Ping-Pong Tests On Distributed Processes Using Java bindings of Open-MPI and Java Sockets with Applications to Distributed Database Performance

Mehmet Can Boysan[0000−0001−6087−4242]

Institute of Computer Science, J. Liivi 2, Tartu 50409, Estonia
mehmet.can.boysan@ut.ee

Abstract. The use of distributed database solutions is becoming more widespread due to their higher performance and storage capabilities compared to relational databases. Since these systems rely heavily on inter-process communications, an investigation on the effect of network latency is needed. In this paper, we examine the Java bindings of Open-MPI library running on InfiniBand and TCP/IP stack and the Java Socket API for TCP/IP communications with a simple ping-pong test with analysis of latency on performance of distributed in-memory key-value stores that operate in single data centers.

Keywords: Distributed databases, network latency, Ethernet, InfiniBand, Java Sockets, TCP, MPI, Java

1 Introduction

Distributed in-memory key-value stores are becoming more widespread to be used as the preferred caching solutions because of their superior performance and higher scalability compared to relational databases. However, the distributed nature of such systems require intensive inter-process communication to be able to provide acceptable levels of consistency, availability and fault-tolerance. One way to achieve these properties is by using consensus protocols to decide on the valid state of the system. This requires a lot of message passing between the processes. Therefore, measuring the communication latency is important.

In this paper, the Java bindings of the Open-MPI library and Java Sockets have been used to develop a program that can send Ping-Pong messages between the processes to compare communication latencies on InfiniBand and 10Gb Ethernet interconnects. The motivation behind this study is to provide some results for communities that seek high performance and specifically want to use Java as implementation language.

The paper is structured as follows. First, a background and an analysis of the existing literature is given in section 2. In the next section, the testing environment and the methodology used for latency evaluation is described. The collected results are analyzed in section 4 and the report is finalized with a conclusion.

2 Background and Literature Review

NoSQL databases is being preferred instead of the relational databases since they can be deployed in a distributed fashion which allows them to scale horizontally when more power and performance is needed. One possibility is to use such systems as distributed in-memory key-value stores that are able to serve as high performing caching solutions. Some popular examples for such systems are Apache Ignite [17] and Hazelcast [13].

When such systems are deployed in a distributed manner however, they need ways to provide high availability, consistency and fault tolerance. Brewer's theorem on the other hand suggests that achieving them in case of a system failure is impossible [4]. Therefore, most of these systems apply some known consensus protocols like two/three phase commit (2/3PC) [5, 1], Paxos [2] and Raft [10]. The problem with this approach is that many messages of relatively small sizes need to be passed between the connected processes in order to reach an agreement on the valid state of the overall system. On slow networks, as the number of processes increase, the overall performance would be negatively affected because of the added overhead of these message transmissions.

In order to mitigate such a performance degradation, different interconnects such as InfiniBand can be chosen instead of 10 Gigabit+ Ethernet. There have been some studies conducted such as [8, 6] that compare these technologies by doing point-to-point communication benchmarks which show that InfiniBand outperforms 10 Gigabit Ethernet in terms of communication latency in the tested High Performance Computing (HPC) environments. As for the inter-process communications, any parallel communication library or language such as PCJ [9] or Titanium [3] can be chosen, but we prefer to stick with the well-established Message Passing Interface (MPI) [14] as the message passing solution.

Today, large vendors like Amazon provide highly scalable cloud infrastructures that use 10Gb+ Ethernet instead of InfiniBand interconnect which do not provide better performance than a typical mid-range Linux cluster [7]. In some cases, setting up an in-house cluster might not be feasible, hence, sticking to a plan offered by such vendors might still be the preferred way for the businesses to fulfill their requirements. This means that such systems need to rely on the 10Gb+ Ethernet interconnect which might prevent the system from reaching its full potential.

Another point in this discussion is that the distributed database solutions that use Java as their implementation language might not take full advantages of the native C implementations of the MPI communication standard. We have identified a number of Java implementations of this interface namely, mpiJava [15], MPJ Express [16] and also found that the Open-MPI distributions started including the Java bindings of their C implementations [11]. Although, some latency analysis comparing Java Open-MPI bindings with the original implementation is done, there seems to exist no literature that compares any MPI Java implementation with Java's TCP/IP socket layer in terms of communication latency. Hence, this paper intends to fill this gap by providing some experiments in this regard.

3 Test Environment

3.1 Test Hardware and Software

The tests were done on the *Rocket cluster* located in the High Performance Computer Center of University of Tartu [19]. Table 1 illustrates the hardware properties of a single compute node in the cluster. At most 10 compute nodes (out of 135) were used in the tests without any modifications on their existing hardware or software. Each compute node uses CentOS 7.4 as its operating system. Open-MPI version 1.8.4 and Oracle Java 8 with development kit (JDK) and runtime environment (JRE) version 1.8.0_25 are used.

Table 1: Hardware Specifications of a Single Compute Node in the Rocket Cluster

CPU	2xIntel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz (20 cores total) (4 CPU cores used in tests)
RAM	64GB RAM
Storage	1TB HDD (860GB usable)
Network	4x QDR Infiniband, 8 Mellanox switches
	10Gbit/s Ethernet, ConnectX-3 MT27500 Mellanox switch

3.2 Methodology

The software created for this investigation can be found in [12].

Ping-Pong Test Setup: In the code, a distributed process object, referred to as a `Role` was created. A `Role` can be thought of as a single processing unit and is supposed to be deployed to different cluster nodes. In this case, it is created to serve ping-pong messages. In a cluster of N `Role`s, a single leader is selected which sends a ping message to all the `Role`s in the cluster including itself. The time starting from the *first ping* and ending with the *last pong* gives the round trip latency of a single leader trying to send a consensus message. Individual ping-pong message round trip latencies between the `Role`s were also recorded.

Each `Role` uses Java's Socket API for TCP/IP communication, and `MPI.iRecv` routine for MPI's InfiniBand and TCP/IP communications. A single message listener accepts each connection in a loop and once a message arrives, a separate thread processes it. Messages are sent in a non-blocking manner, meaning that each message is sent in a dedicated thread without waiting for its completion. Messages are sent with Java sockets for TCP/IP and for MPI, with `MPI.iSend` routine.

The Ping and Pong messages are initially constructed as Java objects which are first marshalled into JSON strings and then converted into byte arrays prior

to sending. Upon receival, the byte array is first converted back to a JSON string and unmarshalled back to its Java object representation for further processing. Also note that a fixed message size of 353 bytes was used.

Test routines were implemented to send the ping-pong messages between the processes in separate phases called "warmup" and "full-load". The warmup phase was run with 100 iterations and the full-load phase was run with 500 iterations. A result collector object was created and was run on a separate thread to collect the latency results. All the test phases were repeated 200 times to minimize the effects of the cluster network load on the results.

Both the Java socket and MPI latency tests were run separately on a number of compute nodes varying from 1 to 10. They were submitted as batch jobs to the job scheduler of the cluster [20].

OSU Latency Test in Java: In order to complement the work implemented in the previous section, we also wanted to measure the one-way communication latency of the Java sockets and Open-MPI in a simpler way. Therefore, we decided to use the standard OSU Micro-Benchmarks, located in [18]. These tests are offered by the *Ohio State University* to provide ways to measure the network communication performance of MPI configurations. Specifically, the point-to-point *osu_latency* test was chosen that would allow us to benchmark Java socket and Open-MPI latencies in the cluster. However, the OSU tests are implemented in C, therefore to allow for a direct comparison, the *osu_latency* test was also converted to Java.

We implemented three different versions of this test, one using the Open-MPI library and the others using Java Sockets. The Open-MPI version is a one-to-one translation of the test. The first socket test opens and closes a socket each time a message needs to be sent, whereas the second one keeps the connection open until all message transactions are done. The first method provides a better fault tolerant system, where if an endpoint is dead, we would get immediate notification that the socket connection could not be established. On the other hand, the second one provides a performance efficient methodology due to the eliminated overhead of opening and closing a socket when sending a message. Also note that, in all the tests, inter-process communications are done synchronously in a single Java thread.

Finally, the tests were run with 1000 iterations, including the Java Virtual Machine (JVM) warmup period.

4 Experimental Results

4.1 Ping-Pong Latency Test Results

Fig.1 (a) shows the average round trip latency (in milliseconds) of a single ping message sent from a single process to N nodes varying in sizes from 1 to 10 when Java's Socket API and Open-MPI library are used on different transmission protocols. Fig.1 (b) on the other hand shows the average round trip latency (in

milliseconds) of a message sent from one node to the other, indicating the average round trip latency of the point-to-point communication on Java sockets and Open-MPI library. The results show that Java TCP/IP sockets are more stable and efficient than the Java bindings of Open-MPI running on both InfiniBand and TCP/IP stack when node count is less than 8. And the latency values in all the cases converged when the node count is 8 and more. However, what was interesting to observe was that Open-MPI on both InfiniBand and TCP/IP stack showed higher average latency results when the number of nodes are kept between 1 and 3. Since the tests were run multiple times with sufficient number of iterations, the problem seems not to be related with Java's internal mechanisms like failure in optimizing the runtime performance with just-in-time compilation or overhead associated with the garbage collection. Instead, this seems like an internal issue with the Java bindings of Open-MPI.

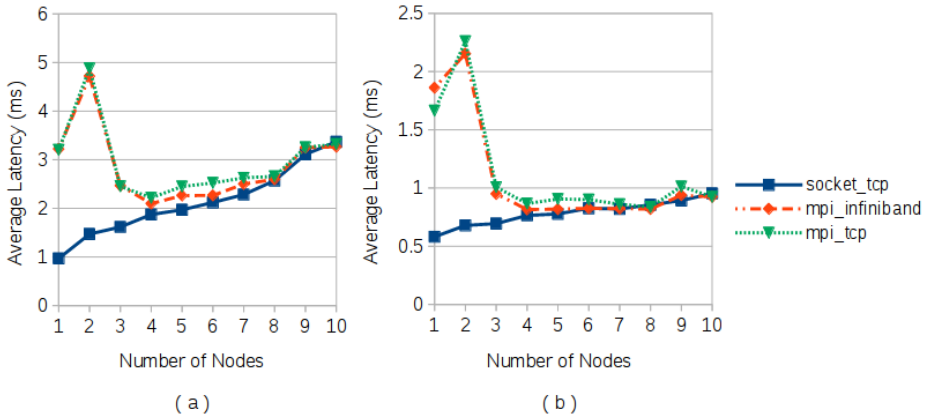

Fig. 1: (a) Average round trip latency (in milliseconds) of a message of size 353 bytes sent from a single node to N nodes. (b) Average round trip latency (in milliseconds) of a point-to-point communication between 2 nodes, using messages of size 353 bytes.

4.2 OSU Point-to-Point Latency Results

Fig.2 shows the average one-way latency (in microseconds) of point-to-point communication of two nodes in varying data sizes. These are the end results of the OSU point-to-point one-way latency benchmarks performed with Java Sockets that use TCP/IP, C implementation and Java bindings of Open-MPI that use InfiniBand and TCP/IP as the underlying communication stack. The Java socket solution included the methodology with opening and closing a socket when a message needs to be delivered each time (java_socket_tcp_open_close) and the one with an always-open connection throughout the test's lifecylce (java_socket_tcp_always_open). It can be seen that the best latency values are

obtained with the C implementation of Open-MPI running on InfiniBand interconnect. The slightly higher latency values observed in Open-MPI Java bindings compared to Open-MPI C version seem to have originated from the added overhead of calls being made to the C compiled binaries of Open-MPI. However, the reason for the high latency jump from data of sizes 32768 to 65536 in Open-MPI versions running on TCP interconnect is currently not known. Java sockets on the other hand, performed worse than the provided MPI solutions. Although for small data sizes, the "always open" socket solution performed better than the "open-close" solution, similar results were observed when data size is over 4096 bytes. This means that the added overhead of opening and closing a socket becomes irrelevant when a data with larger size need to be transferred.

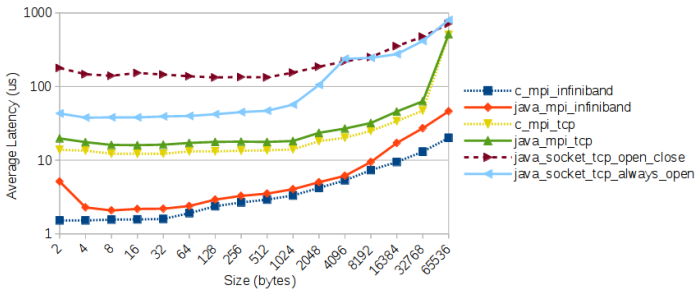

Fig. 2: Average one-way latency (in microseconds) between two nodes on varying data sizes.

The average latency when using OSU tests was lower with a factor of over a thousand than those described in section 4.1. The reason for it is because of the complexity of the project, which was created to serve as a basic simulation of the consensus messaging between the distributed processes. From data marshalling to result collection, a lot of internal processing happens even on a single ping-pong request-response pair, which we believe is expected in such systems.

The other important point was to see that the performance of the socket and the Open-MPI solutions gave opposite results when Fig.1 and Fig.2. are compared. The main distinction between the two benchmarks is that OSU tests use a synchronized communication strategy with a single Java thread to send and receive messages, whereas the project described above runs on a multi-threaded environment asynchronously. We can conclude that the Java bindings of Open-MPI is not optimized well enough to run concurrently.

5 Conclusion

We have compared the average round trip latency values of the Java bindings of the Open-MPI library running on InfiniBand and TCP/IP stack with the Java's Socket API for TCP/IP communications. The comparison was made with a

simple ping-pong test that is intended to reflect a basic distributed database system that uses consensus protocols to reach an agreement on the valid state of the overall system. In addition, we have provided the results collected for the point-to-point osu_latency benchmarks implemented in Java to compare the average latency values between the C implementation and Java bindings of the Open-MPI library and implementations made with the Java Socket API.

We have seen high differences in latencies when osu_latency and ping-pong test results are compared. We concluded that this is caused by the additional functionality that needed to be implemented to give support for a distributed database solution. We have also observed that, regardless of the interconnect, Java bindings of Open-MPI performed poorly than the Java Sockets when multiple Java threads are used to provide concurrent communication.

Although further analysis is needed to investigate the latency performance with a newer version of Java bindings of Open-MPI, these results show that Java Sockets should be preferred instead to develop a distributed database system that will operate in single data centers.

Acknowledgements

Thanks to B.K Muite and A. Jasinski for helpful discussions and suggestions. This work was carried out in the High Performance Computing Center of University of Tartu. This work is partially funded by the Estonian Research Council [IUT34-4].

References

[1] D. Skeen and M. Stonebraker. "A Formal Model of Crash Recovery in a Distributed System". In: *IEEE Transactions on Software Engineering* SE-9.3 (1983), pp. 219–228. ISSN: 0098-5589. DOI: 10.1109/TSE.1983.236608.

[2] L. Lamport. "The Part-Time Parliament". In: (1998). URL: https://www.microsoft.com/en-us/research/publication/part-time-parliament/.

[3] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, et al. "Titanium: a high-performance Java dialect". In: *Concurrency and Computation: Practice and Experience* 10.11-13 (1998), pp. 825–836.

[4] E. Brewer. "Towards robust distributed systems". In: *PODC*. Jan. 2000, p. 7.

[5] J. Gray and L. Lamport. *Consensus on Transaction Commit*. Tech. rep. 2004, p. 32. URL: https://www.microsoft.com/en-us/research/publication/consensus-on-transaction-commit/.

[6] HPC Advisory Council. "Interconnect Analysis: 10GigE and InfiniBand in High Performance Computing". In: *HPC Advisory Council, Tech. Rep* (2009).

[7] K.R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H.J. Wasserman, and N.J Wright. "Performance analysis of high performance computing applications on the amazon web services cloud". In: *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE. 2010, pp. 159–168.

[8] R. Ismail, N.A.W.A. Hamid, M. Othman, R. Latip, and M.A. Sanwani. "Point-to-point communication on gigabit ethernet and InfiniBand networks". In: *International Conference on Informatics Engineering and Information Science*. Springer. 2011, pp. 369–382.

[9] M. Nowicki, Ł. Górski, P. Grabrczyk, and P. Bala. "PCJ-Java library for high performance computing in PGAS model". In: *High Performance Computing & Simulation (HPCS), 2014 International Conference on*. IEEE. 2014, pp. 202–209.

[10] D. Ongaro and J. Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Philadelphia, PA: USENIX Association, 2014, pp. 305–320. ISBN: 978-1-931971-10-2. URL: http://dl.acm.org/citation.cfm?id=2643634.2643666.

[11] O. Vega-Gisbert, J.E. Roman, and J.M. Squyres. "Design and implementation of Java bindings in Open MPI". In: *Parallel Computing* 59 (2016), pp. 1–20.

[12] M.C. Boysan. *mboysan/ping-pong-mpi-tcp: Ping pong test with TCP and MPI*. https://github.com/mboysan/ping-pong-mpi-tcp. (Accessed on 06/04/2018).

[13] *Hazelcast the Leading In-Memory Data Grid - Hazelcast.com*. https://hazelcast.com/. (Accessed on 06/04/2018).

[14] *MPI Documents*. http://mpi-forum.org/docs/. (Accessed on 06/04/2018).

[15] *mpiJava Home Page*. http://www.hpjava.org/mpiJava.html. (Accessed on 06/04/2018).

[16] *MPJ Express Project*. http://mpj-express.org/. (Accessed on 06/04/2018).

[17] *Open source memory-centric distributed database, caching, and processing platform - Apache Ignite*. https://ignite.apache.org/index.html. (Accessed on 06/04/2018).

[18] *OSU Microbenchmarks*. http://mvapich.cse.ohio-state.edu/benchmarks/. (Accessed on 07/04/2018).

[19] *Rocket Cluster - High Performance Computing Center, University of Tartu*. https://hpc.ut.ee/en_US/web/guest/rocket-cluster. (Accessed on 07/04/2018).

[20] *SLURM - High Performance Computing Center, University of Tartu*. https://hpc.ut.ee/en_US/slurm. (Accessed on 07/04/2018).