A Lightweight and High Performance Remote Procedure Call Framework for Cross Platform Communication

Hakan Bagci and Ahmet Kara
Tubitak Bilgem Iltaren, Ankara, Turkey

Keywords: Remote Procedure Call, High Performance Computing, Cross Platform Communication.

Abstract: Remote procedure calls (RPC) are widely used for building distributed systems for about 40 years. There are several RPC implementations addressing different purposes. Some RPC mechanisms are general purpose systems and provide a number of calling patterns for functionality, hence they do not emphasize performance. On the other hand, a small number of RPC mechanisms are implemented with performance as the main concern. Such RPC implementations concentrate on reducing the size of the transferred RPC messages. In this paper, we propose a novel lightweight and high performance RPC mechanism (HPRPC) that uses our own high performance data serializer. We compare our RPC system's performance with a well-known RPC implementation, gRPC, that addresses both providing various calling patterns and reducing the size of the RPC messages. The experiment results clearly indicate that HPRPC performs better than gRPC in terms of communication overhead. Therefore, we propose our RPC mechanism as a suitable candidate for high performance and real time systems.

1 INTRODUCTION

RPC enables users to make a call to a remote procedure that resides in the address space of another process (Corbin, 2012). This process could be running on the same machine or another machine on the network. RPC mechanisms are widely used in building distributed systems because they reduce the complexity of the system and the development cost (Kim et al., 2007). The main goal of an RPC is to make remote procedure calls transparent to users. In other words, it allows users to make remote procedure calls just like local procedure calls.

The idea of the RPC is first discussed in 1976 (Nelson, 1981). Nelson studied the design possibilities for RPCs in his study. After Nelson, several RPC implementations developed such as Courier RPC (Xerox, 1981), Cedar RPC (Birrell and Nelson, 1984) and SunRPC (Sun Microsystems, 1988; Srinivasan, 1995). SunRPC became very popular on Unix systems. Then it is ported for Solaris, Linux and Windows. SunRPC is widely used in several applications, such as file sharing and distributed systems (Ming et al., 2011).

RPC mechanism is basically a client-server model (Ming et al., 2011). Client initiates an RPC request that contains the arguments of the remote procedure through a proxy. The request is then transferred to a known remote server. After the server receives the request, it delegates the request to a related service that contains the actual procedure. The service then executes the procedure using received arguments and produces a result. Server sends this result back to client. This process is illustrated in Figure 1.

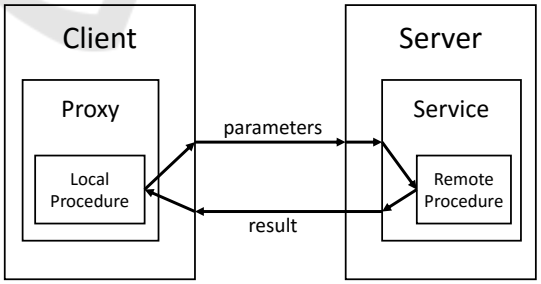


Figure 1: RPC Client Server Communication.

One of the major concerns in high performance computing systems and real-time systems is to reduce the size of data transfer. Therefore, a system that wants to make a remote procedure call needs a lightweight RPC mechanism with minimal overhead. Furthermore, most of the RPC mechanisms operate through networks, hence it is important to reduce the size of the RPC packets. There are several research

Bagci, H. and Kara, A.
A Lightweight and High Performance Remote Procedure Call Framework for Cross Platform Communication.
DOI: 10.5220/0005931201170124
In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA*, pages 117-124
ISBN: 978-989-758-194-6
Copyright © 2016 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

work focused on reducing the overhead of RPC mechanisms (Bershad et al., 1990; Satyanarayanan et al., 1991).

