http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

OPEN ACCESS
Information Technology Journal
ISSN 1812-5638
DOI: 10.3923/itj.2022.1.7

Review Article

Asynchronous RPC Interface in Distributed Computing System

Gajendra Sharma

Department of Computer Science and Engineering, Kathmandu University Dhulikhel, Nepal

Abstract

Remote Procedure Call (RPC) is defined as a model for inter-process communication in distributed systems. RPC is simple as well as powerful. However, most of the RPC systems used synchronously in nature and synchronous RPC systems are not able to exploit fully parallelism in distributed applications. Therefore, various asynchronous RPC systems have been designed to achieve higher parallelism while retaining the simplicity of synchronous RPC. Asynchronous RPC calls do not block the client and the replies can be received when they are needed. Asynchronous RPC calls can be categorized into two types depending on whether the calls return a value. Most asynchronous RPC systems only support calls that do not return a value. An analysis and comparison of various asynchronous RPC systems were presented in this paper. Among several asynchronous RPC interfaces, the Mercury interface seems to be appropriate to use in high-performance computing systems where a large volume of data needs to be transferred.

Key words: Distributed systems, remote procedure call, synchronous RPC asynchronous RPC, parallelism, throughput, inter-process communication

Citation: Sharma, G., 2022. Asynchronous RPC interface in distributed computing system. Inform. Technol. J., 21: 1-7.

Corresponding Author: Gajendra Sharma, Department of Computer Science and Engineering, Kathmandu University Dhulikhel, Nepal

Copyright: © 2022 Gajendra Sharma *et al.* This is an open access article distributed under the terms of the creative commons attribution License, which permits unrestricted use, distribution and reproduction in any medium, provided the original author and source are credited.

Competing Interest: The authors has declared that no competing interest exists.

Data Availability: All relevant data are within the paper and its supporting information files.

INTRODUCTION

Many distributed systems have been based on an explicit message exchange between processes. Remote Procedure Call (RPC) is a technique that follows a client/server model and allows local calls to be transparently executed onto remote resources[1]. RPC is a widely-used communication mechanism in distributed systems and applications such as Amoeba distributed operating system.

Sprite Net system and Andrew File System, the local function parameters are serialized into a memory buffer and then sent to buffer to a remote target where deserialization takes place and the corresponding function call gets executed. Libraries that implement such a technique can be found in various domains such as web services with Google Protocol Buffers[2]. RPC realization can also be achieved using a more object-oriented approach with frameworks such as CORBA or Java RMI[3] where abstract objects and methods can be distributed across a range of nodes or machines. In a distributed system, the various steps of an application workflow need to be distributed. Most of these RPC systems are synchronous and hence fail to exploit fully the parallelism inherent in distributed applications. This severely limits the kind of interactions the distributed application can have, resulting in lower performance. To achieve concurrency, the user has to resort to other means such as light-weight processes (threads) or the low-level inter-machine message-passing mechanism. If the host operating system does not support thread as in the case of Unix, costly heavy-weight processes have to be used instead. Both of these solutions are not attractive to the users. The first solution is difficult to debug and does not scale well in a large distributed environment. The second solution is much more difficult to use than the RPC mechanism. Because of this, various asynchronous RPC systems have been designed and implemented to achieve higher parallelism while retaining the familiarity and simplicity of synchronous RPC. Asynchronous RPC calls do not block the caller and the replies can be received as and when they are needed, thus allowing the client execution to proceed locally in parallel with the server invocation.

The asynchronous RPC mechanisms included in the discussion are Athena Non-blocking RPC, NCA Maybe RPCo Sun Batching RPC, Remote Pipes, Stream (Promises), Future and ASTRA. The comparison is based mainly on the following characteristics of the asynchronous RPC mechanisms" Support for the receipt of the reply message, Transport protocols order of delivery of call and replies messages, Call semantics, Optimization for low-latency or high-throughput.

