https://doi.org/10.31449/inf.v42i4.1510 Informatica **42** (2018) 555–561 **555**

Analysing RPC and Testing the Performance of Solutions

Sandor Kiraly
Eszterhazy Karoly University, Eger, Eszterhazy ter 1., Hungary
E-mail: kiraly.sandor@uni-eszterhazy.hu

Szilveszter Szekely
Imperial College, Kensington, London SW7 2AZ, UK
E-mail: szekelyszilv@gmail.com

Keywords: remote procedure call, marshalling, Google protocol buffers, JSON-RPC, XML-RPC, performance test

Received: January 29, 2017

In distributed computing, network sockets provide mechanism for a process to establish a remote connection to another process and send messages back and forth. This interface makes possible a proper mechanism that allows a program running as a process on computer A to call a procedure or a function on remote computer B and pass parameters to it. In the case of synchronous Remote Procedure Call (RPC), processes on computer A need to wait for the finishing of execution of procedures on computer B. When the called procedure finishes, produces its result and passes it to the process on computer A that can continue execution. The question is what happens between the time of the remote procedure call and arrival of the returned values and how much the caller must wait for result. Prompted by the release of Protocol Buffers and gRPC by Google, this paper answers that question, describing the structure of third generation RPCs and analysing them putting the focus on performance and the way of marshalling parameters. To facilitate the choice between them this paper represents the results of performance tests carried out by the authors.

Povzetek: Podana je analiza oddaljenih klicev (RPC) v distribuiranih sistemih predvsem v smislu performans.

1 Introduction

While developing computer applications, using procedures and functions is very common. In most cases the subroutines work independently so they could even be run on a remote computer. To reach the remote subroutine (procedure or function) network communication is necessary that is performed via RPC mechanisms. Since the caller and callee procedures run on different machines, they execute them in different address spaces, and different operating system which cause complications. Parameters and results also have to be passed, which can be complicated, especially if the software architectures are not identical or the data structures are complex. Still, most of these can be dealt with, and RPC is a very popular technique that underlies many distributed systems. [1]

To understand the working of RPC it is necessary to examine how local procedure calls are implemented. Before calling a procedure the processor stores the local variables and the state of the caller procedure on the stack while the running of the current procedure will be suspended. To perform the call, the caller pushes the parameters onto the stack in order, last one first. The processor transfers the control to the address determined by the call. In the callee procedure, the compiler is responsible for saving the necessary registers, allocating stack space for local variables, and then restoring the registers and stack prior to the return from the callee. After the procedure has finished running the processor puts the return value in a register, removes the return address, and transfers control back to the caller. The caller then removes the current parameters from the stack, returning it to the original state.

This method cannot be performed if the callee procedure is stored on a remote computer since there are two different running contexts. To solve the problem, another function is used that looks like the remote procedure and it contains code for sending and receiving messages over the network. Its name is stub function. Figure 1 represents the working of remote procedure call for a function pow that returns a long value

More text of the introduction. More text of the introduction. More text of the introduction. More text of the introduction.

The sequence of operations labeled in Figure 1 is as follows:

The client calls a local function (1) that seems to be the actual function but it is the client stub function that serializes the parameters into a message (raw byte stream) (2), and then sends the message to the server machine (3) using socket interfaces. The server stub deserializes the parameters from the raw message (4), and then calls the server function (5) passing it the arguments that it received from the client using the standard calling sequence. After completing the server function, it passes the return value to the server stub (6) that serializes it into a message (7) to

send to the client stub. The message is sent back across the network (8) and the network layer passes the message to the client stub (9) that reads and deserializes it then returns the result to the client function (10).

