JRPC: Connecting Java Applications with Legacy ONC RPC Servers

Martin Gergeleit

University of Magdeburg
Computer Science Department, Distributed Systems Institute
gergeleit@ivs.cs.uni-magdeburg.de

Abstract. One of the benefits of Java is the ability to write GUI-interfaces for legacy services in a platform independent way. These Java front-ends allow for a seamless integration of these existing services into an intranet or even the whole internet. However, while Java was designed for network connectivity it initially suffered from a lack of supported middleware solutions. The current approach of integrating RMI and CORBA can be considered as an obvious sign for the need of this kind of integrating technology. However, even with this integration on its way, still a very large base of installed RPC systems, namely all services based on ONC (Open Network Computing) RPC, are still not covered. In order to fill this gap RPC for Java (JRPC) has been developed as the first full ONC RPC binding for Java. This paper describes the design and the implementation of this Java binding for ONC RPC and the enhancements to the standard functionality that were required for integrating RPC functionality in Web-based applications.

1 Introduction

Over the last years the Internet has become an important vehicle for corporate computing. Web services are now treated as business-critical systems. By building applications and accessing databases on corporate Intranets as well as the Internet, competitive advantage will be gained. The major issue concerning Internet application development for the Web is whether to build them from zero or to leverage existing technology and applications. Many companies will find more benefit from making the Web an extension of this proven foundation. Concerning Web application development, most IT-managers simply want to keep and extend their existing applications running properly on existing legacy servers. An Internet server along with smart Java Applets as clients will make perfect front-ends to these existing systems.
For accessing a DBMS from Java this problem has already been addressed with the introduction of JDBC. The situation is different for application servers. Servers that provide specific functionality via an RPC-style (Remote Procedure Call [Birell84]) interface are currently much harder to integrate, even if their invocation-oriented client/server computing paradigm is closely related to Java's object-oriented view [Aldrich97]. Standard Java RMI is out of scope for connecting existing servers, as it only provides Java-to-Java connectivity. Of course, the current efforts to establish

CORBA [OMG95] and the underlying IIOP protocol as the common infrastructure for a heterogeneous environment are a great step into a unified object-oriented world [JavaIDL98], but it does not solve the problem of integrating existing non-CORBA compliant servers into Java-based (Web) applications. The activities of a number of organizations, for making OSF/DCE (including its RPC) and also DCOM fully available for Java can be considered as an obvious sign for the need of this kind of integrating technology. But, probably the largest based of installed RPC systems, namely all services based on ONC (Open Network Computing) RPC [Sri95], are still not addressed by any of these approaches. Sometimes it is even not the actual server implementation that has to be preserved, but the existing interfaces and protocols described by an XDR specification. In other cases one might just want to have Java code, that can read and write XDR-streams, as other (C-) applications often use this platform independent encoding format for serializing data. In order to fill this obvious gap RPC for Java (JRPC) has been developed by the author as the first full ONC RPC binding for Java. It comprises a set of tools and libraries that enables Web application designers to create ONC RPC clients and servers in pure Java. A commercial implementation of ONC RPC for Java derived from JRPC is available from Distinct Corporation [Distinct98].
In the next section this papers shortly summarizes ONC RPC in general and explains the basics of building distributed client/server applications with RPC. Section 3 discusses the design of the Java binding for ONC RPC and the enhancements to the standard functionality that were required for integrating RPC functionality in Web-based applications. Section 4 describes in more detail implementation issues, presents performance figures and gives a short application example. Finally, section 5 concludes the paper with a summary.