The design of an asynchronous RPC mechanism is motivated mainly by the need to achieve high parallelism while retaining the simplicity and familiarity of the RPC abstraction. The limited degree of parallelism can be achieved by creating multiple lightweight processes (threads) for each RPC call[3]. This allows the client to make multiple calls to multiple servers and still be able to execute in parallel with the servers. The existing RPC interface follows a client-server architecture where a remote call results in different steps depending on the size of the data associated with the call. Every RPC call sent through the interface results in the serialization of function parameters into a memory buffer, which is then sent to the server using the network abstraction layer interface. Therefore, it is required to limit memory copies at any stage of the transfer, especially when transferring large amounts of data[4,5]. Therefore, if the data being sent is small, it is serialized and sent using small messages, otherwise, a description of the memory region that is to be transferred is sent within this same small message to the server. A large amount of data needs to be transferred to several nodes in a distributed system. Therefore, the resulting overhead during communication among several nodes needs to be minimized and there should be the efficient transfer of a large volume of data. The main purpose of this study was to analyze and compare various asynchronous RPC systems.

LITERATURE REVIEW

A large data can be transferred in a network file system by making use of remote procedure call, where the serialization of arbitrary data structures takes place resulting stream of bytes, which are sent to a remote resource, which can de-serialize and get the data back from it. By providing a network abstraction layer, the RPC interface gives the ability to the user to send small data and large data efficiently, using either small messages or Remote Memory Access (RMA) types of transfer that fully support one-sided semantics present on recent. With the popularity of the Internet and the improvement of information technology, digital information sharing increasingly becomes a trend. Several universities pay attention to the digital campus and the construction of the digital library has become the focus of the digital campus. Manageable, authenticated and secure solutions are needed for remote access to make the campus network be a transit point for the outside users. Remote Access IPSEC Virtual Private Network provides the solution of remote access to e-library resources, networks resources and so on very safely through a public network[4].

ASTRA is built within the framework of SHILPA-a Distributed Computing Environment[5]. The main design objective of SHILPA is to provide a generic distributed computing platform for building distributed applications on the interconnection of local area networks in a heterogeneous environment. ASTRA calls are similar to stream and future calls in that they can defer receipt of results. The client can make an ASTRA call in the C language using the following primitives: RPCXID RPC. Cintasycail (cinthandler, service, call_option, ...)

Decoupled and Asynchronous Remote Transfers (DART)[6,7] project provides a similar type of work but DART is not defined as an explicit RPC framework. A large amount of data is transferred using a client/server model from applications running on the compute nodes of an HPC system to local storage or remote locations. DART requires explicit requests to be sent by the user and there is no inherent limitation for the integration of such a framework.

A similar project named I/O Forwarding Scalability Layer (IOFSL) makes use of RPC to specifically forward I/O calls. It defines an API that locally serializes function parameters and sends them to a remote server, where they can in turn get mapped onto file system-specific I/O operations. IOFSL not only sends a specific set of calls, like the ones that are defined through the API but also handles a various set of calls, which can be dynamically and generically defined. IOFSL allows support for dynamic connection as well as fault tolerance. In addition, it defines two types of messaging i.e. unexpected and expected.

Sandia National Laboratories' Network Scalable Service Interface (Nessie) system provides a simple RPC mechanism originally developed for the Lightweight File Systems project[4]. It has an asynchronous RPC Mechanism. The RPC interface of Nessie directly relies on the Sun XDR solution which is mainly designed to communicate between heterogeneous architectures, even though practically all High-Performance Computing systems are homogeneous.

It provides a separate mechanism to handle bulk data transfers, which can use RDMA to transfer data efficiently from one memory to the other and supports several network means of transport. The Nessie client uses the RPC interface to push control messages to the servers.

DESIGN CRITERION

Existing synchronous RPC systems are designed for low-latency to improve the response time, whereas the asynchronous RPC systems are mostly designed for high-throughput. It is desirable to structure an asynchronous RPC system such that either low-latency or high-throughput can be achieved. In this case, the user can specify explicitly the optimization needed at run-time and mix low-latency calls with high throughput calls. There are many criteria required in the design of an asynchronous RPC system.

Firstly, an asynchronous RPC system must have the look and feel of an asynchronous RPC system, except that the client does not wait for a reply after making an asynchronous call. In this case, the client may or may not be able to defer receipt of return replies. In addition, all calls should be received and executed by the server in the order called by the client to preserve the correct call semantics. Therefore an asynchronous RPC system should retain all the benefits that a conventional synchronous RPC system has to offer and yet allow parallel execution of the client and the server.