Figure 1 represents a remote procedure call applying passing parameters by value which is simple since it just copies the value into the network message. Passing by reference is more complex. To enable this technique it is necessary to send a copy of the arguments over, place them in memory on the remote computer, pass a pointer to them to the server function, and finally send the object back to the client, copying it over the reference. For complex structures, it is needed to copy the structure into a pointerless representation, transmit it, and reconstruct the data structure on the remote machine. [2][3][4]

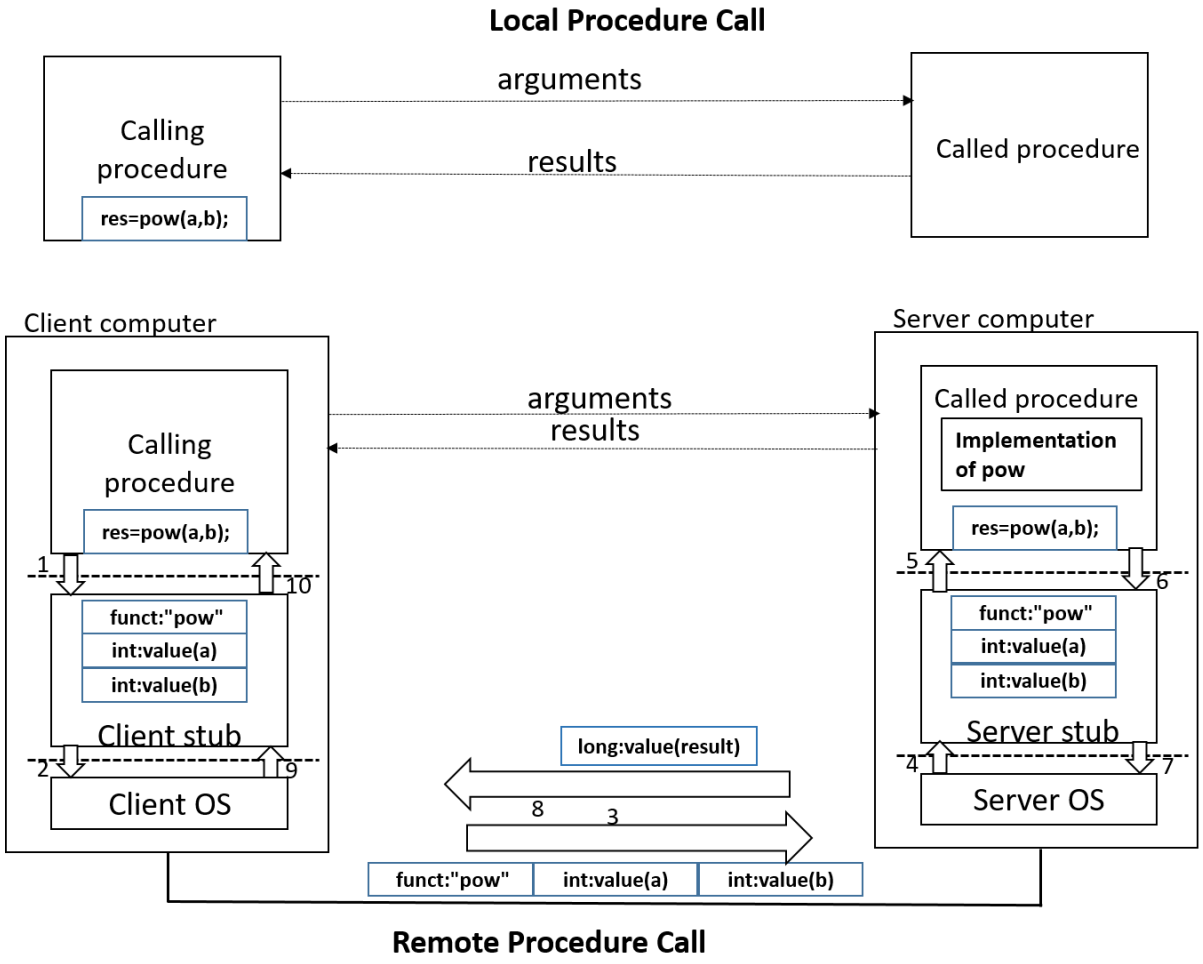

Figure 1: The RPC mechanism comparing with the Local Procedure Call.

Both the client program and the callee function see only ordinary, local procedure calls, using the normal calling conventions. Only the stubs know that the call is remote. It also means, the performance of RPC depends on the stub implementation apart from the network conditions.

