PROTOTYPE OF A SIMPLE ZEROMQ-BASED RPC IN REPLACEMENT OF CORBA IN NOMAD

Y. Le Goc*, F. Cecillon, C. Cocho, A. Elaazzouzi, J. Locatelli, P. Mutti, H. Ortiz, J. Ratel,
Institut Laue-Langevin, Grenoble, France

Abstract

The Nomad instrument control software of the Institut Laue-Langevin (ILL) is a client server application. The communication between the server and its clients is performed with CORBA, which has now major drawbacks like the lack of support and a slow or non-existing evolution. The present paper describes the implementation of the recent and promising ZeroMQ technology in replacement to CORBA. We present the prototype of a simple RPC built on top of ZeroMQ and the performant Google Protocol Buffers serialization tool, to which we add a remote method dispatch layer. The final project will also provide an IDL compiler restricted to a subset of the language so that only minor modifications to our existing IDL interfaces and class implementations will have to be made to replace the communication layer in NOMAD.

INTRODUCTION

Nomad has been designed 10 years ago as a client server application where the server is written in C++ to have a direct access to the C driver layer and the main client is written in Java to have a portable and reactive GUI application. Some other client applications have been developed and, among them, we can cite the Nomad Web Spy and the Nomad Monitor [1]. At the time of the initial design, CORBA [2] was the best middleware to provide interoperability between a wide range of languages and systems and it was logically chosen for the communication between the server and the clients in Nomad. The Nomad server currently is based on omniORB and the clients on JacORB.

The CORBA standard offers lots of features and specifies the implementation of an Object Request Broker (ORB). The project was ambitious and enabled to write distributed applications where network issues could be "forgotten". Indeed the programmer can manage CORBA objects almost like normal objects and write complex client server interactions. These advantages can become disadvantages because this can increase the network communications as the server can become a client and vice versa. The CORBA component is a monolith which offers so much possibilities that when a problem occurs, it is difficult to know where to begin. From our own experience we had trouble with the C++ binding. For example we could not find any documentation on how to ensure that a CORBA object was still alive at the end of a client call. Moreover from the design point of view, CORBA is intrusive as any object managed by the CORBA framework must inherit a CORBA generated class – stub for the client side, and skeleton for the server side. This constraint does not help to design clear applications where a large part of code can depend on CORBA, although a better design is to restrict the use of CORBA to the communication layer. Some other shortcomings are exposed in Ref. [3].

In Nomad we need to have a strong relation between the server and the GUI client. We have numerous IDL operations defined in many files. On one hand, the client to server operations mainly include the hierarchical data model, e.g. the controllers and drivers hierarchy and the composite control sequence building [1]. These request messages must be synchronous as the client is expecting a response result. On the other hand, the Nomad server to GUI client operations mainly include the execution states for the sequencer and the commands of the controllers and drivers. They are called events and some of them could be asynchronous but with reception guarantee. Only a few of them could be asynchronous without reception guarantee. Notice that a simplification of the CORBA operations design could be considered. Indeed once the CORBA architecture is set up, it is very easy to add a new operation.

CORBA implementations are now declining and a bunch of new technologies emerged in recent years as the need for large distributed systems is increasing and CORBA did not manage to impose its architecture. Nomad contains many C++ and Java implementations of abstract methods defined from IDL files. However the Nomad application has not a large number of clients. In that conditions, how can we replace CORBA in Nomad with the minimum amount of work?

EXISTING SOLUTIONS

The existing middleware solutions can be classified into message-oriented, data-oriented, service-oriented and object-oriented. CORBA is an object-oriented middleware. By simplifying, data-oriented and service-oriented middlewares are built on top of a message layer and object-oriented middlewares are built on top of a service layer.

The closest solution to CORBA is Ice [4] developed by Michi Henning, a CORBA expert who rewrote an ORB implementation by taking into account all the drawbacks of CORBA [5]. The Slice language is very close to CORBA IDL which would imply a minimum code porting. However Ice has still a monolith architecture,

*legoc@ill.fr

ISBN 978-3-95450-139-7

does not seem to have a large community and it is not a standard. Replacing CORBA by Ice would introduce a dependency to a single company.

Currently there is no other object-oriented middleware available. Other solutions are exposed in Ref. [3] among which ZeroMQ [6] is considered the best. ZeroMQ has been already chosen by CERN and ESRF for TANGO [7]. ZeroMQ benefits from a large active community, it is multi-language mainly based on a C library bound to different languages. We can find the project jeroMQ [8] that is the rewriting of the C library in Java. This project is useful since some Java contexts do not permit the binding to a C library (Java Web Start, Androïd, etc.). Moreover ZeroMQ has good performances. In the following, we will consider the use of ZeroMQ in replacement of CORBA in Nomad.

PROTOTYPE SOLUTION

ZeroMQ provides an "intelligent" message layer but cannot replace immediately CORBA in Nomad. One solution would be to transform all the IDL operations into ZeroMQ messages but the number of operations and the amount of code to rewrite to define synchronous messages discards this solution. CORBA standard is very large and we only use a small part in Nomad. Thus we can imagine to write a simple ORB based on ZeroMQ that implements a restriction of the CORBA functionalities. The restrictions include a subset of the CORBA IDL language and a simple API to associate server objects to client proxies. As a consequence we will be able to reuse our class design and minimize the code porting.