Secondly, an asynchronous RPC system must be designed to be transport independent to suit different types of application needs. Generally, clients and servers are involved in two kinds of interactions, intermittent exchange and extended exchange. By intermittent exchange we mean the client makes a few occasional Request-Response (RR) types of calls. By extended exchange we mean the client is either involved in bulk data transfer or makes much RR type of calls to a particular server. An asynchronous RPC system should ideally incorporate both virtual-circuit and datagram transport protocols, to allow the application to choose the best transport that meets its needs. To achieve optimum performance, a virtual circuit could be selected for extended exchange since it provides better flow and error control with negligible processing overhead. On the other hand, the datagram is more suitable for intermittent exchange due to its simplicity. Thirdly, an asynchronous RPC facility must be optimized for intra-machine calls. Existing synchronous RPC systems are designed for low-latency to improve the response time, whereas the asynchronous RPC systems are mostly designed for high-throughput. It is desirable to structure an asynchronous RPC system such that either low-latency or high-throughput can be achieved. In this case, the user can specify explicitly the optimization needed at run-time and mix low-latency calls with high-throughput calls. IPSEC Virtual Private Network establishes a safe and stable tunnel that encrypts the data passing through it with secured algorithms. It is to establish a virtual private network on Internet so that the two long-distance network users can transmit data to each other in a dedicated network channel. Using this technology, the multi-network campus can communicate securely in the unreliable public internet[8].

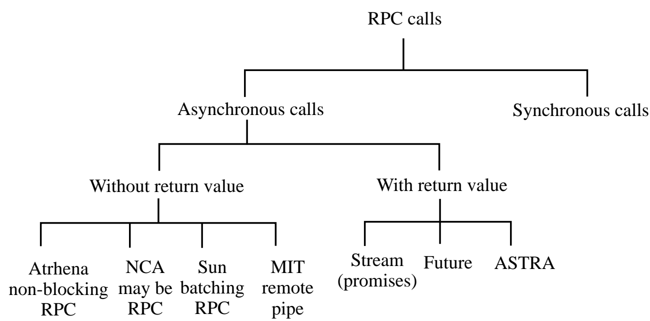

Fig. 1: Taxonomy of RPC calls

ASYNCHRONOUS RPC SYSTEM

Asynchronous RPC calls are divided into two types depending on whether the call returns a value. Most asynchronous RPC systems only support calls that do not return a value and few support both classes. A classification of the asynchronous RPC is shown in Fig. 1.

ASYNCHRONOUS RPC WITHOUT RETURN VALUE MIT PROJECT ATHENA NON-BLOCKING RPC

The objective of MIT's project Athena[9,10] was to integrate various computing and communication resources for educational purposes. Athena RPC was developed under the constraints imposed by the coherence model 0 of Project Athena. Some of the constraints included no modification to the Unix kernel, support of RPC in a heterogeneous environment and support of multiple language suites. It was implemented as a prototype in the BSIM. RPC provides both blocking (synchronous) and non-blocking (asynchronous) calls. Athena non-blocking RPC was developed primarily to improve the performance of applications where no information or status needs to be returned from the called procedure. To reduce latency, Athena non-blocking RPC sends out its call message immediately after each call. In addition, it does not differentiate between inter-machine and intra-machine calls and hence no optimization is performed for local intra machine calls.

NCA MAY BE RPC

NCA/RPC[10,11] was developed by HP/Apollo as part of the Network Computing Architecture(NCA). It provides a rich set of RPC calls for the programmer: A normal blocking RPC which is termed send-wait-reply, an asynchronous RPC which is called maybe RPC, broadcast RPC and broadcast/maybe RPC. NCA may be RPC does not attempt to buffer the call messages, the call message is sent immediately to achieve low latency. In addition, it does not optimize intramachine calls since it does not distinguish between intra-machine and inter-machine calls.

SUN BATCHING RPC

Sun ONC/RPC was developed by Sun Microsystems as part of Open Network Computing (ONC). Sun batching RPC is one of the call types provided by Sun RPC, others are normal synchronous RPC and broadcast RPC. Batching RPC allows a series of calls to be made from the client to the server. Each RPC call in the pipeline requires no reply from the server and the server can not send a reply message. The last call must be a normal blocking RPC to flush out the pipeline of calls.

Sun RPC provides two types of interface for application programmers. One is available as library routines. The other interface uses an RPC specification hmguage (RPCL) and a stub generator (RPCGEN). The RPCL is an extension of the External Data Representation (XDR) specification. To use batching RPC, one can use RPCGEN or the library routines. Sun batching RPC makes use of TCP to buffer call messages and send them to the server in one Unix write () system call. This greatly decreases the system call overhead, thus improving performance and throughput. However, no optimization is done for intra-machine calls in Sun batching RPC.