Most languages were not designed to handle remote procedures natively with built in transparent stubs. That is the reason why they are not capable of generating the necessary stub functions. To enable them for performing remote procedure calls, the commonly adopted solution is to provide a separate compiler that can generate both the client and server stub functions. The input of this compiler comes from the remote procedure call interfaces written by a programmer. These are written in an interface definition language (IDL) for example proto3 in gRPC. After the RPC compiler is run, the server and client programs can be compiled and linked with the appropriate stub functions. Both the client and the server codes need to be changed to initialize the RPC mechanism.

2 RPC APIs

RPC implementations generally use supporting libraries to complete the stub operations. They must provide the following operations:

Name service operations: They must register themselves and support servers to advertise these bindings and clients to find them.

Binding operations: They establish client/server communications using the appropriate protocol.

Endpoint operations: They register endpoint information (protocol, port number, machine name) to the name server and listen for procedure call requests.

Security operations: They provide the authentication procedure and a secure communication channel between the two computers

Internationalization operations (possibly): They include functions to convert currency formats, time formats and language-specific strings through string tables.

Marshaling/data conversion operations: They pack data into package for transmitting onto a network and functions to reconstruct it. Sometimes, they have to serialize the messages as well.

Stub memory management and garbage collection: It may occur that stubs need to allocate memory for storing parameters, particularly in case of accomplishing pass-by-reference technique. RPC library needs to allocate and clean up such allocations. For RPC packages that support objects, the RPC system must provide the deletion of unnecessary references to objects.

Program ID operations: They allow applications to access identifiers of sets of RPC interfaces for communication.

Object and function ID operations: They support passing references to remote functions or remote objects to other processes. [5]

The more effective the implementation of these operations the faster the RPC solution will be.

3 Third generation RPCs and Web Services

Microsoft DCOM (Distributed Component Object Model) and CORBA (Common Object Request Broker Architecture) were the first RPC solutions that supported the object oriented programming techniques, and CORBA also includes IDL to specify the name of classes, their attributes, and their methods. It based on binary serialization. [5]

The increasing popularity of internet use led that web browsers became the dominant model for accessing information. Clients access the service via the HTTP protocol that allows services to be published, discovered, and used in a technology-neutral form.

Web server is configured to recognize the part of the URL pathname and pass the request to a specific plug-in module. This module can strip out the headers, parse the data (if needed), and call any other functions or modules as needed. [6][7]

XML-RPC

XML-RPC is one of the simplest web service approaches that was designed in 1998 as an RPC messaging protocol for serializing procedure requests and responses into human-readable XML. The XML format uses HTTP protocol to send data from a client computer to a server computer using traditional web ports for RPC.

XML-RPC does not define any standard methods for generating stub functions or handling remote procedures. It only focuses on messaging and therefore consists of only three small parts:

XML-RPC data model is a set of types used in passing parameters, return values, and faults (error messages).

XML-RPC request structures that contain method and parameter information for supporting HTTP requests.

XML-RPC response structures that contain return values or fault information for supporting HTTP responses.

For the performance test several libraries are available for example Apache XML-RPC that was selected to compare to other solutions.

3.1 SOAP and WSDL

The XML-RPC specification was used as a basis for creating SOAP (Simple Object Access Protocol) that is an open-standard, XML-based messaging protocol for exchanging information among computers. It is platform- and language-independent and enables client applications to easily connect to remote services and invoke remote methods. For creating a standardized messaging structure it is necessary to define a service definition document in WSDL (Web Services Description Language) so that to create and check the proper SOAP messages. Though, WSDL is an XML document, it is hard to create and read it by human, therefore tools such as *Java2WSDL* or *wsdl.exe* (in .NET) are used to generate template code for programmers. [5]

SOAP and WSDL are complex and highly-verbose formats, therefore their performances are naturally worse, than XML-RPC. Furthermore, if correctly implemented all XML-RPC libraries are compatible the same cannot be said about SOAP. The protocol has extensions which are not all implemented in all libraries. These properties make it somewhat unsuitable for our cross platform testing and was therefore omitted from the tests.

