RDMA Optimization on MongoDB

Fan Lu Tao Fang Sen Li Ziyu Zhang Yufeng Chen

Director: Prof. Hong An `han@ustc.edu.cn`

University of Science and Technology of China

{fanlu95, ftao, vincentx, zzymm, wuthering125}@mail.ustc.edu.cn

Abstract

MongoDB is a NoSQL database which stores the data in form of key-value pairs. It is an open-source, document database which provides high performance, high availability and automatic scaling. MongoDB is widely used in data intensive applications such as web and Artificial Intelligence(AI) service. The communication module in MongoDB is called Transport Layer, which is implemented by TCP/IP-based Boost.Asio. Nowadays, the InfiniBand interconnect network is widely configured on high performance clusters, which features higher throughput and lower latency than the traditional TCP/IP network. However, the official MongoDB has not adopted this technology.

In this paper, we present a compact design of MongoDB using Remote Direct Memory Access (RDMA) over InfiniBand. Taking advantage of RDMA technology, we improve the performance of MongoDB standalone instance and clusters.

As MongoDB is still under development, all of our work following discussed was conducted on the v4.0.0 of MongoDB. The performance evaluation shows our RDMA-based design obtains 3.58X/6.16X at peak *Put/Get* operations speed-up on ordinary data size and 6.60X at peak speed-up on neural network module query over the original system.

Key Words: MongoDB, NoSQL, RDMA, InfiniBand, Web and AI applications

1 Introduction

The NoSQL database movement came about to address the shortcomings of relational database and meet the requirements of data intensive applications such as AI and web services. Among many open-source NoSQL databases, MongoDB [1] stands out for its high performance, high availability and automatic scaling. It provides high performance data persistence by reducing I/O activities, and can be deployed as replica set or shard distributes to achieve high availability and high scalability on clusters. A record in MongoDB is a document which is composed of fields and value pairs. This kind of structure corresponds to native data types of many programming languages, reduces the need for expensive joins and make dynamic schema support fluent polymorphism.

However, the existing MongoDB implementation is built upon Boost.Asio [2], which is a cross-platform C++ library for network and low-level I/O programming. Boost.Asio provides a great degree of portability, but at the price of performance due to the high latency and limited bandwidth of TCP/IP. Therefore, we conduct a research on whether MongoDB could benefit from using high performance network such as InfiniBand and its RDMA capability.

The rest of this paper is organized as follows:

In Section 2, we introduce our analysis on MongoDB and its Transport Layer as well as some views related to InfiniBand and RDMA. Section 3 presents our contribution to MongoDB Transport Layer and explains how we design and implement the RDMA-based MongoDB in detail. Section 4 demonstrates the evaluation of our newly designed MongoDB. Finally, Section 5 concludes our work and describes future directions for RDMA-based MongoDB.

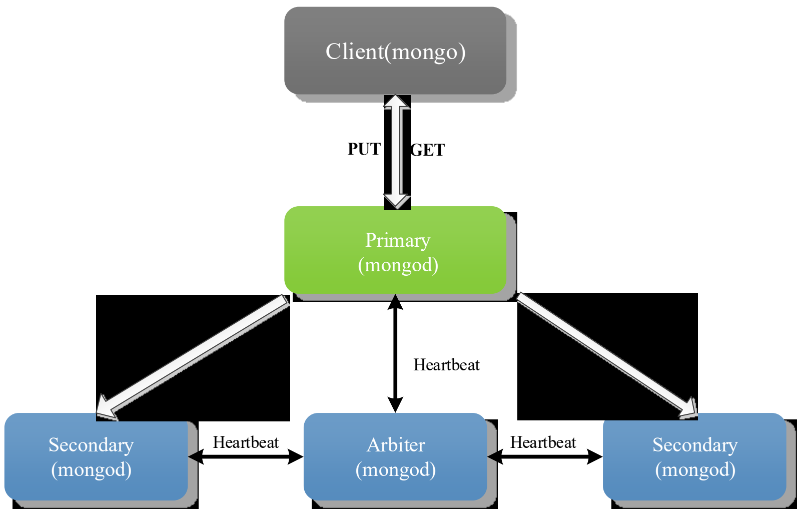
2 Preliminary

2.1 MongoDB

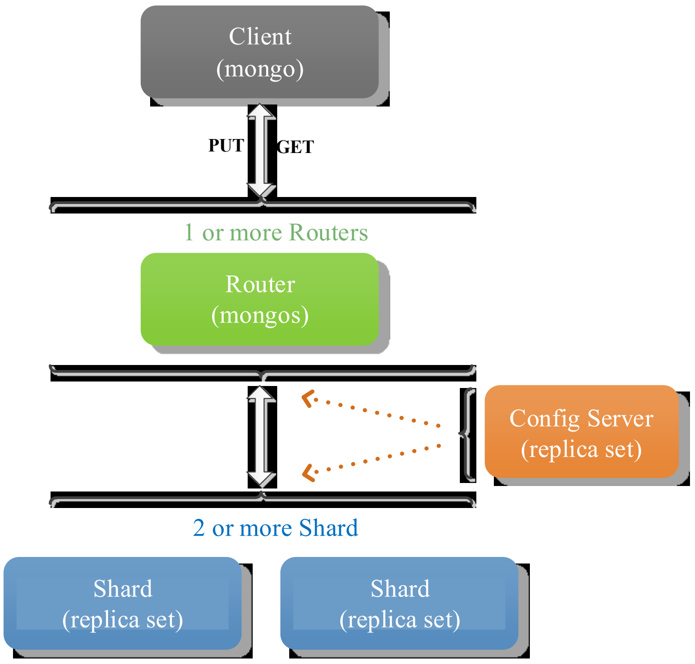
In this section, we briefly discuss our study on MongoDB and its source code.

MongoDB consists of three major components as mongod, mongos and mongo client. Mongod follows a position of the primary daemon process in the MongoDB system, handling data requests, managing data access, and performing background management operations. Mongos is a routing service for MongoDB shard configuration, which behaves identically to any other MongoDB instance from the perspective of the client. Mongo client gives an interactive JavaScript shell interface to MongoDB, which is required to manage the database directly. It also provides a fully functional JavaScript environment for use with the MongoDB, which enables us to write our own benchmarks.

With these three major components, Users can deploy MongoDB as replica set or sharded cluster as shown in Figure 1. A replica set in MongoDB is a group of mongod processes that maintain the same data set, which provides a level of fault tolerance and increased read capacity. It contains several data bearing nodes (one primary node and some secondary nodes) and optionally one arbiter node. The primary node receives all write operations from clients and records all the changes of its data sets in operation log (oplog). The secondaries replicate the primary's oplog and apply the operations to their data sets such that their data sets reflect the primary's. Each mongod process requires heartbeats to check others' status. If the primary is unavailable, the arbiter will participate in election in order to re-elect an eligible secondary to be the new primary. A sharded cluster consists of the mongos, config servers and shards. The mongos acts as a router, providing an interface between clients and the sharded cluster. The config servers store the configuration settings for the cluster while the shards maintain a subset of the data set, and they both should be deployed as a replica set. Thus, shared cluster mode is suitable for a huge cluster but not a small cluster. Standalone instance (with only one mongod process) is also supported in MongoDB but just recommended for test and development.