REMOTE PIPE

The remote pipe was designed to allow bulk data and incremental results to be efficiently transported in a type-safe manner. These objectives are realized using a communication model called the Channel Model. The Channel Model consists of three basic elements: remote procedure, remote pipe and channel groups. In the Channel Model, a node is similar to a process. A node may contain several channels. A channel is either a remote procedure or a pipe. The difference between a remote procedure and a pipe is that the former is a synchronous RPC while the latter is an asynchronous RPC that does not return a value. A node can import any channel exported by any other node or possibly re-export them to other nodes. This makes the channel a first-class value that can be freely exchanged among nodes. The importing node can group channels into a set called a channel group. A channel group controls the sequencing of the calls, data sent to a channel within a channel group are received in the order sent. The performance of the Channel Model can be optimized by

buffering and combining pipe calls destined for the same sink node into a single message to reduce the message-handling overhead and hence improving the throughput. This is similar to Sun batching RPC. Other optimizations include combining pipe calls with procedure calls to flush out the buffered pipe calls, combining pipe returns to a single message to reduce message handling and system call overheads, pre-allocating processes in a process pool to eliminate fork overhead and factoring packages and groups to save space. In the Channel Model, no attempt is made to differentiate between inter-machine and intra-machine calls. As a result, intra-machine calls are not optimized.

ASYNCHRONOUS RPC WITH RETURN VALUE

The RPC systems discussed to provide some form of asynchronous implementation but do not include a mechanism to support return results. This shortcoming limits the design of distributed applications to strictly unidirectional exchange from client to server. There are three choices open to the application programmer in these systems: First, program the application using synchronous RPC call and sacrifice concurrency, second, structure the application in such a way that no reply from servers is needed, Third, directly program on top of the transport layer. Because of these shortcomings asynchronous RPC systems that can defer receipt of replies such as stream (promises), future and ASTRA have been developed.

STREAM

Stream in the MIT Mercury system is the first RPC system that combines synchronous and asynchronous calls with return value in a clean and uniform way[12]. Stream provides three kinds of calls: normal synchronous RPC calls, stream calls and send. Stream calls is a kind of asynchronous RPC call with a reply message. Send, on the other hand, is similar to Sun Batching RPC and remote pipe calls, in that the client is not interested in the reply. In addition to the above three calls, stream provides a "flush" primitive that can be used to flush out the buffered call or reply messages and a "synch" primitive that will block the caller until the processing of all the earlier calls has been completed. A stream-based transport protocol such as TCP is used for transporting and sequencing the stream call and reply messages reliably. It simplifies the implementation of the stream and also provides at-most-once call semantics. However, the fact that the stream relies solely on a specific reliable stream-based transport makes it more suitable for bulk data transfer rather than low-latency calls. Moreover, the use of TCP leads to higher overheads for most transactional applications where a request-response protocol is more appropriate. Like Sun batching RPC and remote pipes, it was designed mainly to achieve high-throughput where call messages are buffered and flushed when convenient. This is to reduce the system call overhead. Although low latency can also be achieved by explicitly flushing out the calls, it is however somewhat inconvenient.

Again, no optimization is done for intra-machine calls. Recent smartphones have the capacity for capturing most of human characteristics. The biometric information is stored in the bank’s server in encrypted form which is used to verify mobile banking user’s identity over secured communication channel[13,14].

ASTRA

ASTRA[7] is built within the framework of SHILPA-a Distributed Computing Environment for the Department of Information Systems and Computer Science (DISCS) at the National University of Singapore (NUS). The main design objective of SHILPA is to provide a generic distributed computing platform for building distributed applications on the interconnection of local area networks in a heterogeneous environment. ASTRA integrates both low-latency and high-throughput communication into one single asynchronous RPC model. The user can specify explicitly whether low-latency or high-throughput is the main concern for an invocation and the system will optimize the call accordingly. It differs from other asynchronous RPC systems such as stream and future that are designed to achieve only one of them but not both. Unlike stream and future ASTRA provides highly optimized intra-machine calls. For an intra-machine call ASTRA will bypass the data conversion and network communication and directly uses the fastest native IPC mechanism provided by the local operating system. This is a unique feature provided by ASTRA. However ASTRA does not incorporate concepts like Future Set and Funnel. The flow control in ASTRA is done by the underlying transport protocol.