3.2 JSON-RPC

JSON (JavaScript Object Notation) is another marshaling format. JSON is based on JavaScript and does not need to be generated since it is human readable and writable, and it contains less redundancies. It was introduced as the "fat-free alternative to XML" as it has much less markup overhead compared to XML. This is just a messaging format and JSON do not offer RPC libraries and support for stub operations.

JSON-RPC is very similar to XML-RPC but encoded in JSON instead of XML. As XML-RPC was available before JSON-RPC this RPC has enjoyed less uptake. While JSON has less markup overhead the format is still textual and the savings are not large. This was also evident as for the example none of the available Ruby libraries had documentation. [5]

3.3 Google RPC and Google's Protocol Buffers

gRPC (Google RPC) is a cross-platform, language and platform independent, general-purpose infrastructure used by Google Inc. and they made it public in 2015. It can automatically generate idiomatic client and server stubs for service in a variety of languages and platforms. It uses Protocol Buffers that is a flexible, efficient, automated mechanism for binary serialization of structured data. [8]

Prompted by this newly released RPC solution, with this paper we aim to compare its use and performance to other popular solutions that predate it.

Users need to define how they want their data to be structured once in Protocol Buffers language (proto3) and the signature of the methods that will be called remotely. Then they can use a generated source code to easily write and read their structured data to and from a variety of data streams and using a variety of languages. Figure 2. shows the relevant sections of the proto file used for the performance test.

```
service Database {
  rpc Request(InfoRequest) returns (Info) {}
}
message Info {
  int32 id = 1;
  string first_name = 2;
  string last_name = 3;
  int32 age = 4;
  string email = 5;
  string phone = 6;
  bool newsletter = 7;
  float latitude = 8;
  float longitude = 9;
  bytes photo = 10;
}
message InfoList {
  repeated Info infos = 1;
}

message InfoRequest {
  int32 id = 1;
  bool photo = 2;
}
```

Figure 2: Services and messages defined in Protocol Buffers.

The defined data structure is stored in .proto files. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs. Once the user defined their messages, they run the protocol buffer compiler for their application's language on their .proto file to generate data access classes. These provide simple accessors for each field as well as methods to serialize/parse the whole structure to/from raw bytes – so, for instance, if the chosen language is C++, running the compiler on the user's .proto file will generate a class. User can then use this class in his application to populate, serialize, and retrieve the class protocol buffer messages. The compiler also provides the stub implementations that can be inherited to code the remote function definition.

The protocol buffer message encoded in binary format is much smaller than its XML code but is not human-readable and human-editable. Protocol buffers result not only binary format but are 3 to 10 times smaller and 20 to 100 times faster than XML for serializing structured data that may one of the reasons for the higher performance of gRPC. [9]

4 The performance test of the implemented RPCs

Based on the structure of RPC the performance differences of the different RPC solutions must come from the differently implemented stub operations. The RPC solution that performs stub operations the fastest way and produces the shortest data for sending must have the best performance.

We have performed benchmarks to test the performance of each of these RPC methods and compare them against each other. (See the signature of the methods in Figure 2.)

With these benchmarks the aim was to measure the processing overhead of the RPC methods and their implementations.

For the request method we have written server and client implementations in C++, Java, and Ruby. The server part reads sample data that has multiple data formats, including strings, integers, floats, and 1MB of binary data. After the data has been read it starts listening for connections from the client. The client can only send one request to the server, which is requesting one of the data items with an option to specify whether to include the binary data part or not. The request method in the client program was invoked 100 times, the client program was run 10 times.

The data on the server component was serialized from memory where it was loaded previously, which was not part of the measurement. The client component did no processing on the data apart from printing receipt of request with the identifier from the current item to

standard output. This was to prevent potential elision of deserialization.

The RPC methods would usually be part of a system that further processes data in either a synchronous or asynchronous manner that would have different performance and latency implications. Asynchronous or non blocking systems are usually preferred for more optimal resource usage on both client and server side. With non blocking operations the components would send further requests needed to fulfill their answer, but they would not wait for the answer actively while holding up resources. Instead these systems store that a request is pending, suspend execution of the routine, and continue to do other outstanding operations that they have the necessary data for. When the answer arrives from the server, they load the previously stored request and execution state and continue from the point where execution was suspended.

Our implementation of the server and client do not follow this asynchronous model of operation, but instead blocks until the response arrives from the server. The reason for this is to have more reliable and stable measurements. As we focus on the RPC itself, the server and client components do minimal processing, there are no further requests to wait for. Using an asynchronous model would mean more outside effects on the measurements, as asynchronous signaling is less predictable than synchronous blocking operations.

For gRPC the gRPC and Protobuffers library were used, for XML RPC and JSON RPC the most popular library was selected for each language. These are: for XML-RPC in C++ xmlrpc-c[10], in Java Apache XMLRPC [11], in Ruby the standard library XMLRPC [12] for JSON-RPC in C++ jsonrpccpp[13], in Java JSON-RPC 2.0 by [d]zhuvinov [s]oftware [14], in Ruby jsonrpc2.0 with webrick [15]. The only restriction was that it needed to be able to start listening for connections without a large framework that it would be deployed part of. This means that for example Servlet based Java implementations were excluded.

Docker containers were created for each of these server and client implementations so that they had a runtime environment that is not dependant on the host system. This caused some overhead when starting the client programs, as a new Docker instance had to created for each run, but we found that this did not influence our overall conclusion.

We used a Linux rack to run the server instances and a commodity laptop to run the client instances to simulate somewhat real conditions and connected both of them to the subnet with 125 MBit/s wired connections to exclude the interference in WiFi or otherwise long distance internet connection.

With the RPC method we cross tested all of the languages with each other to get more measurements and lessen the influence of particular implementations on the overall results [16].

It also has to be noted while XML-RPC implementations were easy to find, JSON-RPC is not as widespread judging from the available libraries. The only server library available for Ruby had some issues and no documentation. Table 1 and Table 2 show the results.

		server		
	grpc small	**cxx**	**java**	**ruby**
client	**cxx**	1.446586288	1.538543454	1.843524988
	java	2.385020082	2.574738704	2.862809575
	ruby	2.335048487	2.357542191	2.329401348
	xmlrpc small	**cxx**	**java**	**ruby**
client	**cxx**	1.745901795	1.789834515	6.283093411
	java	1.872503282	1.932996376	6.336315439
	ruby	2.531053102	2.39889046	6.991524778
	jsonrpc small	**cxx**	**java**	**ruby**
client	**cxx**	1.746023073	5.997436199	6.166025146
	java	1.857895238	6.242739725	6.315714135
	ruby	2.245318859	6.360743144	6.627618996

Table 1: The measured average values in seconds after 100 invokes and 5 runs with small test data.

The overall results we have found in our test runs it that overall gRPC performed the best of all three, with XML RPC and JSON RPC having similar performance characteristics with the differences between mainly attributable to implementation details of the libraries. (Table 1)

With small test data, without the 1MB binary, we found that while the different methods had similar performance, in most cases the gRPC was slightly faster except, for example, in the java server java client case where the gRPC implementation did 2.5s while the XML RPC finished in under 2s. In XML-RPC, the Ruby server implementation almost tripled the amount of time required to run the tests regardless of client language. The same can be observed in JSON RPC with the Java and Ruby server implementation. With small test data, C++ implementations were faster than the Java or Ruby ones.

The languages, in which the stub operations are implemented also influences the performance. All RPC solutions performed better in C++ with small test data.

		server		
	grpc big	**cxx**	**java**	**ruby**
client	**cxx**	10.49837347	19.78831593	10.61724809
	java	11.50234759	11.65334163	20.65961758
	ruby	11.44378246	15.36368512	16.35586611
	xmlrpc big	**cxx**	**java**	**ruby**
client	**cxx**	16.87760948	23.02763573	17.56123346
	java	16.70705216	22.97051365	16.33389231
	ruby	31.4515749	36.40461197	26.93085479
	jsonrpc big	**cxx**	**java**	**ruby**
client	**cxx**	23.85746603	23.59290045	23.09974722
	java	18.68594349	21.19607064	18.29131204
	ruby	17.81739152	17.89413377	17.48205703

Table 2: The measured average values in seconds after 100 invokes and 5 runs with binary data.

With the inclusion of the binary data the differences were more pronounced (see Table 2). gRPC performed better except one case. How much faster it was depended on the language combination used. Only the Ruby server with the Java client did beat the time of the gRPC solution. The XML-RPC Ruby client was generally slower than other clients, taking almost twice the time to complete the test runs.

The increased performance of gRPC can be attributed to the transmission format. Both XML and JSON are textual formats. While binary versions exist, these are not as widely used and the RPC libraries do not use them. Because of their text nature to include binary data in them it needs to encoded to some representation that only uses printable ASCII characters, in most cases to Base64. This increases the data to be transmitted by 4/3 and the overhead of the markup structure is also not insignificant. gRPC uses Protobuffers as its wire format, which is a binary format. Binary data can be included as is, no conversion necessary. It also does not add much overhead to the structure, only field identifiers are added for backward compatibility.

5 Conclusions

In this paper, the structure of third generation RPCs was analysed to find answers for the differences in the performance of different RPC solutions: Google RPC, XML-RPC and JSON-RPC. The chosen libraries implemented the stub operations in different ways and used different formats for marshalling. gRPC with Protocol Buffers performed best in our tests because of the fast binary serialization method of structured data, that resulted in smaller sized encoded messages. Our tests proved, that the chosen computer language has an influence on the performance of RPC invocations. gRPC proved faster in C++ implementations than in Java or Ruby with small test data. In case of XML-RPC and JSON-RPC, Ruby server with Java client proved to be the fastest with large test data.

References

[1] Andrew D. Birrell and Bruce Jay Nelson (1984). Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984, Pages 39-59. https://doi.org/10.1145/2080.357392

[2] Andrew S. Tanenbaum, Robbert van Renesse (1988). *A Critique of the Remote Procedure Call Paradigm.* Available at http://www.cs.vu.nl/~ast/Publications/Papers/euteco-1988.pdf

[3] Andrew S. Tanenbaum, Maarten van Steen (2016). *Distributed Systems: Principles and Paradigms*. Pearson Education Inc. ISBN:978-15-302817-5-6
Andrew D. Birrell (1985). Secure Communication Using Remote Procedure Calls. *ACM Transactions on Computer Systems*, Vol. 3, No. 1, February 1985, Pages 1-14. https://doi.org/10.1145/214451.214452
[4] Paul Krzyzanowski (2012). *Remote Procedure Calls* Available at https://www.cs.rutgers.edu/~pxk/417/notes/08-rpc.html
[5] Michael D. Schroeder and Michael Burrows1 (2006). *Performance of Firefly RPC*. http://web.mit.edu/6.826/www/notes/HO11.pdf
[6] Hakan Bagci and Ahmet Kara (2016). A Lightweight and High Performance Remote Procedure Call Framework for Cross Platform Communication. *ICSOFT-EA 2016 Abstracts*. Available at: http://www.scitepress.org/DigitalLibrary/PublicationsDetail.aspx?ID=Rqt07DUDIy8=&t=. https://doi.org/10.5220/0005931201170124
[7] *What is gRPC?* Available at http://www.grpc.io/docs/guides/
[8] *Protocol Buffers.* Available at https://developers.google.com/protocol-buffers/docs/overview#whynotxml
[9] *XML-RPC for C and C++*. Available at http://xmlrpc-c.sourceforge.net/
[10] Apache XML-RPC. Avilable at https://ws.apache.org/xmlrpc/
[11] *XML-RPC for Ruby*. Available at https://github.com/ruby/xmlrpc
[12] *JSON-RPC 2.0. Essential Java libraries and tools for JSON-RPC 2.0development.* Available at http://software.dzhuvinov.com/json-rpc-2.0.html
[13] *JSON-RPC 2.0*. Available at http://software.dzhuvinov.com/json-rpc-2.0-client.html
[14] *JSON-RPC 2.0. for Ruby*. Available at https://github.com/chriskite/jimson
[15] *Downloadable programs and the environment for the benchmarks*: Available at https://github.com/ksanyi007/rpc