Other RPC mechanisms focus on providing different types of calling patterns such as asynchronous and parallel remote procedure calls (Satyanarayanan and Siegel, 1990; Ananda et al., 1991). However, increasing the number of calling patterns brings extra overhead to the system. The size of the RPC packets increases as new high level features are added to the RPC mechanism. This is because of the increasing size of protocol control data.

gRPC is a language-neutral, platform neutral, open source RPC mechanism initially developed at Google (Google, 2015b). It employs protocol buffers (Protobuf) (Google, 2015a), which is Google's mature open source mechanism, for structured data serialization. gRPC aims to minimize data transfer overhead while providing several calling patterns. gRPC is a general purpose RPC mechanism, thus has several high level features, which bring extra overhead in data transfer. For example, an identifier is transferred to the server for each parameter. Therefore, it is not suitable for systems that aims minimal data transfer overhead.

In this paper, we introduce a lightweight and high performance remote procedure call mechanism, namely **H**igh **P**erformance **R**emote **P**rocedure **C**all (HPRPC), that aims to minimize data transfer overhead. We employ our own structured data serialization mechanism, named Kodosis. Kodosis is a code generator that enables high performance data serialization. The main contributions of our work are given below:

- A novel lightweight and high performance RPC mechanism is proposed.
- A code generator that enables high performance data serialization is introduced.
- A comparison between the performances of our RPC mechanism and gRPC is given which demonstrates that our RPC mechanism performs better than gRPC in terms of data transfer overhead.

The rest of the paper is organized as follows: In the next section, the research related to the RPC mechanisms is briefly explained. In Section 3, we provide the details of HPRPC. The experiment results are given in Section 4. Finally, we conclude the paper with some discussions.

2 RELATED WORK

RPC mechanisms are widely used for building distributed systems since mid 1970s. There are several RPC implementations in the literature. In this section, we briefly describe the RPC mechanisms that are related to our work.

XML-RPC (UserLand Software Inc., 2015) enables remote procedure calls using HTTP as transport protocol and XML as serialization technique for exchanging and processing data (Jagannadham et al., 2007). Since HTTP is available on every kind of programming environment and operating system, with a commonly used data exchange language as XML, it is easy to adapt XML-RPC for any given platform. XML-RPC is a good candidate for integration of multiple computing environments. However, it is not good for sharing complex data structures directly (Laurent et al., 2001), because HTTP and XML bring extra communication overhead. In (Allman, 2003), the authors show that the network performance of XML-RPC is lower than *java.net* due to the increased size of XML-RPC packets and encoding/decoding XML overhead.

Simple Object Access Protocol (SOAP) (W3C, 2015) is a widely used protocol for message exchange on the Web. Similar to XML-RPC, SOAP generally employs HTTP as transport protocol and XML as data exchange language. While XML-RPC is simpler and easy to understand, SOAP is more structured and stable. SOAP has higher communication overhead than XML-RPC because it adds additional information to the messages that are to be transferred.

In (Kim et al., 2007), the authors present a flexible user-level RPC system, named FlexRPC, for developing high performance cluster file systems. FlexRPC system dynamically changes the number of worker threads according to the request rate. It provides client-side thread-safety and supports multithreaded RPC server. Moreover, it supports various calling patterns and two major transport protocols, TCP and UDP. However, these extra capabilities of FlexRPC could cause additional processing and communication overhead.

In recent years, RPC mechanisms has started to be used in context of high-performance computing (HPC). Mercury (Soumagne et al., 2013) is an RPC system specifically designed for HPC systems. It provides an asynchronous RPC interface and allows transferring large amounts of data. It does not implement higher-level features such as multithreaded execution, request aggregation and pipelining operations. The network implementation of Mercury is abstracted which allows easy porting to future systems as well as

efficient use of existing transport mechanisms.

gRPC (Google, 2015b) is a recent and widely used RPC mechanism that addresses both issues; providing various calling patterns and reducing the size of the RPC messages. It uses Protobuf (Google, 2015a) for data serialization. Protobuf aims to minimize the amount of data to be transferred. Since gRPC is a recent and well-known RPC framework and Protobuf is a mature serialization mechanism, we compare our HPRPC system's performance with gRPC. The comparison results are presented in Section 4.