(a) Replica Set Architecture



(b) Sharded Cluster Architecture

Figure 1: MogoDB Cluster Architecture

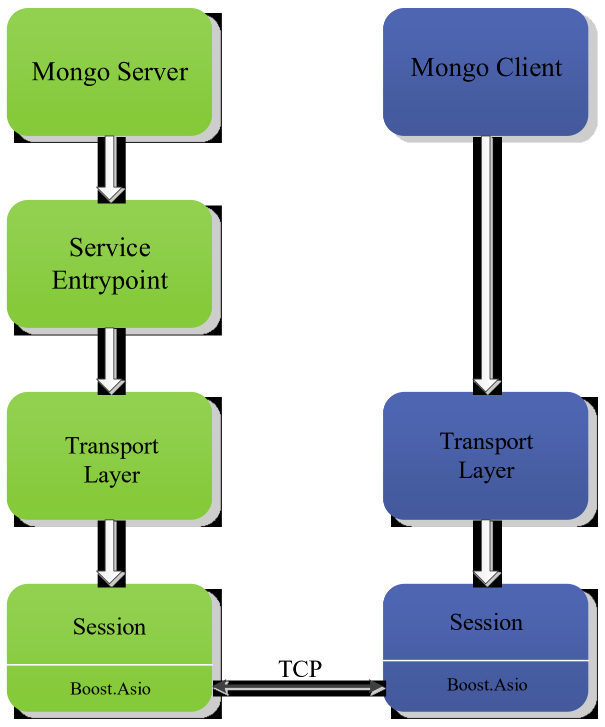


Figure 2: A Brief Hierarchical Structure of MongoDB

MongoDB implements the basic communication module named Transport Layer (located in "/src/mongo/transport") to adapt "server - client" mode. Figure 2 describes a simple hierarchy about the two sides. The Service Entry Point is the interface between the Transport Layer and mongod/mongos server. On server side, the Transport Layer will wait for a TCP connection after created. For each connection, it will create a Session which starts with receiving message. While on the other side, the Transport Layer will connect to a remote host and create a Session which starts with sending message. After the connection is established, the Transport Layer starts to communicate with the other side with Boost.Asio which is based on TCP/IP protocol. The detailed communicate logic will be introduced in Section 2.2.

2.2 Transport Layer

In this part we briefly introduce the detailed logic of Transport Layer, the communication module of MongoDB which is built upon Boost.Asio.

The Transport Layer creates Session objects and maps them internally to **transport::Endpoints**, and each Session will loop in a receive-message, process-message and send-message cycle.

On the server side, the Service State Machine (SSM) holds the state of a single connection/session and represents the lifecycle of each request as a state machine. The SSM will schedule tasks on the Service Executor based on current state. It is the glue between the Service Entry Point and Transport Layer that ties network and database logic together.

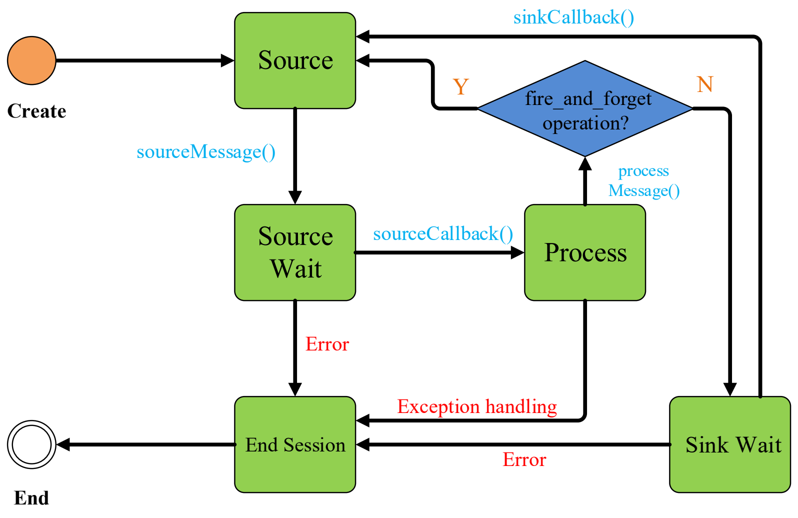


Figure 3: Service State Machine

Figure 3 shows the SSM logic.

The SSM includes seven states, which is shown in the following:

- **Created** : It is set in the SSM constructor, which means that a new SSM is created for a given session context. No operation has been performed except the initialization in this state.

- **Source** : It will be set in **_runNextInGuard()** for the first time, which means that the SSM is ready to request a message from network after **Created**. It also can be set in **_sinkCallback()** after the server has sent a message, or in **_processMessage()** when server has handled a fire-and-forget operation. In this state, SSM will schedule the receiving task on the Service Executor.

- **Source Wait** : It will be set in **_sourceMessage()**, which means that the SSM is waiting for the new message to arrive from the network. In this state, the SSM will call **Session::sourceMessage()** or **Session::asyncSourceMessage()** to receive a message, and both functions will call **Session::sourceMessageImpl()**. The **Session::sourceMessageImpl()** is implemented by Boost.Asio synchronous and asynchronous read.

- **Process** : After the SSM calls **_sourceMessage()** and receives a message from the network, the SSM state will be set to **Process** in **_sourceCallback()**. The SSM will schedule a processing task on the Service Executor immediately to execute the message through the database.

- **Sink Wait** : If the server need to respond after message processing, it will call **_sinkMessage()** and change the state to **Sink Wait**. **Session::sinkMessage()** or **Session::asyncSinkMessage()** will be called in this stage to respond, and both functions will call **Session::write()** which is supported by Boost.Asio synchronous and asynchronous write.

- **End Session** : If there is an error during communication or handling request, the state will change to **End Session**. The SSM will be invalid after this state.
- **Ended** : This state means the session has ended, and it is illegal to call any method.

On the client side, the communication logic is simpler than server's. The client will send message, receive message and process message in a loop after the connection is established. Client handles the sending and receiving tasks by calling **Session::sinkMessage()** and **Session::sourceMessage()** directly.

2.3 InfiniBand and RDMA

InfiniBand [3] is an industry standard switched fabric that is designed for high-speed, general purpose I/O interconnect nodes in high end clusters.

One of the main features of InfiniBand is RDMA. This feature allows software to remotely read or update memory contents of another remote process without any software involvement at the remote side [4]. It uses a queue-based model. Each Queue Pair (QP) has a certain number of work queue elements. Upper-level software can place a Work Request(WR) on a QP. The WR is then processed by the Host Channel Adapter(HCA). When work is completed, a Work Completion (WC) notification is placed on the Completion Queue(CQ). Upper level software can detect completion by polling the CQ or asynchronous events.

InfiniBand software stacks also provide a driver for implementing the IP layer, which exposes the InfiniBand device as just another network interface available from the system with an IP address. It is often called "IP over IB" or IPoIB for short.

3 Contribution

3.1 Transport Layer RDMA model

As the upper module of Transport Layer is relatively complex, we focus our attention on the low level of Transport Layer. We find that both server and client side need to call four member functions of Session to complete the message transmission tasks, which are **sinkMessage()**, **asyncSinkMessage()**, **sourceMessage()** and **asyncSourceMessage()**.

For the sending tasks, message sink functions will write messages into Asio stream via TCP and return the status. For the receiving tasks, message source functions will read messages via TCP from Asio stream and return the message with status. So we mainly modified these four member functions of Session to porting RDMA into MongoDB. **sinkMessage()** and **asyncSinkMessage()** will send messages by **RDMA_WRITE_WITH_ IMM** operation.

And **sourceMessage()** and **asyncSourceMessage()** will receive messages by **RDMA_READ** operation.

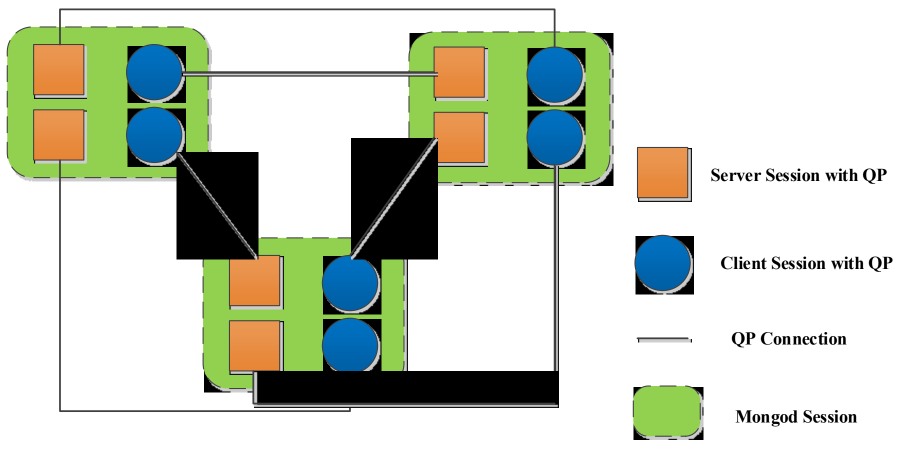


Figure 4: The Model of RDMA Transport Layer

Figure 4 shows our RDMA Transport Layer model (take replica set as an example). Each mongod instance has several sessions which can be divided into server sessions and client sessions. For each pair of associated server session and client session, we create a QP connection, ant then use RDMA operations to handle data transmission.

3.2 Detailed Design

The RDMA based session work flow is shown in Figure 5. When the session is created, it will get a TCP socket connection and the local InfiniBand device information. The session will send message and additional local InfiniBand device information in the first sending, and it will be received in the first receiving of remote host. Then the RDMA connection will be established, and all messages will be transmitted via RDMA.

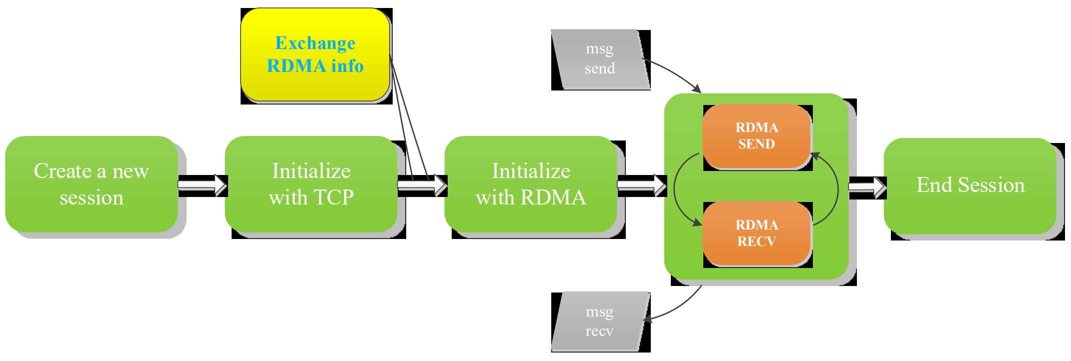


Figure 5: The Workflow of RDMA Communication

For each session, we register only two message buffers whose size are the max value of a single message. Every session repeatedly sets a message in a specified remote memory buffer and gets a message from a local memory buffer. The remote memory buffer will not be covered since the session will be blocked until remote session receives the last message and responds. Our RDMA-based Transport Layer can build message directly by WC information while the original version needs to build message header and message content separately. It also reduces the memory allocation and copy cost.

We also tried to create a new thread for each session to accelerate the transmission. The thread will maintain the RDMA memory buffer status by polling the CQ and reading the WC information. Thus, the four message transmission functions merely need to submit the WR into the QP when the RDMA memory buffer is available. However, this optimization method didn't improve the performance significantly, yet it consumes much more CPU resources. We abandoned this method due to high CPU utilization rate and limited improvement.

The mainly modified source code is located in "/src/mongo/transport/session_asio.h ".

3.3 MongoDB-RDMA

The Transport layer is one module of the MongoDB, all we need to do is just linking it to the MongoDB. We need to add "-libverbs" in liking stage since the "infiniband/verbs.h" is included in "session_asio.h". Thus, we modified the SCons build script located in "/src/mongo/SConsript".

4 Evaluation

4.1 Experimental Setup

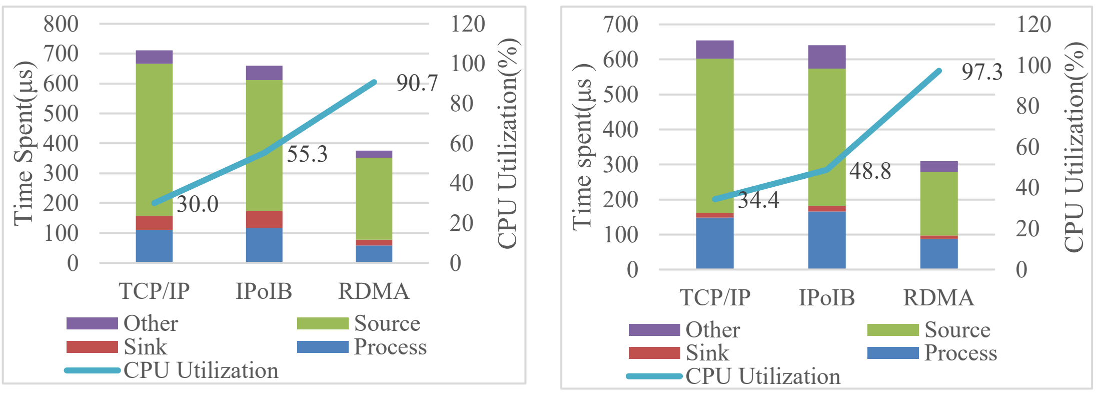
The cluster configuration we use for evaluation is: 8 Sugon nodes with dual Intel Xeon E5-2660 processors, 1000GB HDD and 128GB RAM. Each node runs Centos Linux release 7.3.1611. The nodes are connected with both 1GigE and Mellanox 40G QDR InfiniBand.