2 ONC RPC

ONC RPC (formerly called Sun-RPC) and the related platform-independent data encoding standard XDR (eXternal Data Representation)[Sri95a, Sri95b, Sri95c] have been introduced by Sun Microsystems more than 10 years ago. It is now available for nearly any kind of computing systems, ranging from mainframes to embedded computers. Moreover ONC RPC has become part of the standard distribution of nearly any Unix-workstation. Some very well known services like NFS (Network File System) and NIS (Network Information System) are based on ONC RPC communication. The success of ONC RPC results from its easy paradigm, its simple C-like interface definition language, and its easily portable lightweight infrastructure that makes all heterogeneity transparent to the application programmer. Over the last decade ONC RPC has been used for many client/server application developments from nearly all problem domains.
The major advantage that has made RPC so popular compared to all other communication paradigms is the fact that it is nearly transparent to the application programmer and it provides the same synchronous invocation semantics as a local procedure call (or in the object-oriented case as a local method invocation). The RPC communication paradigm has been adopted by a large number of middleware solutions like ONC

RPC, OSF/DCE RPC, Java RMI or CORBA. When a client wants to use server functionality it simply calls a stub-procedure (or a "proxy") that has the same prototype (name and parameters) as the server procedure. The stub then encodes the call (basically an identifier for the desired procedure and the parameters) into a message using a platform independent data encoding format and uses the RPC runtime infrastructure to transfer the message to the server node. On that node the server RPC runtime receives the call message and passes it to the server stub that decode the call and its parameter. Finally, it calls the server implementing the real functionality. The results of this call are transferred back to the client using the same mechanisms and components in the opposite direction.

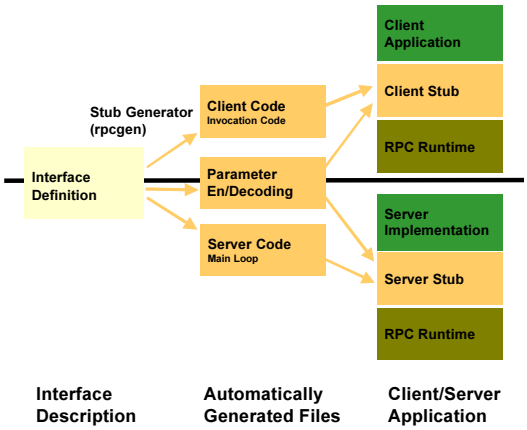

Fig. 1. Steps for building a client/server application with RPC

The client and the server stub are usually not hand-coded but automatically created from an interface definition by a special compiler, often called stub generator. The interface definition is written in a formal language (the IDL - Interface Definition Language, usually C/C++ alike) and it describes the data-types and function-prototypes used for interfacing the server. Given a server, its according IDL file and the RPC infrastructure the only thing one has to do to write a client application, is to run the stub generator on the IDL file, to write the client itself and link the pieces together. Fig. 1 shows these steps for building a client/server application and depicts the involved components.

3 Design of JRPC

In order to adapt an RPC system to a new platform and even to a new language, a new mapping between the IDL and the stubs in the destination language has to be defined. The mapping is then implemented by the stub generator. In addition, the RPC runtime library has to be ported to the new language/system environment. Consequently, JRPC consists of:

- The “jrpcgen” RPC IDL compiler for Java: jrpcgen translates standard RPC/XDR interface definition files into the sources of Java classes that implement the client and the server stubs as well as the XDR conversions for the defined data-structures.
- The JRPC package: The package contains the ONC RPC runtime library. Its API consists of classes that allow writing pure Java clients for standard RPC servers that can be embedded in applets and run by a standard Web-browser. It allows also for writing stand-alone ONC RPC server in pure Java. In addition to the original C-library JRPC supports not only TCP and UDP as transport protocols but it also supports a tunneling mode were RPC data is encapsulated in HTTP transactions.

Table 1. JRPC’s type mapping from XDR to Java

XDR Type	Java Type
(unsigned) int	int
(unsigned) long	int
(unsigned) short	short
(unsigned) char	char
(unsigned) hyper	long
Float	float
Double	double
Bool	Boolean
String	String
Opaque	byte array
fixed length array	Java array
variable length array	Java array
optional data (pointer-like *x)	reference to an object of class x. If x is a basic type a special wrapper class XDRx is used instead
enum x	class x with member variable int value and one constant int per enum constant
struct x	class x with member variables for each struct member
union x	class x with member variables for each union member including discriminant. No overlaying of members is supported (neither for type conversion nor for saving space)
typedef x y	class x with member variable “value” of the redefined type y

3.1 The XDR to Java Type-Mapping

Besides the adaptation of the runtime library, the most important step in bringing ONC RPC to Java is the definition of a mapping of the XDR data-types to Java types and classes. While JRPC has been developed concurrently and independently from other middleware solutions like JavaIDL [JavaIDL98], it is interesting to see that the proposed solutions to the basic problem of expressing the C-style types is very similar (see also [Jain97]).
In many cases, like for the basic types and the C-like *struct*, the mapping is obvious even if in some cases the range of the data-types is different. Because Java has only signed integers, the handling of *unsigned* XDR types is critical. Like in JavaIDL the Java implementation has to insure that possibly resulting negative numbers are handled in the right way. The other possible solution, mapping unsigned types to the next ”bigger” Java type (e.g. Java’s *long* covers the complete range of XDR’s *unsigned*

int), has been abandoned as it does not solve the problem of different ranges in general and it introduces more overhead in the standard case.
XDR *struct*s and *union*s can be mapped directly to Java classes with member variables for each component including *union´*s discriminate. For *union*s no overlaying of members is supported, neither for type conversion nor for saving space, as both mechanisms are not appropriate in a Java environment.
The most notable difficulty in the Java type-mapping is the handling of *typedef*s and *enum*s. Java has neither a notion of type name aliasing nor an enumeration type. The only viable solution that preserves the type names of the XDR definition is the introduction of a new class. In the Java-mapping XDR *typedef*s and *enum*s are mapped to new classes that have only one data member, called "value". In case of an XDR *typedef* "value" is a variable of the redefined type. In case of an *enum* it is simply an integer and the class additionally defines one integer constant per XDR *enum* constant. Thus, the XDR definition "typedef int natural;" basically becomes "class natural {int value;}; in the Java mapping. A complete overview about the mapping is given in the Table 1.

3.2 JRPC inside Applets

JRPC classes and stubs generated by *jrpcgen* can be used inside a browser in an applet and in a stand-alone Java program without any difference. However, by default all Java-enabled browsers do not allow network connections to other hosts than the one an applet was loaded from. This means, as long as the Web-server and the RPC server are hosted on the same machine no difficulties occur. Only if they live on different sites, the Java security manager becomes a problem. In order to connect an applet running inside a browser to an RPC server on an arbitrary machine, additional actions are required:

- Either the JRPC application has to be installed in the local browser's CLASSPATH (not appropriate for a GUI application that is downloaded on demand, hard to maintain, therefore not recommended),
- or one of the certification/capabilities methods to load the applet as trusted code into a less restricted security domain has to be applied.

If both methods are not viable or if a Java client has to be connect to an RPC server that is protected behind a firewall the HTTP-tunneling protocol can be used.

3.3 HTTP-Tunneling Protocol

Most RPC applications have been designed for Intranets (even though nobody knew this term when ONC RPC was originally invented). This means, the RPC protocol doesn't work well over the Internet. On the internet connections to arbitrary ports (like those established by the RPC protocol) are usually blocked by firewalls for security reasons. This is a general problem of all RPC-like systems and is not limited to JRPC. Even if security of a certain RPC server isn't a major concern, the fact that RPC server ports are not known in advance makes it difficult to configure a firewall accordingly.

If this turns out to be a problem it might help to be able to configure the used ports statically. JRPC supports this additional feature for Java clients and servers. In addition, JRPC also has a powerful mechanism that encapsulates RPC requests in standard HTTP transactions. This mechanism enables execution of arbitrary RPC's despite of intermediate firewalls. It also allows calling RPC servers (from within a Java-applet) that are not located on the same host as the web server but elsewhere in the LAN. With this feature the Web-server and other application- or database-servers can easily be separated on different machines.
HTTP-tunneling is implemented by a special protocol-client ("JRPC.ClientHTTP") and a Servlet [Servlet97] (named *rpcgw*). It runs as an add-on to the Web-server at the server-site. The *rpcgw* Servlet translates and executes RPC's that are encapsulated in HTTP-requests. It uses the standard servlet interface for communication with the Web-server (a Java CGI version has also been implemented). *rpcgw* receives the RPC request encoded in an HTTP POST request. It decodes the request parameter, checks an access control list, creates an RPC connection, and forwards the request to the RPC server via the standard RPC protocol. It then waits for the reply and sends the return parameters encapsulated in an HTTP reply via the Web-server back to the JRPC runtime of original requestor. The *rpcgw* is a generic gateway in the sense that it does not know about the server interfaces and the parameter types. It simply forwards all requests. This means, there is no need to adapt *rpcgw* to a specific RPC server interface.
In order to preserve the integrity and the privacy of the server's domain *rpcgw* uses a fine-grained access control list that allows specifying exactly which procedure of which RPC-program/version number can be executed on which host via which protocol. In addition it allows for logging all RPC activities to a local file. Typically, access control will be configured to be highly restrictive and will allow only access to those services, hosts and procedures that are really required by the application and that cannot be abused by an intruder. With this feature of JRPC, that is beyond a straight port of he ONC RPC protocol, a Web-server and other application-servers can separate on different hosts (however, with a significant performance penalty compared to the direct connection).

4 JRPC Implementation

In Fig. 2. an overview of the class hierarchy in the JRPC package is given. The package mainly consists of classes implementing four functional areas: the generic parts of the client and server stubs, the XDR en/decoding of standard data-types, error handling, and ONC RPC's naming service.

4.1 Client and Server Stubs

The generic parts of the client and server stubs are implemented by the two classes "JRPC.JRPCClient" and "JRPC.JRPCServer". They are providing the user APIs for configuration, initialization, naming and binding as well as the standard synchronous invocation of an RPC. Each client and server stub class generated by *jrpcgen* inherits

its basic functionality from one of these two classes and just extends it with the interface of the user-defined procedures.

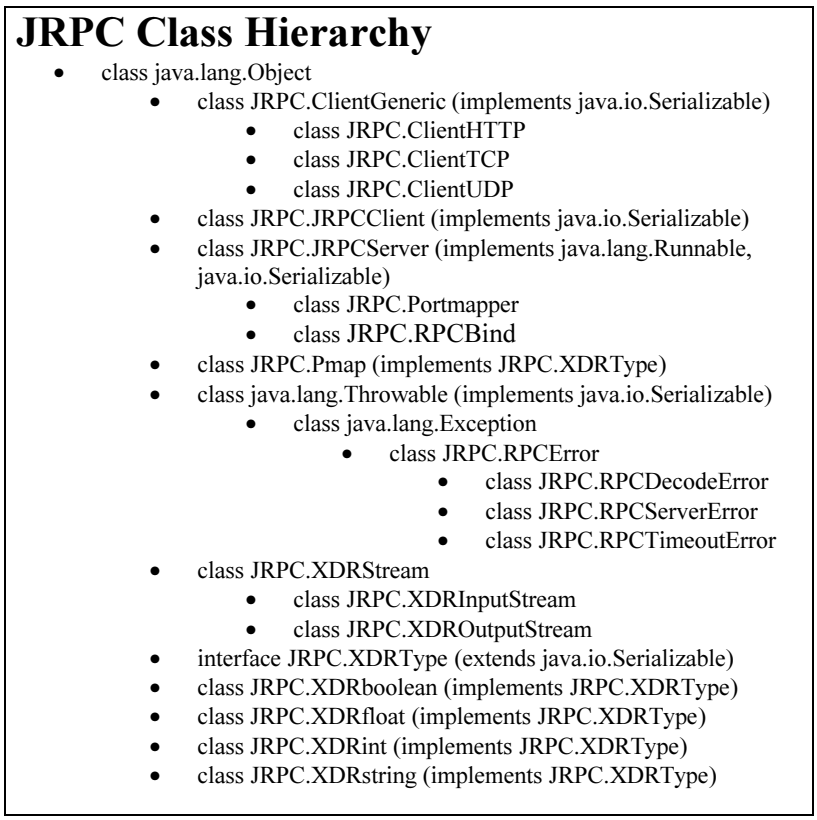

Fig. 2. Basic JRPC class hierarchy (for a complete list see [Distinct98])

Below these higher level interfaces, client and server objects both use additional protocol objects that implement the actual request-reply protocol using the underlying transport layer (at the client site these are objects of subclasses of the public class "JRPC.ClientGeneric"). This two layer approach allows to configure dynamically the protocol used for transmitting RPCs. When extending the JRPC toolkit with the HTTP-tunneling protocol, this mechanism has proofed its flexibility, as it enabled the reuse of existing clients with the new transport without even the need for a recompilation.