3 PROPOSED WORK

High Performance Remote Procedure Call (HPRPC) Framework is designed with three main objectives:

- Lightweight & simple architecture
- High performance operation
- Cross-Platform support

3.1 Architecture

The design of HPRPC is inspired from (Google, 2015c), but it is implemented completely in our research group with our objectives and experiences in mind. It relies on three main components and one controller (Figure 2). *RpcChannel* is the communication layer that is responsible for transportation of data. *RpcClient* is responsible for sending client requests on the channel and *RpcServer* is responsible for handling client requests incoming from the channel. *RpcController*, on the other hand, is the main controller class that directs messages of client/server to/from the channel. We will dig into the capabilities of these classes in the following sections.

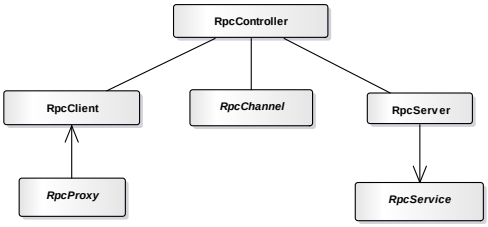


Figure 2: HPRPC Framework Overview.

3.1.1 Channel

RpcChannel implements duplex transportation mechanism to send/receive messages (Figure 3). Client/Server initiates a request by calling the *StartCall* method, which then requests an *RpcHeader* that identifies the message and returns a *BinaryWriter* object to transport the parameters. Caller serializes the method parameters/return value to the *BinaryWriter* object and calls the *EndCall* method to finish the remote procedure call. On the other side the receiving *RpcChannel* deserializes the *RpcHeader* structure and sends it to the Controller with a *BinaryReader* object in order to read the parameters/return value. Controller invokes Client or Server message handling methods according to this *RpcHeader* information.

Currently, *RpcChannel* is implemented only for the Tcp/Ip communication layer. However, the architecture allows any other implementations that include pipes or shared memory mechanisms.

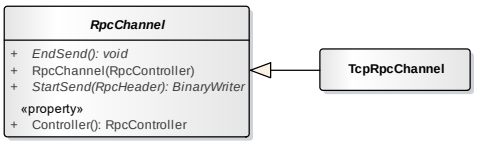


Figure 3: RpcChannel Class.

RpcHeader is a simple struct that consists of 4 fields (Figure 4):

- A unique id that identifies the call
- Type of message that identifies the payload
- Type of service
- Type of called procedure

Header struct specifies its serialization/deserialization methods that use a 4-byte preamble to protect message consistency. If the previous message was not consumed properly, preamble-check throws an exception to prevent unexpected failures.

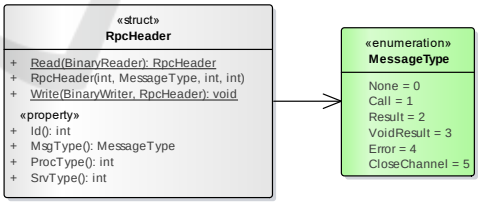


Figure 4: RpcHeader Struct.

3.1.2 Client

RpcClient is the class that is used by the proxies to send method call requests and wait for their responses (Figure 5). It contains different *XXXCall* methods to initiate remote procedure calls and handle their responses. Call requests return a *PendingCall* object that specifies a ticket for the return value. Caller waits on the *PendingCall* object. When a response is received from the server, it returns a value or throws an exception depending on the remote procedure's response.

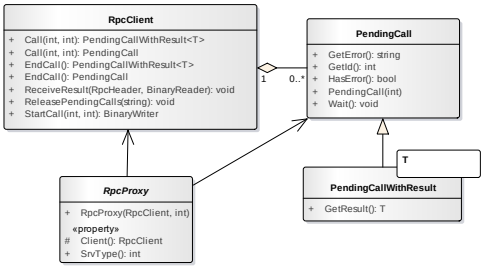


Figure 5: RpcClient Class.

