Netbula LLC
| Site Map | Contact Us |
| ABOUT US | PRODUCTS | SUPPORT | DOWNLOAD | PURCHASE |

JPRC

Products PowerPRC ONC RPC JRPC Anyboard Anyemail

- Introduction
- White Paper
- JRPC API
- Online Tutorial
- Online Demo
- Download
- RPC Forum

JRPC Programming Tutorial

A step by step illustration on using the JRPC SDK

This is a Hello World example, for more elaborate examples, please download the SDK and look at the code under samples/ directory.

In this tutorial, we illustrate how to build a JRPC client for a simple Msg RPC server/client. This tutorial is not about programming ONC RPC, but about how to use the JRPC tool. If you are not familiar with ONC RPC, the book "Power Programming RPC" from O'Reilly is a good guide.

The Msg server is defined by the following RPC IDL

```
%cat msg.x

program msgserv {
         version MSGSERV_V1 {
                 string sendmsg(string)=2;
      }= 1;

} = 1234567;
```

The interface defines an RPC program with a single procedure **sendmsg**, the client sends a string to the server, and the server returns a string back.

The **C/C++** version of the Msg client/server is available from the Netbula ONC RPC For Win32 SDK.

Now let's build the Msg RPC client in Java(tm).

Step 1. Compile the Msg.x with jrpcgen

At the command prompt, run

```
% jrpcgen msg.x
```

The demo package includes jrpcgen binaries for win32, solaris and linux, they generate identical Java(tm) code.

This would produce the following files:

- msgserv.java The RPC program interface definition, including constant definition such as program number.
- msgserv_cln.java The client stub class. This class implements the RPC interface defined above. An RPC client program instantiate an instance of this class and call its methods (remote call).
- msgserv_svcb.java The RPC service. This class inherits the RPC interface and is abstract, the programmer needs to extend this class and supply the implmentation for the interface.

Normally, jrpcgen would produce four kinds of Java(tm) source code files

- XDT classes for user defined types. These classes can be serialized thorugh XDR streams.
- RPC program interface
- Client stub class
- Server stub class

In our case, there is no XDT classes, because **string** is a built-in type.

Step 2. Code the main client application

This is very easy, we just need to create an instance of the generated msgserv_cln class and call its methods.

```
%cat ClientTest.java


     import netbula.ORPC.*;

         import java.net.*;
         public class ClientTest {
            public ClientTest () {}

        static public void main(String args[]) {

               try {

                 msgserv_cln cl = new msgserv_cln(args[0], "udp");

/*
                   cl.setAuth(new AuthUnix
                        ("localhost", 501, 100, new int[2]));
*/

                   String msg = "hello world\n";

                   System.out.println("sending.. ");

                   for(int i=0; i<5; i++){

                     String reply = cl.sendmsg(msg);

                     System.out.println(
                    "got " + reply +"\n");

                   }

                }catch (rpc_err e) {

                       System.out.println("rpc: " + e.toString());

            }

          }
      }
```

Here, we construct a Msg client which connects to the Msg server on host argv[0] with UDP protocol, send a message, and print out the reply.

Step 3. Compile the client

Make sure that the **netbula.ORPC** package is in the classpath (simply add the orpc.jar file to the **CLASSPATH** environment variable).

Run the source through the Java(tm) compiler:

```
% javac ClientTest.java MSGSERV_1.java
```

This would produce two class files: ClientTest.class and MSGSERV_1.class.

Step 4. Run the Msg client

1. Make sure the Msg server (C version or Java(tm) version) is running on localhost
2. Run the client
   ```
   % java ClientTest
   ```

If the server is running, you should see the client print out the reply from the server, otherwise, it will print out an RPC error: Program not registered.

That is it!

Now, let's build the Msg server in Java(tm)

Step 5 Code the Msg server

The jrpcgen generates msgserv_svcb.java, which defines an abstract class msgserv_svcb with an abstract function **sendmsg**. To fully implement the server, one needs to derive a class which supplies a body for the sendmsg function.

```
    import netbula.ORPC.*;

        class msgsvc extends msgserv_svcb {
        //implement the server function,
                //let's just echo the msg back

            String sendmsg(String msg) {
               System.out.println("got msg from client "+ msg);
                return msg;
            }

        //main function runs the server

            public static void main(String srgv[]) {

               //let's run the server using the run() method in Svc
         //For more flexibility, one could use the TCPServer and UDPServer directly
```

```
                    new msgsvc().run();

                }

 }
```

Step 6 Compile and run the java server

```
%javac msgsvc.java
%java msgsvc
```

***) Implement the Msg server/client in C**

1) Use rpcgen to compile msg.x file into client server stubs. rpcgen is available on unix, rpcgen for win32 is available from Netbula.

2) Code the server implmentation
See the ***cservs*** directory in the JRPC package for sample code.

File Transfer Server/Client with JRPC

Next, we look at a more interesting example, a JRPC server/client that transfer multiple files via RPC mechanism. This example is under **samples/filexfer** directory of the JRPC package.

The Netbula JRPC API has a class named XDTFile, this is a class to serialize a disk file to and from an XDR stream.

The .x file for the file transfer RPC interface is listed below:

```
%import  netbula.ORPC.*;

struct NFiles { XDTFile files<>; };
program FileXFER{
    version v1{
       void xferFile(NFiles)=1;
     } = 1;
} = 12345678;
```

This is a very simple interface, we defined a struct NFiles, which contains a variable length of XDTFile. The xferFile function takes NFiles as an argument, so it can transfer any number of files.

Now, the server code (FileServer.java). The server saves the files (which is done by XDTFile) received and print out a message.

```
import netbula.ORPC.*;
import java.io.*;

public class FileServer extends filexfer_svcb{

        public void xferfile(NFiles in_arg){
     for(int i=0; i<in_arg.files.length; i++) {
                System.out.println("Received file: "+
                in_arg.files[i].receivedFilepath()+
                    " " + in_arg.files[i].byteCount()+
                         " bytes transfered");
                System.out.println("saved file: "+
            in_arg.files[i].savedFilename());
     }
        }

        static public void main(String args[]) {

        rpc_err.debug=true;
        FileServer server = new FileServer();
        try {
                 server.run();
                 System.out.println("server exited");
        }catch(rpc_err e) {
                    System.out.println("Fail to run server:"+e.toString());
        }
    }


}
```

The client code (FileClient.java). The client sends the files listed on the command line to the server.

```
import netbula.ORPC.*;

public class FileClient{


    static public void main(String args[]) {
        try {
             if(args == null || args.length < 2) {
                System.out.println("Usage: java fileClient server_hostname file1 [file2 file3 ..]");
                System.exit(1);
             }
       String servhost = args[0];

             /* use TCP, UDP is not reliable */
       filexfer_cln cl = new filexfer_cln(servhost, "tcp");

       System.out.println("Connected to  " +servhost);

        /* send the files listed on command line args to the server */

        NFiles nf = new NFiles();
        nf.files = new XDTFile[args.length -1];
             for(int i=0; i< args.length-1; i++)
                nf.files[i] = new XDTFile(args[i+1]);

        cl.xferfile(nf);//send all the files over

             for(int i=0; i< args.length-1; i++)
                System.out.println(args[i+1]+ " "+ nf.files[i].byteCount()+ " bytes sent");


       }catch (Exception e) {
             System.out.println("rpc: " + e.toString());
             e.printStackTrace();
       }


    }

}
```

You will probably agree that the above is a small amount of code which does some useful work, transfer any number of files to another machine.