At the server site there is typically a one-to-many relation between the higher-level server-object the lower-level protocol objects. Protocol objects are created dynamically, they are active (i.e. they contain a thread and are implementations of the java.lang.Runnable interface), and each active client connection is represented by one object. This can be used to increase the concurrency in the server (see 4.5).

4.2 XDR Types

All classes that implement the “JRPC.XDRType” interface are used for the XDR en/decoding of standard Java data-types. Using these classes, *jrpcgen* builds the XDR code for the elaborated user-defined data-types (like e.g. XDR “struct”s and “union”s). The interface “JRPC.XDRType” itself provides the polymorph coding and encoding methods, that are used at by the generic stub classes to handle invocation parameters of any type. It defines the two methods "xdr_encode()" and "xdr_decode()" that are used by the stub implementation for marshalling and unmarshalling the parameter into and from an "XDRStream" object.

4.3 Error Handling

The enhanced fault model is one of the main reasons that the use of an RPC instead of a local procedure call can never be completely transparent to the calling application. However, Java’s exception handling allows to hide this additional complexity much better than the very basic error number approach in the C binding. JRPC introduces a new class of exception (“JRPC.RPCError”) that can be thrown by any RPC invokation. It is subclassed into server, decoding, and timeout exceptions for more detailed error handling. A server exception is thrown if a server is unavailable during the bind phase, a decoding exception signals inconsistencies occurring during XDR decoding, and a timeout exception obviously indicates a timeout condition in the client while waiting for a server’s reply.

4.4 Naming and Binding

The classes “JRPC.Portmapper” (RPCBIND version 2) and “JRPC.RPCBind” (RPCBIND version 3 and 4) (both subclasses of “JRPC.JRPCServer”) and “JRPC. Pmap” at the client site are implementing the ONC RPC naming service [Sri95c], that is required for locating server port addresses. Like in the original ONC RPC C-binding this naming service itself is implemented as an RPC service (but with a well-known port number in order to avoid the recursive need for a naming service). Usually, naming and binding is hidden in the stubs, but sometimes it is useful for an application to use this interface directly.

4.5 Synchronization

In contrast to the standard C binding the JRPC implementation uses multi-treading. UDP requests and each single TCP connection to an RPC server are handled by separate threads. This introduces the need for synchronizing on global data structures of the server implementation. Currently this is accomplished by a single lock in the server stub that enforces mutual exclusion between invocations of application provided procedure implementations. This reduces server site concurrency but it simplifies server implementation as procedures can be written in strict sequential programming model. However, if performance is the key issue and if increased concurrency

can help to speed up an I/O bound server, there is the option to skip this global lock and to implement more fine-grained synchronization at application level.

4.6 Performance

In order to evaluated the performance of JRPC it was compared with the original RPC 4.0 C-implementation. The round-trip time for an RPC using TCP has been measured with an increasing number of data bytes (0, 100, 1000, 2000, and 20000 bytes in both directions encoded as integers). All measurements have been made on two Celeron 400 Mhz PCs running Windows NT 4.0 connected by a switched 100 Mbit Ethernet. The Java code was executed by the Microsoft Java VM and the C code was compiled with the MS Visual C++ 6.0 compiler.

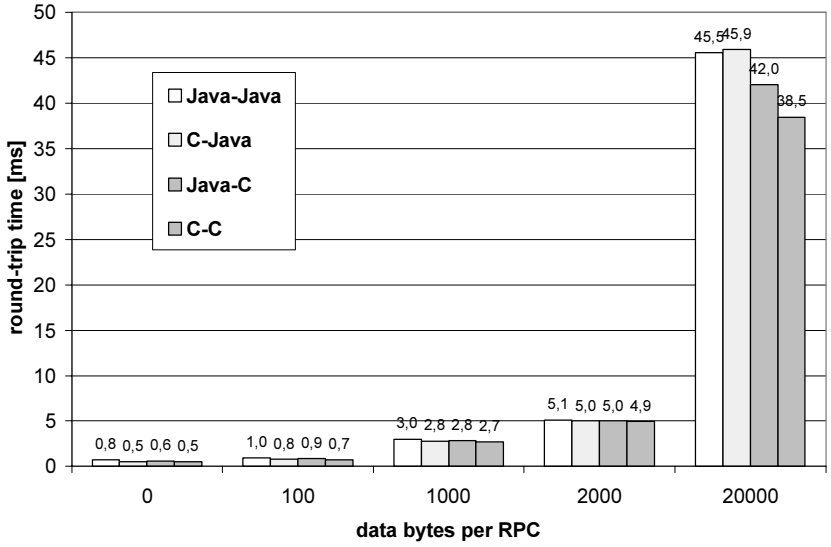

Fig. 3. Performance comparison RPC 4.0 (C) vs. JRPC

Figure 3 shows the performance results for all possible combinations of client and server-implementations. It turns out that for the typical case of a JRPC client and a C-sever the RPC performance is only about 10-20% slower than with a native C-implementation of the client. This is quite good result given that Java is not a compiled language and it will be encouraging for migrating C-applications to Java.

4.7 Example

Finally, we want to look at a short example in order to illustrate the Java language binding. Consider an RPC interface definition file "demo.x" as given in Figure 4. It defines a very simple service that returns a sequence of consecutive lines from a text. This interface definition file contains three type definitions (“struct request”, “struct

result", and "typedef res_list") and a program definition ("server interface DEMO_SERVER"). The program definition contains only one procedure ("get_line()"). In our example the input type of "get_line()" contains two integers specifying a range of lines (named "from" and "to"). The output type "res_list" is a pointer to a linked list of "result" structures. Each element of this list describes one line of the result (line number and content string). In this example it is important to define the "typedef result" because it is a restriction of ONC RPC that a procedure can have only plain type names in its signature (e.g. "result *get_line(request)" would be not allowed). However, as any level of typedefs are allowed this is not really a restriction of semantics.

```
struct request {
          int from;
          int to;
          };

struct result {
          string   line<>;
          struct result *next;
          };
typedef result *res_list;

program DEMO_SERVER {
          version DEMO_VERSION {
            res_list get_line(request) = 1;
          } = 1;
} = 0x20000023;
```

Fig. 4. Example interface definition file "demo.x"

Jrpcgen translates this interface definition into Java stubs (only the client stub is given here). It creates four Java classes (and thus, four files) that implement the client stub for calling the described "demo" service: one file per type definition (named "request.java", "result.java", and "res_list.java") and one client stub file (named like the XDR file: "demo.java").
The generated file "demo.java" is listed in Figure 5. An object of the defined stub class "demo" represents a client to a "demo" server. Without going into all details we can easily see, that this class consists of a number of constants, a constructors, and a public method "get_line_1()" that has the same signature as "get_line()" in the XDR file. Like all client stub classes in JRPC, the class "demo" is derived from the base class "JRPCClient". Similar to the C-binding of ONC RPC, the constants are the program's number and version as well as an ordinal number for each procedure. In order to build a new RPC client the constructor "demo()" simply needs a sever-address and a Boolean that indicates whether TCP or UDP should be used. Actually, *jrpcgen* creates a number of other, more sophisticated constructors, that also allow to configure the client object but they have been omitted here for simplicity. Finally, the "get_line_1()" method is the method we have to invoke when we want to interact with the server. The extension "_1" results from the fact that this is the implementation of