Proxies are custom generated classes that transform user requests to the *RpcClient* calls. For each method call, the proxy class calls the *XXXCall* method of the *RpcClient* and waits for the *PendingCall* object. This mechanism introduces transparency for the user on the HPRPC framework. In Figure 6, it is shown that the *IConnectionTester* interface is implemented on the *ConnectionTesterProxy*. User calls the *KeepAlive* method of this interface and does not deal with the HPRPC framework details.

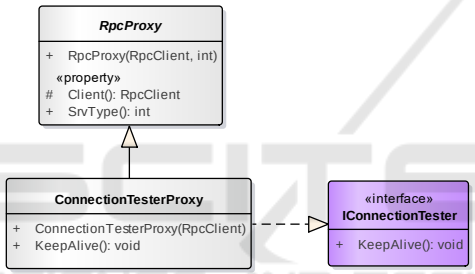


Figure 6: RpcProxy Class.

3.1.3 Server

RpcServer is responsible for handling client requests (Figure 7). According to the *ServiceType* attribute it calls the related *RpcService*'s *Call* method. Then serializes its *ReturnValue* or in case of an error the exception information for the client.

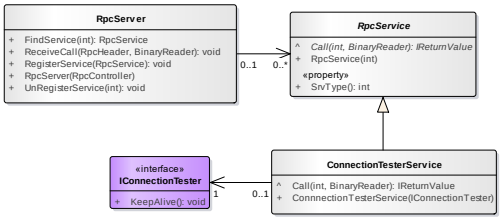


Figure 7: RpcServer Class.

Services are custom generated classes that override the *Call* method of *RpcService* class. They invoke the relevant method of their inner service instance with the appropriate parameters. Instead of implementing service methods like proxies, service classes invoke service calls on an object that they own. As in Figure 7, when the client calls the *KeepAlive* method, *RpcServer* directs this call to *ConnectionTesterService* which subsequently calls the *KeepAlive* method of its own object.

The *Call* method of an *RpcService* class returns an *IReturnValue* object that holds the return value of the procedure call. As in Figure 8, return values can be void or a typed instance of *ReturnValue* class. They are handled on the *RpcServer* to transfer the method results.

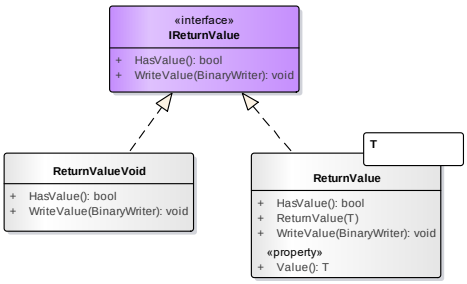


Figure 8: RpcReturnValue Class.

3.2 High Performance

One of the main objectives of HPRPC is having *High Performance* operations. To achive this objective we focused on two major areas:

- Having lightweight design
- Optimization of data serialization

3.2.1 Lightweight Design

The main objective of RPC is to establish a connection between two endpoints easily with a procedure call. Any other capabilities like asynchronous call, multi-client/multi-server, heterogeneous platforms etc. are additional features to facilitate special user needs, but may reduce performance because of their extra control mechanisms and data exchange costs. For the *High Performance* objective in HPRPC, we skipped these features to increase overall performance. Basic features of our *Lightweight Design* can be outlined as follows:

- Synchronous Call: Our clients can call one procedure at a time.
- Single Client/Single Server: Our client and server classes allow single endpoints.
- Homogenous Platforms: Although we support cross-platform communications, the endpoints must be homogenous. Therefore, their byte order (Little/Big Endian), string encoding etc. should be identical.

We implemented our architecture ready for an asynchronous call with the *PendingCall* class usage. In the future, we are planning to add asynchronous calls optionally with a limited performance loss.