FUTURE

Future[15] is an asynchronous RPC provided in the CRONUS system. A future is an object that is returned after each client invocation. It can be used to claim the result of an invocation at a later stage. Futures are created and claimed by the stub procedures which are automatically generated from a

specification of the remote operations. Future also provides an abstraction called Future Set. This allows multiple futures to be grouped into a set. Future Set facilitates the management of futures and eliminates the strict ordering of claim operations. Future was implemented on both TCP and UDP. TCP is the main transport protocol supported in future. With TCP, the delivery of the call and reply messages are guaranteed. On the other hand, the future does not provide any end-to-end mechanism on top of UDP. Thus UDP-based future calls are neither reliable nor dependable. Although TCP is a sequenced transport protocol, the future makes no guarantees concerning the order of delivery of the call messages. The call messages may be reordered in the process of buffering before it is transmitted. This is a serious drawback since the order of execution in the server may be different from the order called by the client. Unlike most of the asynchronous RPC systems, future was designed mainly for low-latency. The call message is sent immediately for each request made and the returned results can be claimed in any order. In the current implementation, the future does not bypass the expensive data conversion and network communication for intra-machine calls[16]. It was investigated that most of the banks are using the card management software and most of them are not efficient enough to satisfy the need of users using it and the institutions. It is recommended that the institutions should implement few concepts to minimize the operational risk and add extra features to the existing system[13,14].

MERCURY

It is an asynchronous RPC interface specifically designed for use in High-Performance Computing (HPC) systems that allows the asynchronous transfer of parameters and execution requests and direct support of large data arguments. Mercury interface is generic to allow any function call to be shipped. Moreover, the network implementation is abstracted, allowing easy porting to future systems and efficient use of existing native transport mechanisms. It provides a reusable RPC library for use in HPC that can serve as a basis for services such as storage systems, I/O forwarding, analysis frameworks and other forms of inter-application communication[13].

The RPC interface follows a client/server architecture. It involves two types of transfers: Transfers containing typical function parameters referred to as metadata and transfers of function parameters describing large amounts of data, referred to as bulk data. Every RPC call sent through the interface results in the serialization of function parameters into a memory buffer, which is then sent to the server using the network abstraction layer interface. It is required to limit memory copies at any stage of the transfer, especially when transferring large amounts of data. Therefore, if the data sent is small, it is serialized and sent using a small message, otherwise, a description of the memory region that is to be transferred is sent within this same small message to the server. Mercury provides direct support for handling remote calls that contain large data arguments. Mercury's network protocol can support the scalability of up to thousands of clients. However, mercury does not offer support for cancelling ongoing RPC calls. Sharma *et al.*[14] proposed a framework that brings an abstraction and its infrastructure for multimedia materials storage, management, real-time review of multimedia on-demand and a shared whiteboard on web browser to distance learner. For the design of the proposed system, it is necessary to consider different requirement and management of quality of service.

DECOUPLED AND ASYNCHRONOUS REMOTE TRANSFERS (DART)

DART enables fast, low-overhead and asynchronous access to data from a running simulation, and support high-throughput, low-latency data transfers[15,17]. The primary goal of DART is to efficiently manage and transfer large amounts of data from applications running on the compute nodes of an HPC local storage or remote locations, to enable remote application monitoring, data analysis, code coupling and data archiving. The key requirements that DART is trying to satisfy include minimizing data transfer overheads on the application, achieving high-throughput, low-latency data transfers and preventing data losses. DART provides an asynchronous transfer API for single-threaded environments that are common on high-performance computing resources tuned for scientific computing. While DART is not defined as an explicit RPC framework, it allows the transfer of large amounts of data using a client/server model from applications running on the compute nodes of an HPC system to local storage or remote locations, to enable remote application monitoring, data analysis, code coupling and data archiving[17]. The key requirements that DART is trying to satisfy include minimizing data transfer overheads on the application, achieving high-throughput, low-latency data transfers and preventing data losses. Towards achieving these goals, DART is designed so that dedicated nodes, i.e., separate from the application compute nodes asynchronously extract data from the memory of the compute nodes. Participatory Sensing is more into action where people sense data from their devices

and collaborate with other people for a different purpose. As sensor data to be sent to servers, data are sent from different media like 3G Service, Wi-Fi access point, etc.[18]

CONCLUSION

This study discovered the different RPC systems which were designed to achieve higher parallelism while retaining the simplicity of synchronous RPC. Asynchronous RPC calls do not block the client and the replies can be received when they are needed. Among several asynchronous RPC interfaces, the Mercury interface seems to be appropriate to use in high-performance computing systems where a large volume of data needs to be transferred.