We use MongoDB version 4.0 in all our experiment. And we use the filesystem in HDD and memory. The database is deployed as either a standalone instance or a replica set due to our limited nodes. We focus on the performance potentials of RDMA-based MongoDB in AI and traditional web fields. The neural network input data is around 1MB while the traditional web data is around 1KB, so we choose the data set size between 1KB and 1 MB. We also choose several typical neural network models to explore that whether MongoDB can accelerate inference tasks in AI. They are inception-v3 (91MB), resnet152 (219MB) and vgg16 (528MB).

4.2 Detailed Profilings

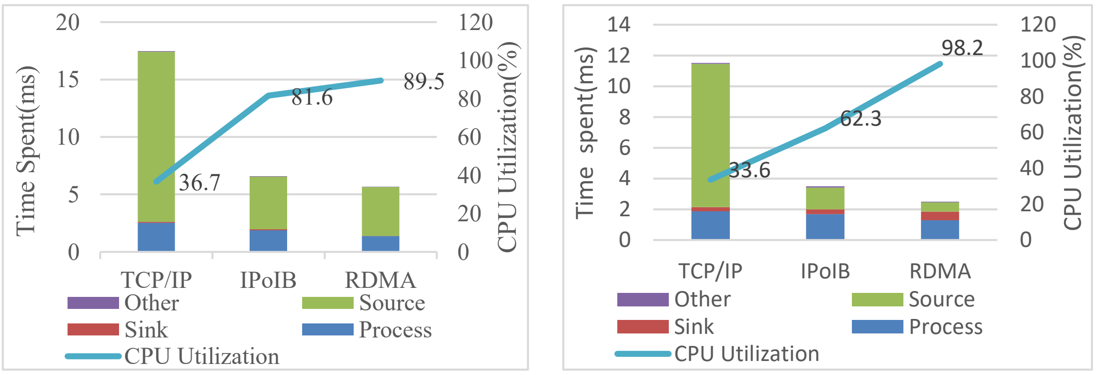
Existing MongoDB makes use of Boost.Asio for communication. As discussed in section 2.3, RDMA can reduce significant overhead caused by TCP/IP. In order to understand the performance improvement of RDMA-based MongoDB, we conduct detailed profilings to measure the time spent in different stages of *Put*/*Get* operations [5]. The major steps involved in an operation for mongo servers are: (1) "Source" for waiting and receiving the request message. (2) "Process" for processing the request through database. (3) "Sink" for sending response message. (4) "Other" for SSM management.

The HDD filesystem profiling results, shown in Figure 6 and Figure7, indicate the time taken at different steps for *Put*/*Get* operations with 1KB and 1MB payload.



(a) Put 1 KB (b) Get 1 KB

Figure 6: Put/Get Processing Time Breakdown and CPU Utilization (1KB Record Size, HDD)

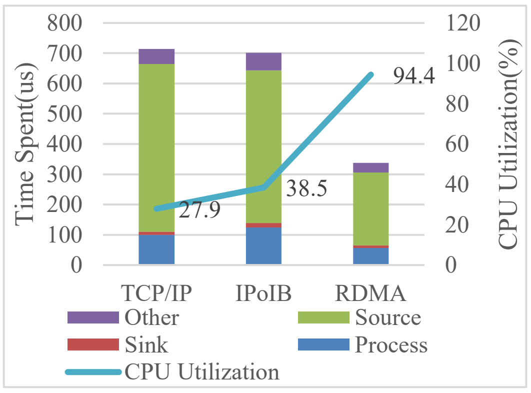


(a) Put 1MB (b) Get 1MB

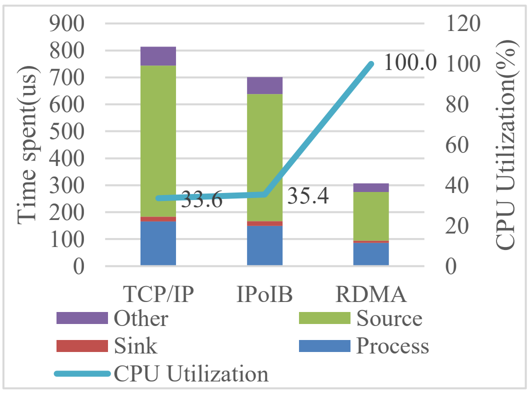
Figure 7: Put/Get Processing Time Breakdown and CPU Utilization (1MB Record Size, HDD)

For 1KB payload *Put* operation (Figure 6 (a)), the time of four stages for RDMA are 274µs, 58.8µs, 18.6µs and 24.1µs, where as it is 509.2µs/437.4µs, 111.4µs/115.9µs, 45.5µs/58.2µs and 44.5µs/47.7µs for 1GigE/IPoIB. The process schedule strategy is well-designed, which allows Process stage to benefit from communication improvement. Our design achieves a performance improvement of 1.89X/1.76X times over 1GigE/IPoIB (TCP version). And we noticed that the Source stage is the major factors contributing to the overall operation latency, while Sink and Process only accounts for less than 30% for all results. So the network latency is the major performance bottleneck in 1KB *Put* operation. Similar trend is observed for *Get* operation, as indicated in the Figure 6(b).

For 1MB payload *Get* operation (Figure 7 (b)), the time of four stages for RDMA are 602.3μs, 1203.3μs, 550μs and 37.5μs, where as it is 9307.6μs/1427.1μs, 1867.5μs/1627.4μs, 275.8μs/313.3μs and 68.9μs/73.8μs for 1GigE/IPoIB. The different results between 1KB *Put* and 1MB *Get* are that: (1) the ratio of Sink stage improved a lot in 1MB *Get* due to the large response message size. (2) the operation time of 1GigE are much higher than others in 1MB *Get* due to the limited bandwidth. Our design achieves a performance improvement of 4.64X/1.40X times over 1GigE/IPoIB (TCP version). For 1MB payload *Put* operations (Figure 7 (a)), We can observe a similar trend, except that the ratio of Sink is very low, which is caused by the small response message in *Put* operation.

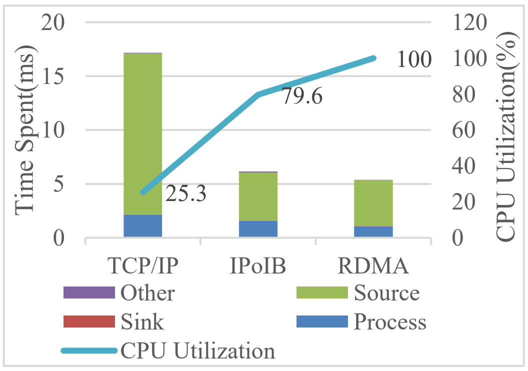


(a) Put 1 KB

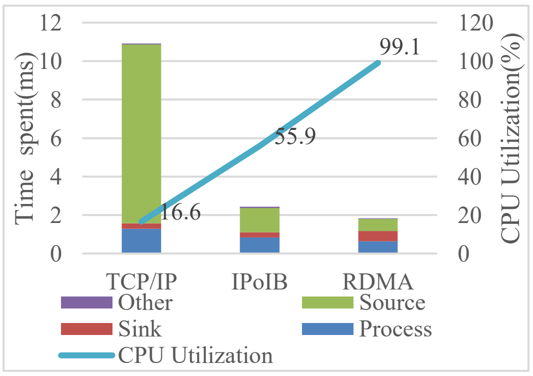


(b) Get 1 KB

Figure 8: Put/Get Processing Time Breakdown and CPU Utilization (1KB Record Size, Memory)



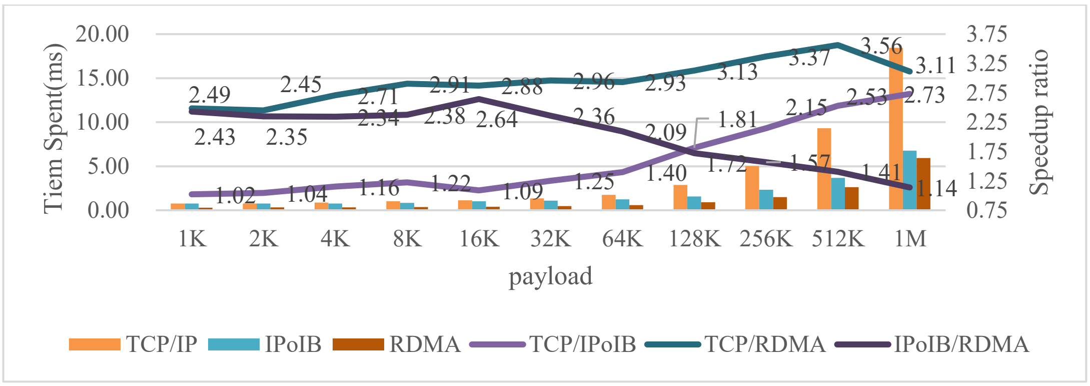
(a) Put 1MB



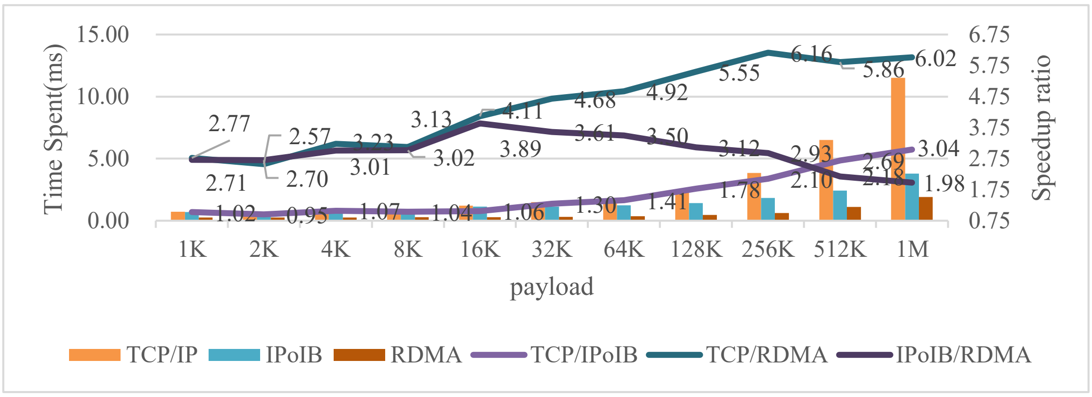
(b) Get 1MB

Figure 9: Put/Get Processing Time Breakdown and CPU Utilization (1MB Record Size, Memory)

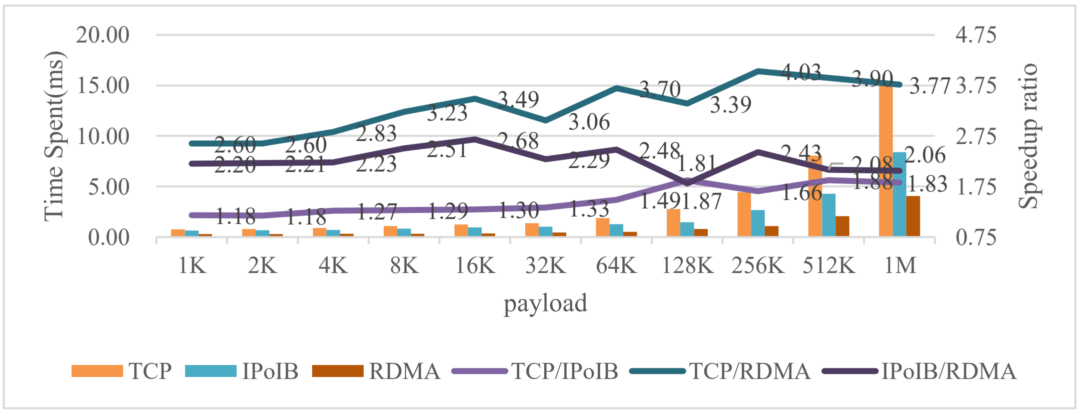
The profiling result in memory filesystem, shown in Figure 8 and Figure 9, also exhibit the time taken at different steps for *Put*/*Get* operations with 1KB and 1MB payload. Since the procedure of reading and writing in HDD is not required, all results improve from several percent to tens of percent compared to the HDD's. However, only the Process stage time is reduced obviously, since it is the only stage that related to filesystem. For 1MB *Put* and *Get* operations, the time of Process stage for RDMA are 1079.02µs and 641.59µs where as it is 1381.85µs and 1293.30µs in HDD filesystem. And for the overall time, our design achieves 1.05X and 1.36X speedup compared to the performance in HDD filesystem, which has gained more improvement than IPoIB and TCP/IP. Since the proportion of overhead in HDD can not be ignored, the RMDA-based version, which access memory directly, will gain more performance improvement.



(a) 100% Put Operation



(b) 100% Get Operation

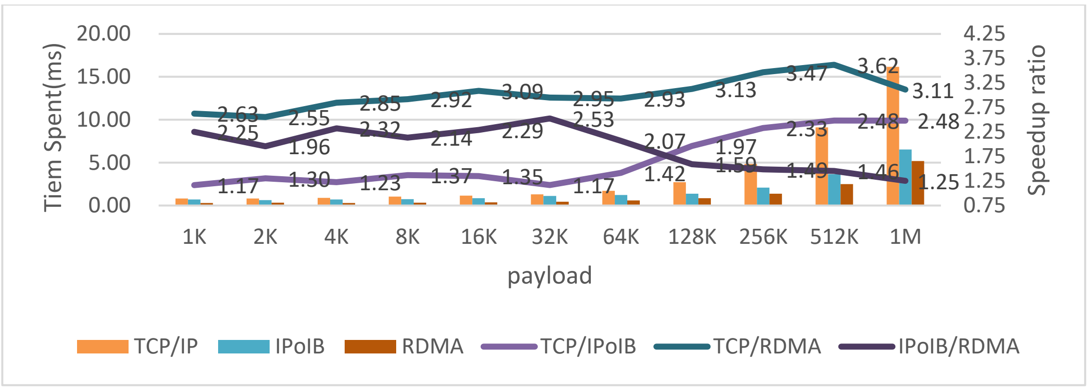


(c) 50% Put + 50% Get Operation

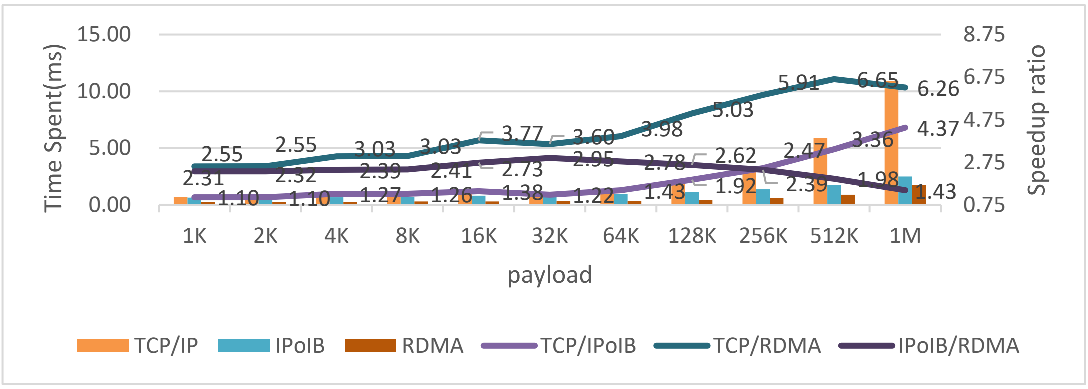
Figure 10: Standalone Server Single Client Performance Evaluation(HDD)

The CPU Utilization data is also shown in these figures. For *Put* operation, The RDMA mode consumes the most CPU resource while the TCP mode consumes the least. It proves that the CPU resource consumption is related to the speed of operation processing. We can observe a similar trend for *Get* operation.

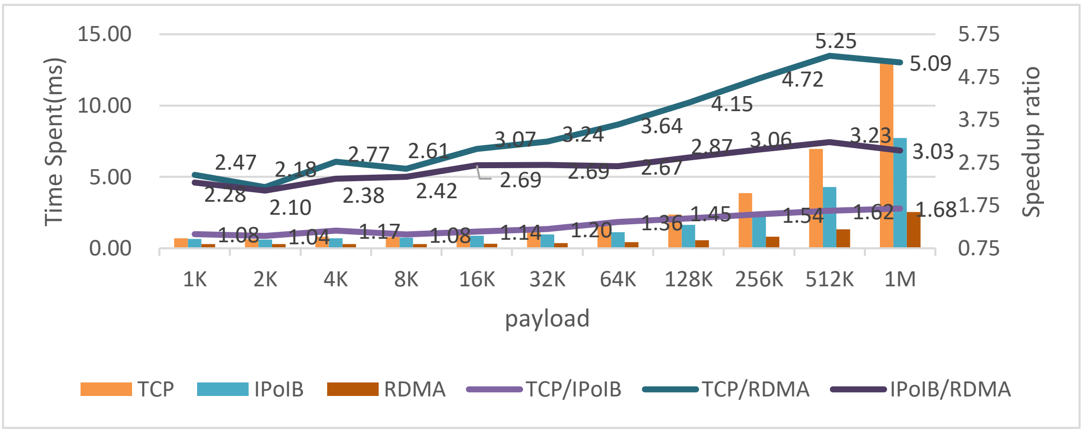
To sum up, RDMA-based design substantially decreases the communication time and reduces the operation latency perceived by end user.



(a) 100% Put Operation



(b) 100% Get Operation



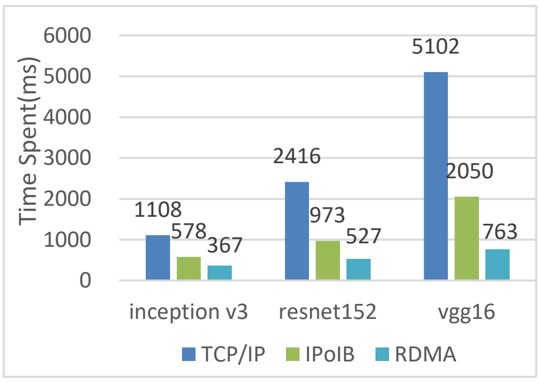
(c) 50% Put + 50% Get Operation

Figure 11: Standalone Server Single Client Performance Evaluation(Memory)

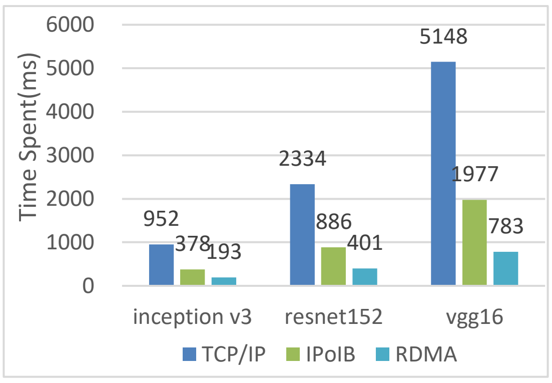
4.3 Standalone-server and Single-client

We designed the experiment of a standalone server with single client to evaluate the latency of basic MongoDB *Put* and *Get* operations. This experiment issues *Put* or *Get* operations of a specified size and measures the average time taken for the operation to complete. As mentioned in Section 4.1, we choose the payload size from 1KB to 1MB.

In Figure 10, the RDMA-based design outperforms the TCP-based design for all data size and operations. For *Put* operations, RDMA-based design is 2.45X to 3.56X times faster than 1GigE in the data range, while the speedup of IPoIB is only 1.02X to 2.73X times. For 1KB *Put* operations, the latencies observed for 1GigE and IPoIB are 778.4μs and 761.5μs. Our design achieves a latency of 318μs, which is 2.49X/2.39X times faster than 1GigE/IPoIB. The similar improvements are observed for the *Get* and 50%*Put* -50%*Get* operations as well. We supposed that the low latency of RDMA and high bandwidth of InfiniBand contributes to this result.



(a) HDD Filesystem



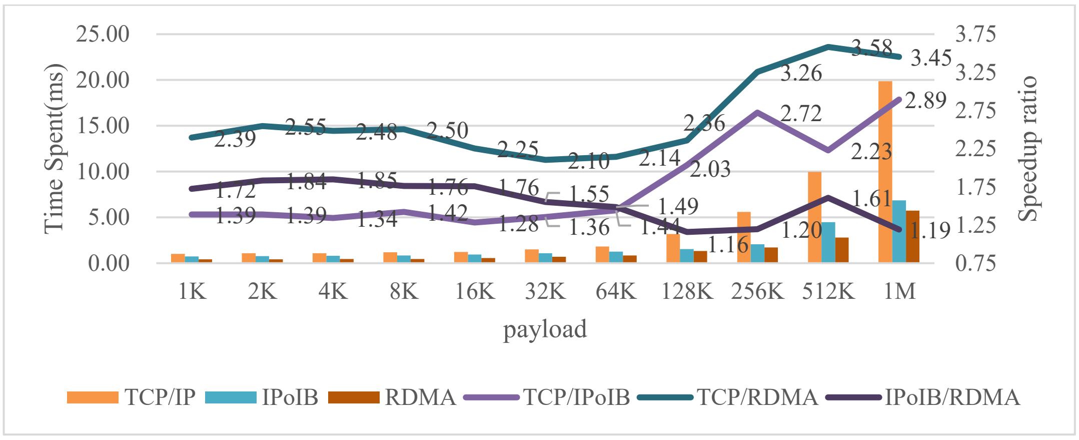
(b) Memory Filesystem

Figure 12: Get Latency of Neural Network Models in Standalone Server Single Client

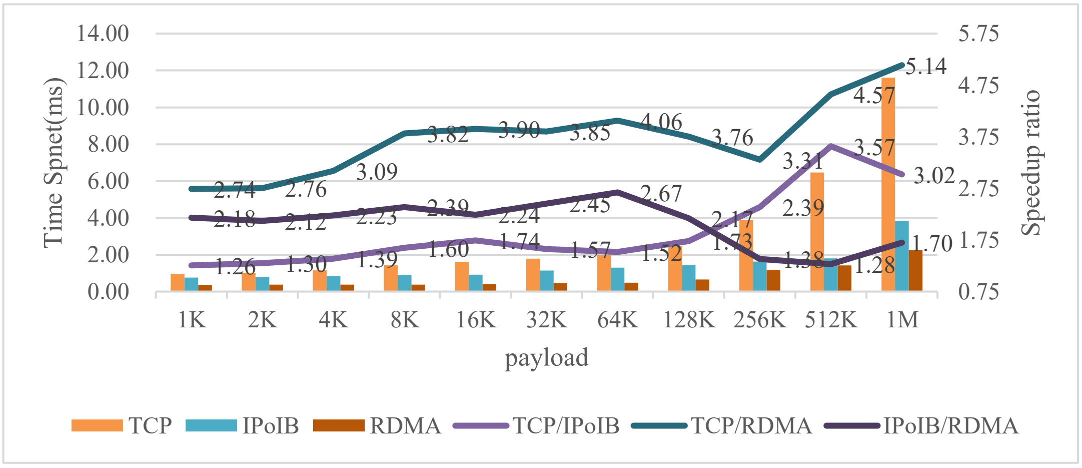
We also measured the *Get* operations latency of typical neural network models since inference application only load models from database. Figure 9 shows the evaluation on inception v3, resnet152 and vgg16, and the RDMA-based design is 3.02X/1.58X , 4.58X/1.85X and 6.69X/2.69X times faster than 1GigE/IPoIB respectively.

4.4 Multi-servers and Single-client

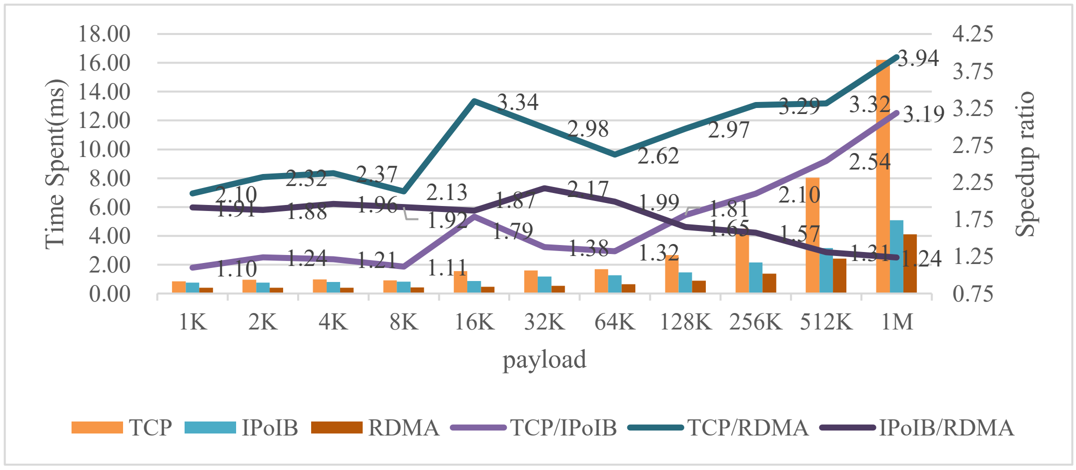
In this experiment, we deploy a replica set MongoDB cluster in four nodes, which consists of one primary node, two secondary nodes and one arbiter node. We use the same data and operation configuration as Section 4.3 to evaluate the performance in cluster mode.



(a) 100% Put Operation



(b) 100% Get Operation



(c) 50% Put + 50% Get Operation

Figure 10: Replica Set Single Client Performance Evaluation

Figure 10 shows the average latencies for all operations in replica set cluster. RDMA-based design substantially reduces the operation latencies compared to TCP-based version. The performance improvement of RDMA over 1GigE network is 2.74X to 5.17X times for *Get* operation, while IPoIB only achieves 1.26X to 3.57X times. The average operation latency for 1KB *Get* over RDMA is 351.2µs, which is 2.74X/2.18X times faster than 1GigE/IPoIB.

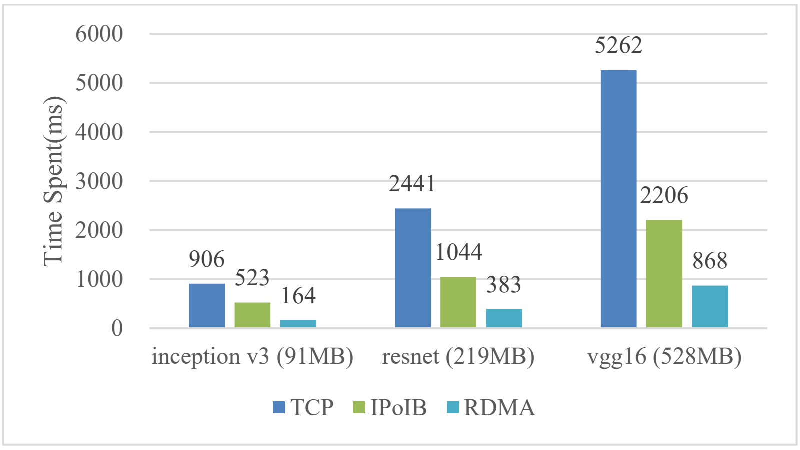


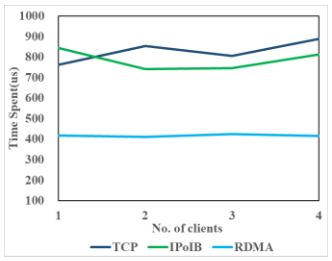
Figure 11: Get Latency of Neural Network Models in Replica Set Single Client

RDMA-based design also significantly reduces the *Get* operation for neural network models as indicated in Figure 11. It achieves 164ms/383ms/868ms latency for inception v3/resnet152/vgg16, which is 5.52X/6.37X/6.06X times faster than 1GigE and 3.19X/2.72X/2.54X times faster than IPoIB.