3.2.2 Data Serialization

A procedure call generally includes some additional data to transfer like method parameters and return value. Hence, performance of an RPC mechanism is highly correlated with the performance of its data serialization/deserialization mechanism. In HPRPC, we implemented data serialization with the help of a code generator named *KodoSis*, which is developed in our research group. It has the following capabilities:

- Consistency: Our data structures are defined in a single configuration file and generated for all endpoints and platforms by *KodoSis*, which allows us to use consistent data structures.
- Integrity: *KodoSis* generates the serialization/deserialization routines for all data structures without any custom code.
- Minimal Overhead: No informative tags are required in data serialization, because both ends are aware of the complete structure of the data.
- Performance: *KodoSis* serialization routines use special mechanisms, specifically for lists of blittable types to increase performance.

3.3 Cross-platform Support

Our architecture has been implemented on several languages and platforms. The same architecture, with identical class and method names, is implemented and tested for multiple languages and compilers. All implementations can communicate with each other, without the knowledge of the other platform.

In Windows operating system, we support C# language with .NET Framework and C++ language with Microsoft Visual C++ Compiler. In Linux like operating systems we support C++ language with *gcc* compiler.

In C++, we employed C++ v11 constructs like *shared_ptr*, *thread*, *mutex* etc. to increase robustness and have simple code. In some legacy systems (gcc v4.3) that does not support C++ v11, equivalent classes of The Boost Framework (Boost, 2015) are employed.

4 EVALUATION

As explained previously, HPRPC aims to minimize communication overhead between the RPC client and the RPC server. In order to achieve this goal, it employs Kodosis to minimize the amount of data to be transferred through the RPC channel. We conduct several experiments to evaluate our proposed work. In these experiments, we compare the performance of HPRPC with gRPC. gRPC is a well-known open source RPC mechanism that is initially developed by Google. It employs Protobuf for data serialization. Similar to Kodosis, Protobuf also aims to minimize the amount of data transferred.

4.1 Evaluation Methodology

Four different operations are chosen to be used in experiments. Each operation uses different kinds of requests and responses. *Average* and *GetRandNums* operations are also used in evaluation of XML-RPC (Allman, 2003). The details of these operations are given below:

- ***HelloReply SayHello (HelloRequest Request)***
  This operation sends a *HelloRequest* message that contains a parameter with type string and returns a *HelloReply* message that contains "Hello" concatenated version of the request parameter. This operation represents a *small request/small response* transaction.
- ***DoubleType Average (Numbers Numbers)***
  This operation returns the average of *Numbers* list that contains 32-bit integers. The size of the *Numbers* list is not fixed. We test the performance of RPC mechanisms under varying numbers of *Numbers* list. This operation represents a *big request/small response* transaction.
- ***Numbers GetRandNums (Int32Type Count)***
  This operation returns a list of random 32-bit integers given the desired number of random numbers. We test the performance of RPC mechanisms under the varying numbers of *count* parameter. This operation represents a *small request/big response* transaction.
- ***LargeData SendRcvLargeData (LargeData Data)***
  This operation receives a large amount of data and returns the same data. There are different versions of *LargeData* that contain different number of parameters. This operation represents a *big request/big response* transaction.

In experiments, we measure the transaction times of each operation for gRPC and HPRPC. All the experiments are repeated for 1000 times and the total transaction times are recorded. In large data experiments, we also measure the RPC packet sizes to compare the

amount of data that is to be transferred through RPC channels.

4.2 Experiment Results

For the first experiment, we measure the transaction time of *SayHello* function which is a *small request/small response* operation. The total transaction times of 1000 operations for gRPC and HPRPC are 305 ms and 279 ms, respectively. HPRPC performs 9.3% better than gRPC. This is due to the fact that Protobuf tags the string parameter in *HelloRequest* message while Kodosis does not use tags in serialization. Therefore, total amount of data that is to be transferred is increased in gRPC as opposed to HPRPC. In this experiment, we only have a single parameter in the request and the response messages. We also examine the performance of the two RPC mechanisms when we increase the number of parameters in a single message. The results of this experiment is presented in large data tests.