version 1 of this “demo” RPC program (defined in the interface definition file). Like in the C-binding the version number from the XDR file is always appended after an underscore.

```
import JRPC.*;
import java.io.IOException;
import java.net.InetAddress;

/**
 * This class was generated by Jrpcgen from the RPC/XDR file "demo.x"
 * It defines the client interface of the "demo" service.
 */
public class demo extends JRPCClient {

/** Program ID  and Version of the interface. */
public static final int DEMO_SERVER = 0x20000023;
public static final int DEMO_VERSION = 1;

/**
 * Creates and connects an RPC client for the "demo" interface.
 * Calls the Portmapper in order to get the port of the server.
 * @param host       The host on which the server lives.
 * @param stream     true for a TCP connection, false for UDP.
 * @exception        JRPC.RPCError The call failed for any reason.
 */
public demo(InetAddress host, boolean stream) throws
RPCError {
   super(host, DEMO_SERVER, DEMO_VERSION, stream);
}

public static final int get_line = 1;

/**
 * Stub method that invokes the server function
 * "get_line" (version 1).
 */
public res_list get_line_1(request arg) throws
                            RPCError, IOException {
          res_list retval = new res_list();
          GetClient().Call(get_line, arg, retval);
          return retval; }
};
```

Fig. 5. Client stub “demo.java” generated by *jrpcgen*

5 Summary

There is a clear demand for Java tools that allow for interfacing with any kind of legacy services. JRPC is the first tool that addresses one big part of these services,

namely all ONC RPC based servers. It implements a Java-mapping of the ONC RPC interface definition language and it provides the required run-time support for building RPC applications in Java. Together with the enhancement of the HTTP-tunneling protocol it allows to develop fully ONC RPC compatible Web-based applications and front-ends that interact with RPC servers even over Internet WAN connections.
JRPC's run-time components as well as the stub generator *jrpcgen* are written completely in Java. In 1998 JRPC has been transferred into commercially available toolkit [Distinct98], that has been certified as "100% pure Java". In addition to the basic functionality described above the commercial toolkit also fully implements enhanced RPC features that are beyond the scope of this paper, like broadcast RPC, indirect RPC, and Secure RPC (a cryptography based authentication scheme, implemented using the JCE 1.2 API [JCE99]).

6 References

[Aldrich97] Jonathan Aldrich, James Dooley, Scott Mandelsohn, and Adam Rifkin: Providing Easier Access to Remote Objects in Distributed Systems, in the Engineering Client-Server Systems mini-track of the Software Technology Track of the 31th Hawaii International Conference on System Sciences in January, 1998

[Birell84] Andrew D. Birell and Bruce J. Nelson: *Implementing remote procedure calls*, TOCS., 2(1):39-59. ACM., Feb. 1984

[Distinct98] *Distinct ONC RPC/XDR Toolkit for Java, Distinct Corporation*, Product Documentation, Feb 1998, http://www.distinct.com/javarpc/javarpc.htm

[JavaIDL98] *Using CORBA and Java IDL*, Sun Microsystems Jan. 1998 http://java.sun.com/products/jdk/1.2/docs/guide/idl/jidlUsingCORBA.html

[JCE99] *JAVA™ Cryptography Extension 1.2*, Sun Microsystems Apr. 1999, http://java.sun.com/products/jce/

[Jain97] Prashant Jain and Douglas C. Schmidt: Experiences Converting a C++ Communication Software Framework to Java, The C++ Report, Jan. 1997

[OMG95] *The Common Object Request Broker Architecture, Revision 2*, OMG, 1995.

[Servlet97] *The Java™ Servlet API*, Whitepaper, Sun Microsystems Oct. 1997, http://java.sun.com/marketing/collateral/servlets.html

[Sri95a] R. Srinivasan: *RFC 1831 – RPC: Remote Procedure Call Protocol Specification Version 2*, Sun Microsystems, Aug. 1995, http://ds.internic.net/rfc/rfc1831.txt

[Sri95b] R. Srinivasan: *RFC 1832 – RPC: XDR: External Data Representation Standard*, Sun Microsystems, Aug. 1995, http://ds.internic.net/rfc/rfc1832.txt

[Sri95c] R. Srinivasan: *RFC 1833 – Binding Protocols for ONC RPC Version 2*, Sun Microsystems, Aug. 1995, http://ds.internic.net/rfc/rfc1833.txt