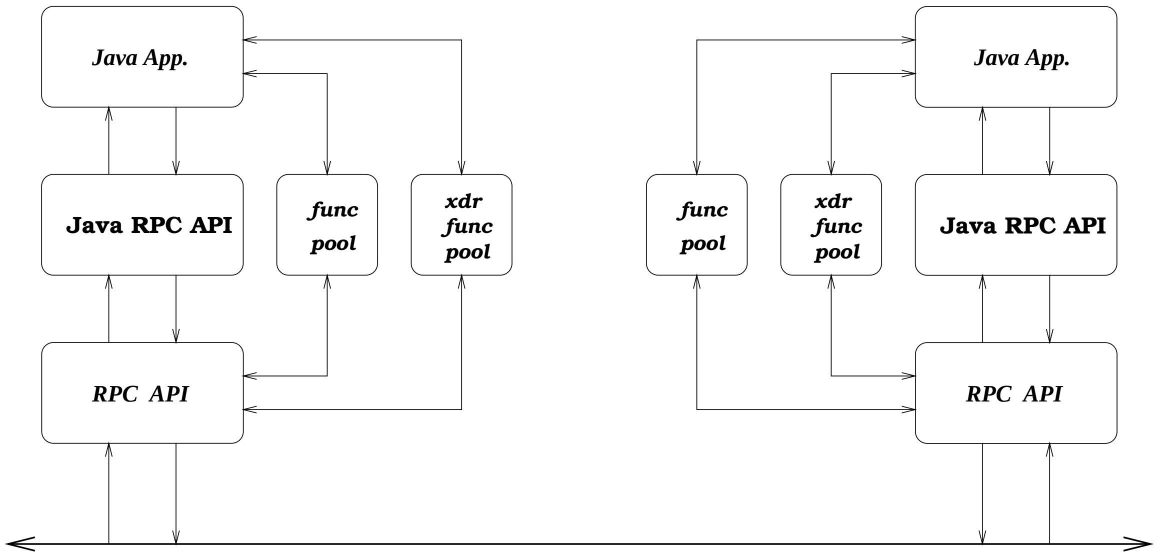

Figure 3: Java RPC API layer exposed

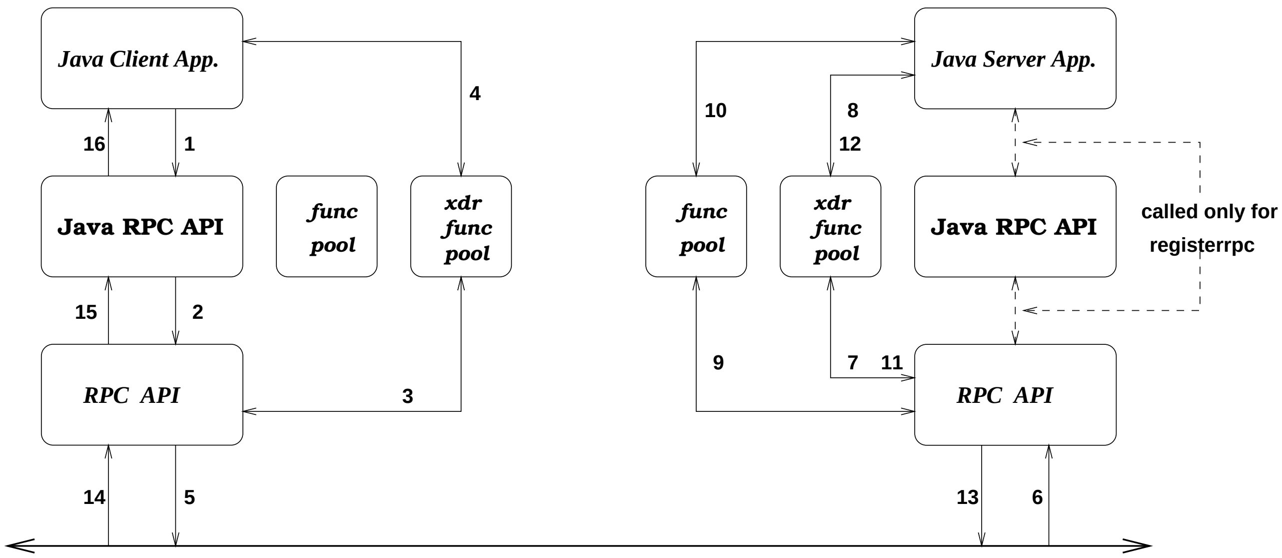

Figure 4: RPC sequence of events

Inproc and Outproc are objects which implement the xdr routines acting as the filters. This is a native function implemented in C in which these objects correspond to handles to those objects. The native function in turn calls the C callrpc function. But we cannot pass these handles directly to the C callrpc since they are not function pointers. This is overcome by writing wrapper C routines (proxy functions) which in turn call these Java xdr routines. The pointers to these wrapper C routines are passed to the callrpc. Thus the native implementation (C code) looks like:

```
HObject *Inproc, *Outproc;
long rpcinterface_callrpc(....., HObject *inproc, ... , HObject *outproc,...)
{
    .....
    Inproc = inproc;
    Outproc = outproc;
    callrpc(...., xdr_in, ... , xdr_out,...);
}

boolean xdr_in(..)
{
    execute_java_dynamic_method(..., Inproc,..);
}

boolean xdr_out(..)
{
    execute_java_dynamic_method(..., Outproc,..);
}
```

*HObject ** denotes a handle to an object. *Execute_java_dynamic_method* calls the specified method of the given object. Here it is used to call the user xdr routine written in Java. Thus there is a sequence of calls from Java to C to Java. Similar things have to be done for some other functions like registerrpc which require functions as arguments.

4.3 XDR Routines

For RPCs the exchange of data between the client and server programs is in an external machine independent format. The Sun RPC library provides xdr routines for basic types such as char, short, int, float as well as for strings, arrays and pointers. xdr routines for complex types are built up using these routines.

We follow a similar approach and provide the basic xdr routines in Java. For complex classes the xdr routines can then be written using these. We use the Sun RPC external data representation standard for our RPC as well. The xdr routines are also implemented as native functions which call the Sun RPC xdr routines. So we don't have to explicitly convert to and from the external data formats. There are a few interesting points though:

4.3.1 Handling indirection in Java

The xdr routines are bidirectional, i.e. they can be used to write out the data or read in the data. In the C xdr routines this is handled by indirection. For example, *xdr_int(int *)* which is the xdr routine for integers expects a pointer to a integer. Thus this function can take in an integer value as well return an integer value by changing the value pointed to by the pointer. But java doesn't support explicit pointers. The indirection there is achieved by using an array. Thus the corresponding function *xdr_int(int [])* in Java expects an array of integers.

4.3.2 Object References

In Java, an object field in an class is actually just a reference to that class and thus equivalent to an pointer. For e.g. consider the declaration:

```
class obj {
    int x;
    obj y;
}
```

The field *y* is a reference to an object of type *obj* and is not initialized until an explicit *new* is done. The C equivalent of this would be :

```
struct obj {
      int x;
      struct obj *y;
}
```

Thus while writing the xdr routine for *obj* the field *y* should be considered as a pointer and an *xdr_pointer* should be used for it. *xdr_pointer* is handled in the same way as in the Sun RPC standard, first a boolean value is put indicating whether the reference is null and then the actual object pointed to is put if the reference is not null.

4.4 Using the RPC package

The java RPC API provides most of the high level and low level RPC functions in our packages. The XDR API provides basic xdr routines for java data types. The entire list is specified in the appendix. Both RPC clients and servers can be developed in Java using these routines. Since we conform to the Sun RPC standards, a Java client can communicate with a C server which uses the same standard and vice versa.

Let us consider the development of xdr routines for Java classes. Consider the class *obj* declared in the previous section. The xdr routine for it will be as shown:

```
class xdr_obj extends func {
    boolean xdr_func(obj P)
    {
        int [] xx = new int[1];
        xx[0] = P.x;
```

```
        if (!xdrInt.xdr_int(xx)) return false;
        P.x = xx[0];

        obj[] yy = new obj[1];
        yy[0] = P.y;
        if (!xdrInt.xdr_pointer(yy)) return false;
        P.y = yy[0];
        return true;
    }
};
```

The use of arrays is necessary for indirection. The above routine is written so that it is bidirectional. *Func* is an abstract class which is used to represent xdr routines. Similarly there is an abstract class *servfunc* which is used to represent functions which the Java server will register. After declaring the xdr routines and the server routines, the user can use RPC calls passing the appropriate arguments as necessary. This is very much similar to the RPC programming in C since our functions correspond one-one with them. More details on using the package is beyond the scope of this report.

5 Summary and Future Work

We have been successful in implementing a RPC package in Java which allows for both high level and low level RPC programming. RPC servers as well as clients can be written in Java. We use the Sun RPC standards so a Java client can call C server using the same standard and vice versa. We observed was that there was a significant difference in execution speeds of RPC applications written in Java and C. This is expected since Java is an interpreted language. An advantage of using Java is that it has automatic garbage collection. Thus we do not have to explicitly free the memory that might be allocated by the Xdr routines.

We have implemented RPC using the first approach of passing the RPC calls to the underlying system RPC calls. Implementing the entire RPC package in Java might be a good idea to make it completely machine independent. Besides if Java clients were communicating only with Java servers then need for external data representation (xdr routines) will be eliminated since Java anyway uses machine independent format.

Another useful thing would be write an **rpcgen** like routine which given the RPC specification file will produce client and server stubs and also the xdr routines in Java.

6 Acknowledgement

We would like to thank Karthik Ramasamy for suggesting this project and helping us with some difficulties.

References

[1] John Bloomer. "Power Programming with RPC", O'Reilly & Associates, Inc.

[2] http://java.sun.com/doc/programmer.html. Documentation on programming in Java.

[3] http://java.sun.com/JDK-beta2/index.html. Documentation for beta release.

[4] http://java.sun.com/1.0alpha3/doc/overview/hotjava/index.html.

[5] SUN Manual.

Appendix

Class RpcInterface :

Client functions:

- static public native int callrpc(String host, int prognum, int versnum, int procnum, func xdr_in, Object in, func xdr_out, Object out);
- static public native int clnt_call(int clnt, int procnum, func xdr_in, Object in, func xdr_out, Object out, timeval timeout);
- static public native int clnt_create(String host, int prognum, int versnum, String nettype);
- static public native void clnt_destroy(int clnt);
- static public native void clnt_pcreateerror(String s);
- static public native void clnt_perrno(int stat);
- static public native void clnt_perror(int clnt, String s);

Server functions:

- static public native void get_myaddress(sockaddr_in addr);
- static public native int getnetname(char netname[], int length);
- static public native int getrpcport(String host, int prognum, int versnum, int proto);
- static public native int registerrpc(int prognum, int versnum, int procnum, servfunc procname, func inproc, func outproc, Object inobj, Object outobj);
- static public native void svc_run();

Class XdrInterface:

- static native boolean xdr_string(String s[], int maxlength);
- static native boolean xdr_byte(byte val[]);
- static native boolean xdr_short(short val[]);
- static native boolean xdr_int(int val[]);
- static native boolean xdr_long(long val[]);
- static native boolean xdr_float(float val[]);
- static native boolean xdr_double(double val[]);
- static native boolean xdr_char(char val[]);

- static native boolean xdr_boolean(boolean val[]);
- static boolean xdr_bytes(byte[] val[], int size[], int maxsize);
- static boolean xdr_opaque(byte[] val);
- static boolean xdr_pointer(Object val[], func xfn, objfunc ofn);
- static boolean xdr_array(Object[] val[], int[] size, int maxsize, func xfn, objfunc ofn);
- boolean xdr_vector(Object[] val, func xfn);