The second experiment measures the performance of the two RPC mechanisms using *Average* function which represents a big request/small response transaction. The results of *Average* operation tests are depicted in Figure 9a. In this experiment, clients call the respective *Average* functions from RPC servers for different number of 32-bit integers. When the number of inputs is 10, HPRPC has 8.5% better performance than gRPC. The difference between performances increase and gets the highest value, when the input size increased to 10000. At that point, gRPC's total transaction time is 21.6% higher than HPRPC. The performance difference gets even higher when the input size is beyond 10000. This result gives us a hint about the improvement of HPRPC when the amount of data is increased.

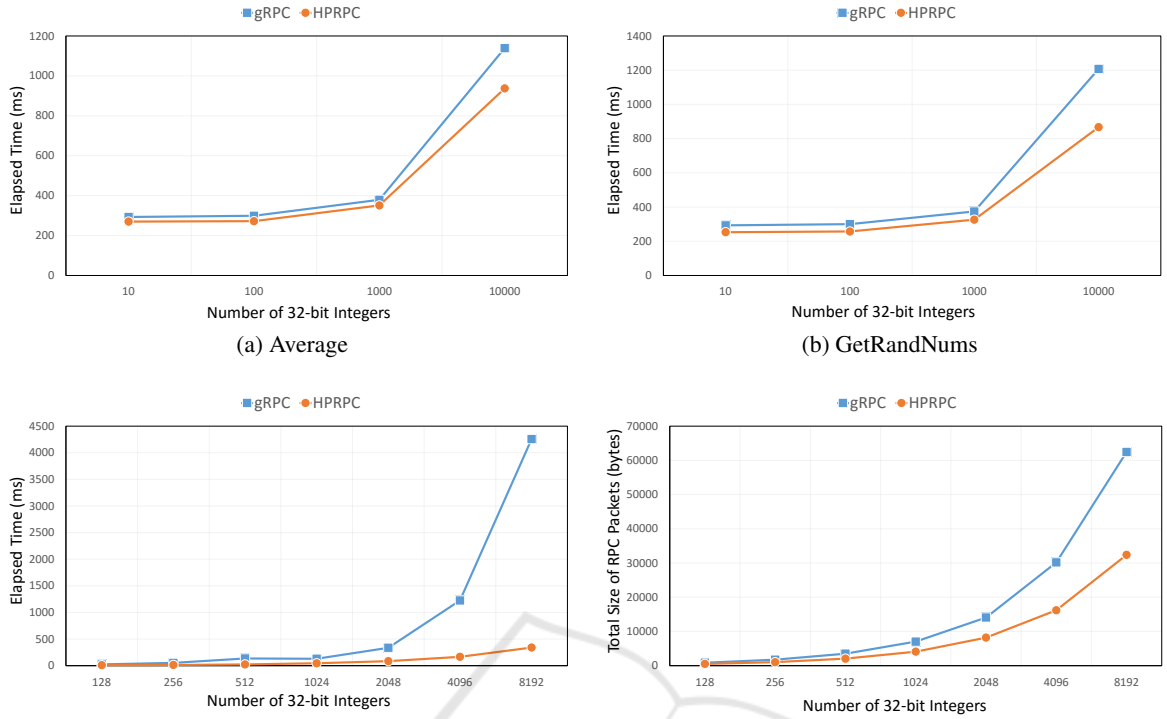
In *Average* function experiments, we only increase the size of the request message. For completeness, we also measure performance for increased response message size. The results of *GetRandNums* function experiments are shown in Figure 9b. As expected, the results of this experiment are similar to the previous experiment' results. In this case, the performance difference increases even more. When 10 random numbers are requested from RPC server, HPRPC performs 12.9% better than gRPC. The difference increases up to 39.2% when requested random number count is 10000.