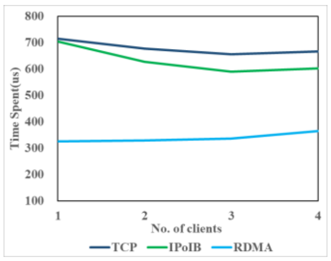
4.5 Multi-servers and Multi-clients

In this experiment, we deploy a replica set MongoDB cluster as Section 4.4 and varies numbers of MongoDB client nodes from 1 to 4, which simulates MongoDB deployment workload. We prepare client nodes in two configurations, 1 client thread per node and 2 client threads per node. For each of these configurations, we use different workloads: (1) *Put* operations with 1KB/1MB payload, (2) *Get* operations with 1KB/1MB payload and (3) 50% *Put* + 50% *Get* operations with 1KB/1MB payload.

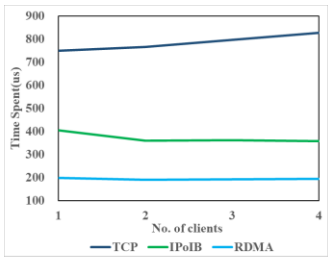
For one thread per node configuration, RDMA-based design significantly reduces all the latencies as indicated in Figure 12. For 4 clients, the *Put* operation latency is 414µs /7.7ms with the 1KB/1MB payload, which is 2.14X/6.67X times faster than 1GigE latency and 1.96X/1.20X times faster than IPoIB latency. For 4 client 1KB and 1MB *Get* operations, RDMA-based design achieves 1.83X/12.9X and 1.65X/1.45X speedup compared to 1GigE/IPoIB. Similar trend is observed for 50% *Put* + 50% *Get* operations, as indicated in the figure.



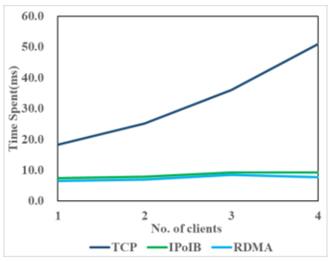
(a) 1 KB 100% PUT



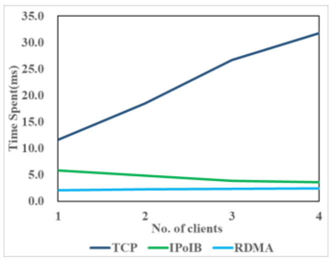
(b) 1 KB 100% GET



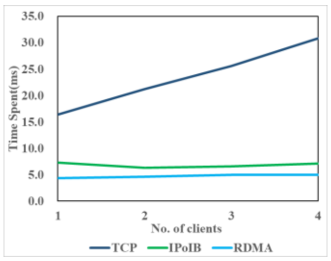
(c) 1 KB 50% PUT+50% GET



(d) 1 MB 100% PUT

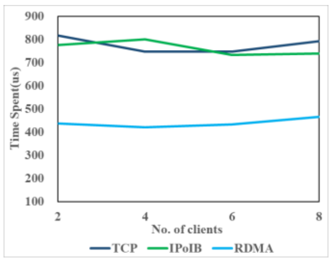


(e) 1 MB 100% GET

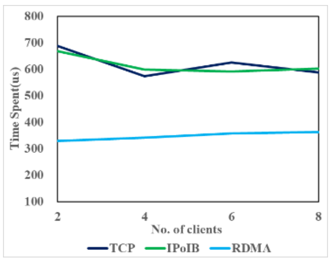


(f) 1 MB 50% PUT+50% GET

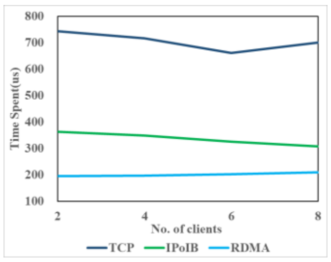
Figure 12: Multi-Servers and Multi-Clients with 1 Thread/Node



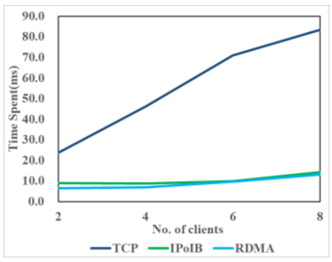
(a) 1 KB 100% PUT



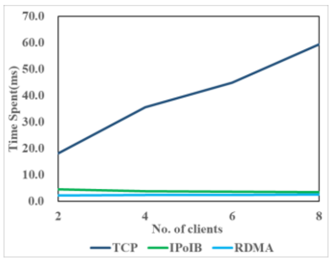
(b) 1 KB 100% GET



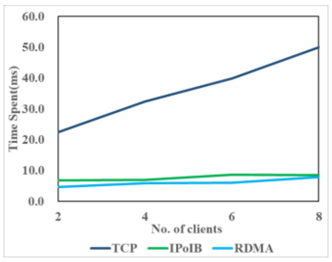
(c) 1 KB 50% PUT+50% GET



(d) 1 MB 100% PUT



(e) 1 MB 100% GET



(f) 1 MB 50% PUT+50% GET

Figure 13: Multi-Servers and Multi-Clients with 2 Thread/Node

For two threads per node configuration, similar trend is observed as shown in Figure 13. For 1KB and 1MB *Put* operations, the latencies are 465µs and 13.2ms on 8 clients. This is 1.71X/1.59X and 6.33X/1.08X times faster than 1GigE/IPoIB. Our design also improves the performance by around 1.62X/1.66X and 22.68X/1.34X times as compared to 1GigE/IPoIB for 1KB and 1MB *Get* operation on 8 clients. We can also observe the similar trend for 50% *Put* + 50% *Get* operations in the figure.

5 Conclusion and Future Work

In this paper, we took on the challenge to improve MongoDB performance by making it RDMA-capable. We performed detailed profilings to reveal the communication overhead caused by TCP/IP and the speedup of RDMA. In our evaluation, the RDMA-based MongoDB shows a low latency and good scalability over the original TCP/IP version.

As part of future work, we plan to evaluate the performance of MongoDB in a large cluster and other data size to expose further potential performance bottlenecks. We also plan to analyze the software overheads of MongoDB in detail and come up with a highly optimized and scalable design.

6 Acknowledgement

Thanks to the HPC Advisory Council for giving us this chance to explore the RDMA technology and NoSQL application.

Thanks to the author of rdmamojo.com Dotan Barak as well. Many questions have been solved according to his blog.

This work was supported by Advanced Computer System Architecture Lab of University of Science and Technology of China.

References

[1] ”MongoDB for GIANT Ideas | MongoDB”, https://www.mongodb.com/

[2] ”Boost.Asio1.66.0 ”,https://www.boost.org/doc/libs/1_66_0/doc/html/boost_asio.html

[3] InfiniBand whitepapers, http://www.mellanox.com/page/white_papers

[4] Pfister, Gregory F. "An introduction to the infiniband architecture." High Performance Mass Storage and Parallel I/O 42 (2001): 617-632.

[5] Huang, Jian, et al. "High-performance design of hbase with rdma over infiniband." Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International. IEEE, 2012.