SIGNIFICANCE STATEMENTS

This study discovered that the mercury interface seems to be appropriate to use in high-performance computing systems where a large volume of data needs to be transferred. It can beneficial for comparing and analyzing various asynchronous RPC systems. This study will help the researchers to uncover the critical areas of RPC interfaces, Mercury interface which is appropriate to use in high-performance computing systems where a large volume of data needs to be transferred.

REFERENCES

1. Bristowe, S.K., S.M. Ghosh, M. Trew and K. Rittenbach, 2021. Virtual overdose response for people who use opioids alone: Protocol for a feasibility and clinical trial study. JMIR Res. Protocols, Vol. 10. 10.2196/20183.
2. Gottumukkala, N.R., R. Nassar, M. Paun, C.B. Leangsuksun and S.L. Scott, 2010. Reliability of a system of K nodes for high performance computing applications. IEEE Trans. Reliab., 59: 162-169.
3. Kanev, S., S.L. Xi, G.Y. Wei and D. Brooks, 2017. Mallacc: Accelerating Memory Allocation. Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. April, 2017 Association for Computing Machinery, 33-45.
4. Sharma, G., 2021. Secure remote access IPSEC virtual private network to university network system. J. Comput. Sci. Res., 3: 16-24
5. Ananda, A.L., B.H. Tay and E.K. Koh, 1991. ASTRA-An asynchronous remote procedure call facility. IEEE Comput. Soc. Digital Lib., 26: 172-179.
6. Docan, C., M. Parashar and S. Klasky, 2010. Enabling high-speed asynchronous data extraction and transfer using dart. Concurrency Comput.: Pract. Experience, 22: 1181-1204.
7. Ren, M., Q. Zhang and J. Zhang, 2019. An introductory survey of probability density function control. Syst. Sci. Control Eng., 7: 158-170.
8. Zhao, X., 2014. Distributed systems software architecture modelling and research methods. Comp. Modell. New Technol., 18: 7-11.
9. Nasr, A.A., N.A. El-Bahnasawy and A. El-Sayed, 2015. Task scheduling algorithm for high performance heterogeneous distributed computing systems. Int. J. Comput. Appl., 110: 23-29.
10. Hajikano, K., H. Kanemitsu, M.W. Kim and H.D. Kim, 2016. A task scheduling method after clustering for data intensive jobs in heterogeneous distributed systems. J. Comput. Sci. Eng., 10: 9-20.
11. Slee, M., A. Agarwal and M. Kwiatkowski, 2007. Thrift: Scalable Cross-Language Services Implementation. https://thrift.apache.org/static/files/thrift-20070401.pdf.
12. Arabnejad, H. and J.G. Barbosa, 2014. List scheduling algorithm for heterogeneous systems by an optimistic cost table. IEEE Trans. Parallel Distrib. Syst., 25: 682-694.
13. Sharma, G., 2020. Significance of biometric user authentication and authorization for mobile banking system. Int. J. Innovative Stud. Sci. Eng. Technol., 6: 7-9.
14. Sharma G. and R. Shrestha, 2020. Analysis of card management and associated operational risk in banks of Nepal. Int. J. Adv. Technol. Eng. Explor., 7: 93-101.
15. Soumagne, J., D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi and R. Ross, 2014. Mercury: Enabling remote procedure call for high-performance computing. https://www.mcs.anl.gov/papers/P4082-0613_1.pdf.
16. Sharma, G. and B.K. Bhattarai, 2018. Adaptive wireless sensor network and internet of things. COJ Rev. Res.,
17. Sharma G., S. Shakya and L.B.R. Thapa, 2019. Distributed multimedia system for distance education. J. Inst. Eng.,15: 14-24.
18. Seymour, K., H. Nakada, S. Matsuoka, J. Dongarra, C. Lee and H. Casanova, 2007. Language Services Implementation. https://www.ogf.org/documents/GFD.52.pdf.