In *Average* and *GetRandNums* experiments, *Numbers* message type is employed as request and response type, respectively. *Numbers* data structure models multiple numbers as a repeated member of the same type. gRPC and HPRPC both have optimizations on repeated members. In the next experiment, we evaluate the performances of RPC mechanisms when a message type consists of non-repeating members. In this experiment, we employ different versions of *LargeData* message type such as LargeData128, LargeData256, etc. The number concatenated to LargeData indicates the number of members in message type. For example, LargeData128 message type contains 128 different members of type integer.

SendRcvLargeData experiments employ 7 different versions of *LargeData* message types to evaluate the performance of RPC mechanisms under the varying numbers of members in the message. In these tests, clients call the respective SendRcvLargeData function from RPC servers, and servers send the same data back to the clients. The results of this experiment are illustrated in Figure 9c. HPRPC performance is nearly 2.7 times better than gRPC when member count is 128. As the number of members in message types is increased, the transaction times of gRPC increase substantially as opposed to HPRPC. In the extreme case of 8192 number of members, gRPC consumes approximately 12.5 times more transcation time than HPRPC. This result clearly indicates that HPRPC outperforms gRPC with increasing number of members in RPC messages.

In order to find out why gRPC needs so much transaction time when the number of members are increased in messages, we further analyze the results of the *SendRcvLargeData* experiments. In this analysis, we compare the transferred data packet sizes of gRPC and HPRPC. The results of this analysis are depicted in Figure 9d. This analysis reveals the fact that gRPC tends to use more bytes than HPRPC when the number of members is increased in messages. This is because of that gRPC employs a unique tag for each member in messages. The messages are serialized with these tags, hence the RPC packet sizes are increased. Moreover, as the number of members are increased, the number of bytes that are used to represent tag values also increase. Tags are useful in some cases such as enabling optional parameters, however tagging each parameter brings extra communication and processing overhead. Therefore, we conclude that tagging parameters is not useful for high performance systems.

HPRPC assumes all parameters in a message as required, and the order of parameters do not change. Therefore, RPC messages are serialized and deserialized in the same order, which eliminates the need for tagging parameters in HPRPC. This assumption gives HPRPC advantage over other RPC implementations in high performance systems. This is due to



(a) Average (b) GetRandNums

(c) SendRcvLargeData (d) Large Data Packet Size

Figure 9: Experiment Results.

the reduced RPC packet size and eliminated extra processing cost of tagging. The results of *SayHello* and *GetRandNums* experiments also verifies with this argument.

Protobuf additionally, enables several features for definition of message types such as optional parameters which brings extra overhead in size of RPC packets. On the other hand, our Kodosis implementation provides a simple message definition language that only enables basic required features. These features of Kodosis help reduce RPC packet sizes, hence the communication overhead.

5 CONCLUSION

HPRPC is an RPC mechanism that focuses on three main objectives: Lightweight & simple architecture, high performance and cross-platform support. In this paper, we give details of our architecture with relevant class diagrams and present some performance evaluations that indicate our advantages over gRPC.

In order to evaluate the performance of our RPC mechanism, we compare HPRPC with gRPC which is a recent and well-known RPC mechanism that addresses both providing various calling patterns and reducing the size of the RPC messages. The results of the experiments clearly indicate that HPRPC outperforms gRPC in terms of communication overhead in all of test cases. This is due to the high level features of gRPC that bring extra overhead in data transfer. In addition, the underlying serialization mechanism of gRPC, Protobuf, tags all parameters in a message. Therefore, gRPC messages tend to occupy more bytes in data transfer. The size of the messages get even higher as the number of parameters in a message are increased. On the other hand, HPRPC provides not only all the required basic features but also ensures reduced RPC message size and minimal processing overhead. This advantage allows HPRPC to perform better in high performance scenarios. We conclude that HPRPC is an appropriate RPC mechanism to be used in high performance systems.