Object-oriented services	**zRI**
Marshalling/Unmarshalling	Protocol Buffers
Message transport	ZeroMQ

Figure 1: The ORB layers in zRI.

To realize the prototype we need a performant multi-language marshalling/unmarshalling library that will be hidden from the user. We can cite Message Pack [9], a library based on binary JSON messages, Thrift [10] a library provided by Facebook and Protocol Buffers [11] a library from Google. By comparing the benchmarks [12], we chose Protocol Buffers for the large community and its performance. As the marshalling/unmarshalling process will be hidden from the user, the lack of readibility and extensibility cited as disadvantages [13] are not a problem.To summarize, we define the prototype project zRI (ZeroMQ Remote Invocation) that is the third layer of a simple ORB (See Figure 1).

We write a simple object layer that is close to a service-oriented solution – we do not allow the references to objects in our restriction of the CORBA IDL language. The project can be seen as the implementation of synchronous typed messages and an automation of the marshalling/unmarshalling code which can be very verbose. Moreover it depends on relatively small components that makes them easier to replace.

The project includes:

- zRIg : an IDL compiler (restricted);
- zRIcpp : the runtime library for C++;
- zRIj : the runtime library for Java.

The IDL compiler is designed to be extensible, so that the zRI project may not be restricted to the ILL and could be provided to the community as an open-source project. We implement the prototype to test the viability of the solution.

IMPLEMENTATION

In this section we will present the execution of a remote method call in zRI and then present the compiler implementation.

Execution

We take the minimal IDL interface A as example:

```
interface A {
        double foo(in short a, in boolean b);
};
```

The compiler zRIg generates the client stub class `A` in Java and the server skeleton class A (abstract) in C++ for which we provide a simple implementation AImpl. Before requesting any remote method call:

- the zRI server is started on the server side with an address;
- the C++ AImpl object is bound to a name;
- the Java A stub object is resolved to the AImpl object by its address and its name.

The Figure 2 shows the sequence diagram of the client synchronous call of a A.foo.

ISBN 978-3-95450-139-7

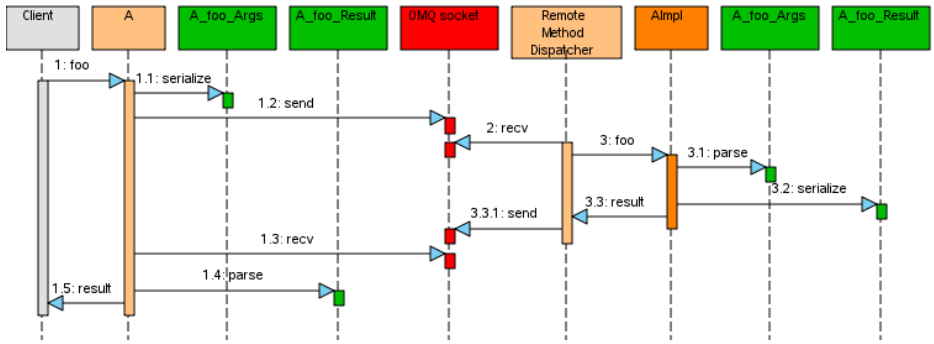

Figure 2: Sequence diagram of a synchronous client call.

When invoking foo, the stub object A opens a socket of type *REQ* (ZeroMQ type) and sends the serialization of a RemoteMethodRequest (not shown) object that contains the object name, the method name and the serialization of a A_foo_Args object containing the arguments a and b. The reception of the message awakes the server RemoteMethodDispatcher object which processes the request by forwarding the call to the real AImpl object after having parsed the content of the message. Note that a socket of type *REP* is open when the server starts. Then the AImpl object parses the string arguments into an A_foo_Args object, calls the real method AImpl::foo and serializes the result with an A_foo_Result object. The RemoteMethodDispatcher object then sends the result message that is parsed by the client object A and returned to the client.

Note that the implementation of the zRI server is not complete, e.g. the process of request in parallel is not implemented yet.

Compilation

The zRIg compiler written in Java generates the stub and skeleton classes as well as the serialization helper classes from the IDL file.

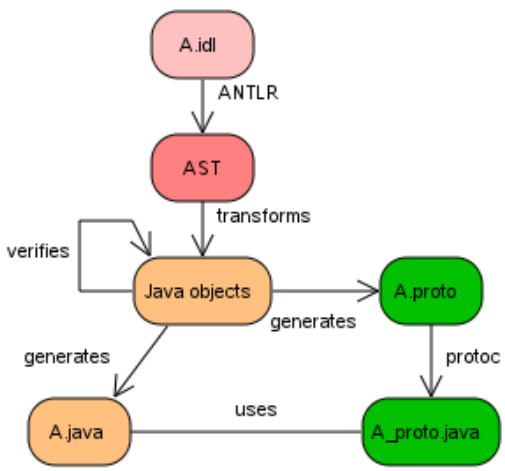

Figure 3: zRIg Compiler workflow for the generation of a Java stub class.

Figure 3 shows an example of the workflow diagram. First, the Abstract Syntax Tree (AST) is built with a parser generated by ANTLR 3 [14] from a CORBA IDL grammar that we found on the web site (slightly modified). The AST is transformed into a Java data structure listing the definitions of the file "A.idl", that is easier to manipulate for the verification and the generation phases. The ANTLR parser checks the lexical and syntaxic errors and we check the semantics errors including name redefinitions, name ambiguities, etc. From the definition data structure, we generate a temporary Protocol Buffers file "A.proto" and "A.java" for the stub class. Then we use the compiler *protoc* provided by Protocol Buffers to compile "A.proto" and generate the helper classes A_foo_Args and A_foo_Result used by the class A to serialize the arguments and parse the result of a method call.

Note that Protocol Buffers defined its own IDL language called proto to describe the messages to marshall and unmarshall. The proto language allows to define message data structures that are composite and typed. We defined a mapping between the IDL types and the proto types so that we can generate the A_foo_Args message with the arguments of foo as attributes, and the A_foo_Result message with the result of foo as attribute.

The zRIg compiler was defined to be easily extended. We based our generation of code on XML code templates easy to modify rather than changing and recompiling Java classes. We currently implemented the following CORBA IDL features: modules, operations with basic types and sequences, typedef. To replace CORBA in Nomad: the following features are missing: struct, interface inheritance, exceptions, preprocessor. New functionalities should be implemented by extending the base class Definition of the zRIg framework.

PERFORMANCE

In Nomad, data transfer can become a bottleneck as we need to transfer from the server quite large acquisition data arrays (> 1 Mo) to be rendered in the GUI at quite high frequencies. We selected the ZeroMQ and Protocol Buffers for their performance. To compare the zRI performance with our current CORBA architecture (omniORB 4.1.4 and JacORB 2.3.1) we defined a simple IDL interface:

```
interface Sequence {
        typedef sequence<double> dseq;
```


ISBN 978-3-95450-139-7

```
        dseq get(in long size);
};
```

We implemented Sequence in both CORBA and zRI so that only a data transfer is made and we compared the time of a client call for different message sizes. The client and the server run on the same machine (Intel Xeon 2.40GHz, 4GB RAM, Linux Suse 11). We present the resulting times in Figure 4.

Size	C++ copy	C++ serial.	Protobuf parsing	Java copy	0MQ + zRI internals	Total zRI	JacORB parsing	Total CORBA
800 KB	0.5	0.5	5	1	3	10	2	5
2 MB	1.5	1.5	12	2	5	20	5	11
8 MB	6.5	7	45	6	10	75	20	45
80 MB	62	72	870	59	62	1125	250	550

Figure 4: Comparison of zRI with CORBA for different array sizes. Time in ms.

The results show that CORBA is almost twice faster than zRI until the size of 8 MB. In zRI, the bottleneck is the Java unmarshalling that takes itself the total time of CORBA. For 80 MB, the Java unmarshalling time is increasing drastically. Note that Protocol Buffers was not designed for large arrays. We also compared Protocol Buffers unmarshalling performance with MessagePack which confirmed to be even almost twice slower. The C++ copy and Java copy occur because the type of the sequence of double in Protocol Buffers is different in zRI C++ and zRI Java. However those results are satisfying for the Nomad requirements.

CONCLUSION

The prototype of a simple ORB built upon ZeroMQ and Protocol Buffers is a success and implementing the required features for replacing CORBA in Nomad should not be an issue. Some real tests will have to be performed. CORBA and ZeroMQ can live together and we will first replace a small part of the Nomad client server communication by zRI.

The zRI project goes beyond the ILL and is a good candidate for open-source release and be opened for other people or organization who either want to replace CORBA or implement new features. An ORB provides interesting features, and it's not because CORBA is declining that the ORB concept is obsolete.

REFERENCES

[1] P. Mutti et al., “Nomad – More than a Simple Sequencer”, ICALEPCS'11, Grenoble, France.

[2] OMG CORBA: http://www.corba.org/

[3] A. Dworak et al. “Middleware trends and Market Leaders 2011”, ICALEPCS'11, Grenoble, France.

[4] ZeroC Ice: http://www.zeroc.com/

[5] M. Henning, “The Rise and Fall of CORBA”, http://queue.acm.org/detail.cfm?id=1142044, 2006

[6] iMatix ZeroMQ: http://www.zeromq.org/

[7] ESRF TANGO http://www.tango-controls.org/

[8] JeroMQ: https://github.com/zeromq/jeromq/

[9] MessagePack: http://www.msgpack.org/

[10] Apache Thrift: http://thrift.apache.org/

[11] Google Protocol Buffers http://code.google.com/p/protobuf/

[12] Thrift Protobuf Compare: http://code.google.com/p/thrift-protobuf-compare/wiki/Benchmarking/

[13] U. Ismail, “A case against using Protobuf for transport in a REST Services”: http://techtraits.com/noproto/

[14] ANTLR: http://www.antlr.org/

ISBN 978-3-95450-139-7