Currently, we implemented HPRPC with C# and C++ languages on Windows and Linux operating systems. In the future we are planning to implement our architecture with Java and increase our cross-platform support. Moreover, we plan to implement asynchronous calling patterns and concurrent execution of requests on server side.

REFERENCES

Allman, M. (2003). An evaluation of xml-rpc. *ACM sigmetrics performance evaluation review*, 30(4):2–11.

Ananda, A. L., Tay, B., and Koh, E. (1991). Astra-an asynchronous remote procedure call facility. In *Distributed Computing Systems, 1991., 11th International Conference on*, pages 172–179. IEEE.

Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. (1990). Lightweight remote procedure call. *ACM Transactions on Computer Systems (TOCS)*, 8(1):37–55.

Birrell, A. D. and Nelson, B. J. (1984). Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59.

Boost (2015). Boost Framework. http://www.boost.org. Accessed: 2015-12-08.

Corbin, J. R. (2012). *The art of distributed applications: programming techniques for remote procedure calls*. Springer Science & Business Media.

Google (2015a). Google Protocol Buffers (Protobuf): Google's Data Interchange Format. Documentation and open source release. https://developers.google.com/protocol-buffers. Accessed: 2015-12-08.

Google (2015b). gRPC. http://www.grpc.io. Accessed: 2015-12-08.

Google (2015c). Protobuf-Remote: RPC Implementation for C++ and C# using Protocol Buffers. https://code.google.com/p/protobuf-remote. Accessed: 2015-12-28.

Jagannadham, D., Ramachandran, V., and Kumar, H. H. (2007). Java2 distributed application development (socket, rmi, servlet, corba) approaches, xml-rpc and web services functional analysis and performance comparison. In *Communications and Information Technologies, 2007. ISCIT'07. International Symposium on*, pages 1337–1342. IEEE.

Kim, S.-H., Lee, Y., and Kim, J.-S. (2007). Flexrpc: a flexible remote procedure call facility for modern cluster file systems. In *Cluster Computing, 2007 IEEE International Conference on*, pages 275–284. IEEE.

Laurent, S. S., Johnston, J., Dumbill, E., and Winer, D. (2001). *Programming web services with XML-RPC*. O'Reilly Media, Inc.

Ming, L., Feng, D., Wang, F., Chen, Q., Li, Y., Wan, Y., and Zhou, J. (2011). A performance enhanced user-space remote procedure call on infiniband. In *Photonics and Optoelectronics Meetings 2011*, pages 83310K–83310K. International Society for Optics and Photonics.

Nelson, B. J. (1981). "Remote Procedure Call" CSL-81-9. Technical report, Xerox Palo Alto Research Center.

Satyanarayanan, M., Draves, R., Kistler, J., Klemets, A., Lu, Q., Mummert, L., Nichols, D., Raper, L., Rajendran, G., Rosenberg, J., et al. (1991). Rpc2 user guide and reference manual.

Satyanarayanan, M. and Siegel, E. H. (1990). Parallel communication in a large distributed environment. *Computers, IEEE Transactions on*, 39(3):328–348.

Soumagne, J., Kimpe, D., Zounmevo, J., Chaarawi, M., Koziol, Q., Afsahi, A., and Ross, R. (2013). Mercury: Enabling remote procedure call for high-performance computing. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE.

Srinivasan, R. (1995). RPC: Remote Procedure Call Protocol Specification Version 2 (RFC1831). *The Internet Engineering Task Force*.

Sun Microsystems, I. (1988). RPC: Remote Procedure Call Protocol Specification Version (RFC1057). *The Internet Engineering Task Force*.

UserLand Software Inc. (2015). XML-RPC Specification. http://xmlrpc.scripting.com/spec.html. Accessed: 2015-12-15.

W3C (2015). SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). http://www.w3.org/TR/soap12/. Accessed: 2015-12-15.

Xerox, C. (1981). Courier: The Remote Procedure Call Protocol. *Xerox System Integration Standard 038112*.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS