Optimizing Redis with RDMA

Liyan Zheng, Kezhao Huang, Tianhui Shi, Zixuan Ma, Yutian Wang, Jidong Zhai
Tsinghua University

1. INTRODUCTION

Redis is an open-source in-memory key-value database. Due to the use of traditional C(TCP) Sockets, it has a great degree of portability but loss of performance.

In order to achieve high throughput, low latency and low CPU utilization in high-performance computing clusters, one approach is to use advanced functions provided by high-performance networks, such as Remote Direct Memory Access (RDMA). Therefore, we combine RDMA with Redis and propose several optimization. And we demonstrate that Redis with RDMA can offer up to 6x throughput and reduce latency by 96.3%.

2. RDMA VERBS COMPARISON

RDMA is an effective technology for high performance clusters. And three RDMA technologies in common use are RoCE, iWARP and InfiniBand (IB). InfiniBand is an edge-cutting communication standard which provides high throughput and low latency for cluster. And we choose this technology to realize RDMA based Redis in the following report.

RDMA verbs are a set of RDMA operations. They can be divided into two categories. Two-sided verbs include SEND and RECV. The other one, one-sided verbs, include READ, WRITE and so on. It is worth noting that one-sided verb is CPU-bypass, which means remote CPU does not participate in the RDMA operations. In many situations, CPU-bypass verbs have higher throughput and lower latency than CPU-involved verbs. But two-sided verbs are still useful and can be used along with one-sided verbs.

3. REDIS

Redis is an in-memory but persistent on disk database. It is single-threaded and can be scaled in terms of I/O concurrency. It can run standalone or as a cluster. It also supports pipelining to reduce the network latency between the server and the client.

3.1. Standalone mode

Redis processes tasks through an event handling mechanism. Two types of events are designed in Redis, one is file events and the other is time events. The file events are designed for passively triggered events, which are mainly network tasks, while the time events are designed for actively triggered events. Each file event is bound to a file descriptor and will be triggered when the file descriptor becomes readable or writable. Obviously, a file event can be triggered multiple times and the time when the event is triggered is unknown. In contrast, each time event has a known trigger time and will be deleted after being triggered.

In the management design of two types of events, the file events adopt an array method, while the time events adopt a linked list. Since we need to repeatedly query the state of the file events, and there is only occasional additions and deletions, using an array to store the file events can greatly reduce the time for a single query. On the contrary, although the time events will be added and deleted frequently, the total amount of time events that exist at the same time is very small, it is more appropriate to use a linked list to store the time events.

Listing 1: Event Loop of Redis

```
while not STOP: // Run until STOP is true
    beforeSleep()
    t_now=getTime()
    t_next=nextTimeEvent() // Return the triggered time
    d=t_next−t_now
    EventList=pollFileEvents(d) // Return an array of file events
    afterSleep()
    for event in EventList:
        if event.readable: // True when read ready
            event.readProc()
        if event.writable: // True when write ready
            event.writeProc()
    processTimeEvents()
```

In each iteration of the Redis event loop, it will first process tasks that need to execute before sleeping. Then it will query the time of the next time event to be processed by traversing the linked list, and get the time remaining. This is to take advantage of the interval between two time events to handle file events as much as possible. When at least a file event is triggered or the wait time is longer than the time remaining, it will exit sleeping and return an array containing the triggered file events. Then it will process tasks that need to execute before sleeping. For each file event in the array, it determines whether to process a read task or a write task according to whether it is ready to read or write. Finally, it can handle all time events that have arrived. This event loop will not stop running until the variable STOP is TRUE.

In order to implement the above mechanism, an IO multiplexing strategy is needed. The original Redis provides four implementations of the above event handling mechanisms, including evport, epoll, kqueue and select. These implementations all hinder the further performance improvements and replace with our implementation which supports RDMA. .

3.2. Cluster mode

3.2.1. Master-Slave Replication. In Redis, users can use a SLAVEOF command to make one server replicate another one. We call the server being replicated a master, and call the other one a slave. Since Redis can change the slave to a new master to continue running once the original master fails, this replication mechanism improves the fault tolerance of Redis.

Replication could be mainly divided into the following three steps:

(1) Build a connection between the slave and its master.
(2) The master send its database to the slave in the form of an RDB file. The slave accepts and loads the RDB file, and updates the state of its database to that of the master's database. This phase is called Sync.
(3) The master keeps sending the write command it accepts and executes to the slave. In this way, the slave can keep the database state consistent with that of the master. In addition, the slave also sends a REPLCONF command to the master at the frequency of once per second to detect the current network connection status and whether there is any commands lost. This phase is called Command Propagate.

3.2.2. Redis Cluster. Redis cluster, which shares data through sharding and provides assignment and failover functions, is a distributed database scheme provided by Redis.

A Redis cluster is usually composed of multiple nodes, which are servers that run in cluster mode. In the beginning, each node is independent, and they add each other to their respective clusters through handshaking.

The entire database of the Redis cluster is divided into 16384 slots. Each key in the database belongs to one of the slots. Each node in the cluster can handle at least 0 slots and at most 16384 slots. After every slot have been handled by some node, the cluster goes online. We can use the CLUSTER ADDSLOTS command to assign some slots to one server.

When a client sends a command related to a database key to a server, the server checks whether the slot to which the key belongs is handled by itself. If not, it will send a MOVED message to the client telling it which server is handling the slot. After receiving the MOVED message, the client sends the original command to the correct server which is handling the slot.

The nodes of the cluster will periodically send some messages, such as sending PING to see if the network connection with other nodes is stable. It should be noted that although masters and slaves are also cluster nodes, the communication between them does not pass through the cluster, because they treat each other as their own clients, that is, the way of communication between the master and the slave is similar to that between the client and the server.

4. REDIS WITH RDMA

4.1. Standalone mode

Through our evaluation, we can conclude that the communication speed between Redis server and client has a great impact on the latency and throughput when requests are posted to server from the client. So our first goal is to change the communication method between client and server into RDMA.

To realize this, we have got four main kinds of RDMA verbs to choose from, which is RECV, SEND, READ and WRITE. Due to the different characteristic of server and client, we figure out their communication modes case by case:

4.1.1. Client to Server. According to former research and experiments, WRITE has the lowest latency and highest throughput among all verbs. However, referring to the Section 3, the server utilizes a poll function to get notified once messages come in. Meanwhile, when posting SEND and RECV, we can also maintain a complete queue (CQ) to get informed when the operation has been accomplished. So it is simple and natural for us to implement RDMA poll function in place of the system poll function while WRITE cannot enable us to know whether the message has come in. Once the memory region is registered and the key and pointer are passed to the client, the client can write anything to that buffer without the server knowing it. WRITE is a one-sided verb, while socket write and SEND are two-sided verbs, making it difficult to make full use of WRITE which is of high efficiency.

Although SEND and RECV are easy to implement, we decide to use WRITE to achieve the best performance. Before that, we have to figure out how to solve the problem of notification.

We divide the receive buffer of the server into several parts which are called slot, each slot is the unit where one message is written. Every time the client sends a message, it will write the message into one slot. We can use slots to get serve notified. When the client writes to the server, besides the message posted, it also adds a flag at the last byte of the slot. The server will read the flag byte of the slot when polling file events. When the flag is set, the server can register the coming message and fire it.

After firing all the incoming messages, server will process them and send feedback to the client. So we can construct poll function for WRITE as well.

Another problem coming to us when applying WRITE into Redis is that WRITE cannot carry the length of the message, but Redis needs to know it when processing the message. Our solution is to encode the length information at the bottom of the slot, after which the server can decode and get the length. This method will make the message longer than it was. But according to the characteristic of RDMA, when the length of the message is short, it will not affect the latency and throughput. Meanwhile the ratio of the redundancy information is only a tiny part when message is getting larger. So the attached information will not affect the performance of the database.

4.1.2. Server to Client. After processing the message, the server has to pass message back to the client. If there are many replies waiting to be sent, we will encode them in one message to reduce traffic. As the server is likely to be faced with a large number of clients, we choose SEND instead of WRITE due to its better scalability. And as clients are aware of how many messages are sent to them at most, they can post RECV in advance to meet all the SENDs.

So we finally implement WRITE-SEND client-server model and achieve 6x throughput than socket version client-server.

4.2. Cluster mode

In terms of communication, the Redis cluster mode has communication between cluster nodes and between masters and slaves which the standalone mode does not have. To saturate NIC, we should execute multiple server processes on one machine. Due to the small amount of communication between nodes, which has little impact on the performance of the cluster, we decide to optimize only the communication between the masters and the slaves.

Since the master sends the write commands to the slave as a client, as mentioned in the Section 4.1, We use WRITE to send write commands for the lowest latency and highest throughput, just as we did in standalone mode.

For logical unification and ease of implementation, we also use WRITE to send messages from the slaves to its master.

The original version of Redis utilizes socket to set up connection and send confirmation message between masters and slaves. To make use of the reliability of TCP, we keep this part when initializing the cluster. When the cluster is running with piles of messages coming in, the server will also WRITE messages to its peer (master or slave). As the client is also writing to cluster, the speed of client-server communication and master-slave communication is theoretically equal, avoiding wasting time backup the message. This makes us achieve roughly equal throughput even when we backup the updates to the slaves.

4.3. Optimization

We also put forward several general optimizations which also improve performance a lot.

4.3.1. Selectively signaled and inline SEND. Signaled SEND and WRITE operations generate a work completion element(CQE) in CQ. This notifies the process a work request has finished. However, this notification is often unnecessary and causes an extra DMA operation. We choose to selective post only one signaled SEND and WRITE operations in a bunch of them.

For common SEND and WRITE operations, RNIC gets data from main memory by a DMA before it transfers the data exactly. Infact, the DMA can be avoided by sending the data together with work request to RNIC. We use an inline work request for small messages to get better performance and get a decrease in average latency by 22.00% in the best case.

4.3.2. Send response immediately. Responses in Redis server are sent back to clients in the beforeSleep function in listing 1. Redis chooses this method to realize the best throughput at the price of larger latency. But for high performance RNIC, sending response immediately after events are processed is a better choice for both latency and throughput. This innovative method takes advantage of the low cost of RDMA post operation and disperses its communication traffic. According to fig **??**, we get a decrease in average latency by 13.60% and a increase in throughput by 14.20% under 4 client connections.

5. EVALUATION

We now evaluate Redis with RDMA on cluster Gorgon to see how Redis with RDMA meets our requirement. The experiment is about latency and throughput. First, we run Redis server on standalone mode. Then, we run them on cluster mode. At last, we compare latency and throughput between Redis with socket and Redis with RDMA, so that we can figure out how RDMA optimizes Redis.

5.1. Experiment setup

To evaluate Redis with RDMA on real deployment, we use an 8-node cluster Gorgon which is showed in table I.

We run latency and throughput experiment of standalone mode server on 7 machines in Gorgon, the 6 client machines run up to 24 client processes each. In cluster mode experiment, we use 3 machines for client and 3 machines for servers. But a master server will not run with its slave on the same node.

We evaluate latency and throughput together. During the experimentation, each client gets its group of operations, sends request, and waits for response, keeps a record of latency and finish time. After all operations finished, we can calculate the average latency and throughput of this client. When using multi clients, we use MPI_Barrier to sync the start time of all process. Once one process has finished all operations of its own, we also stop the operations of other processes and count the operations completed before that. In this way, we can calculate the throughput and average latency of this number of connections.

CPU	Intel(R) Xeon(R) CPU E5-2670 v3 CPU
Memory	128 GB
OS	Ubuntu 16.04.4 LTS
Interconnect network	ConnectX-4 EDR 100Gb/s InfiniBand MT27700
OFED	1.0-0.36.g0ac39b8.43101
Storage	SATA SSD 6.0Gbps

Table I: Gorgon hardware environment

5.2. Benchmark

The main workload parameters affecting the latency and throughput are the length of value and the number of connections. We use YCSB to generate workload with Zipf distribution. We generate 13 groups of workload with different length of value. Each has 1 million operations. We split one group into n parts when evaluating n clients to make the client have similar numbers of operations.

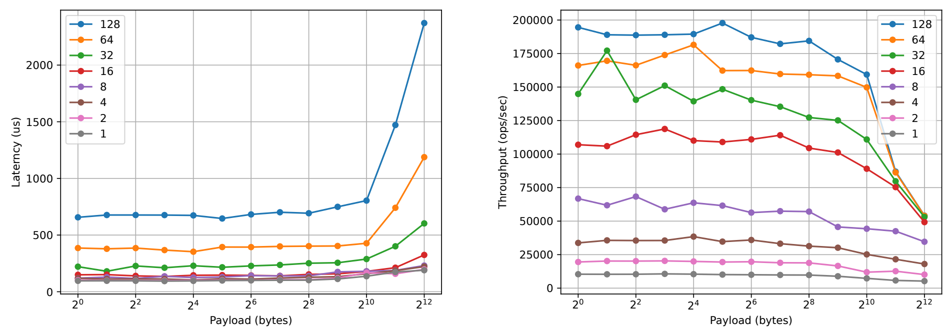

Fig. 1: Latency and throughput of Single machine Redis with socket in different connections

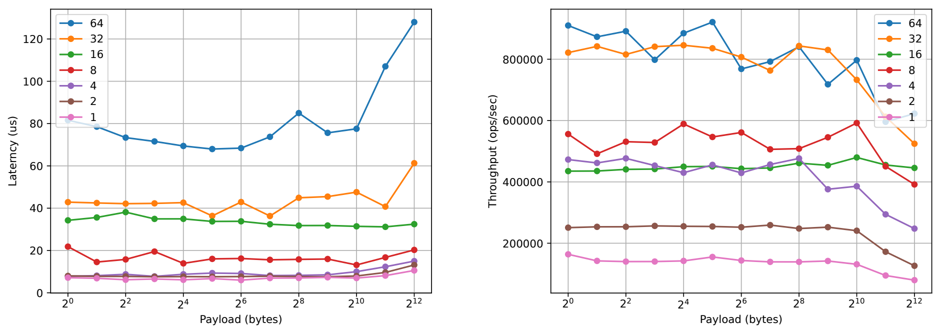

Fig. 2: Latency and throughput of Single machine Redis with RDMA in different connections

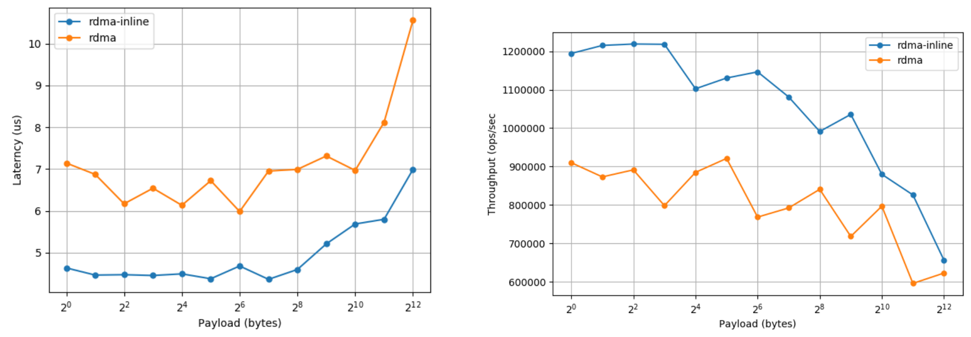

Fig. 3: Latency and throughput of Single machine Redis with RDMA and inline in different connections

5.3. Standalone mode

Fig 1 plots the latency and throughput of Redis with socket. For 32-byte items, Redis with socket has a max throughput of 197762.94 ops/sec and average latency of 98.79 us.

Fig 2 plots the latency and throughput of Redis with RDMA. We can see, 32 clients can saturate the server's throughput. For 32-byte items, Redis with RDMA has a throughput of 921133.17 ops/sec and average latency of 6.72 us.

Fig 3 plots the latency and throughput of Redis with RDMA and inline optimization. With the same number of clients, we can saturate the server's throughput. For 32-byte items, it has a throughput of 1130438.88 ops/sec and average latency of 5.638 us.

5.4. The bottleneck of throughput

It is clear that the maximum throughput of a Redis server cannot saturate IB bandwidth. When transferring the maximum data size in our test (4KB), RDMA Redis can reach up to 20.29 Gbps while original Redis can only achieve 1.67 Gbps. However, even if RDMA Redis can be of 12 times of original Redis, it is still far from the bandwidth limit of IB, which is 100 Gbps. This is because Redis is CPU bounded. With single thread and event handling mechanism, the CPU resource limits Redis from achieving better throughput. To demonstrate this, we deploy more than one server on one node, each of them is connected by four clients. And at this time, the total throughput can increase linearly until it reached the limit of IB bandwidth. Fig 4 shows the trend of total throughput when the number of server instances on one node grows. The Red line indicates the maximum throughput that InfiniBand benchmark achieves.

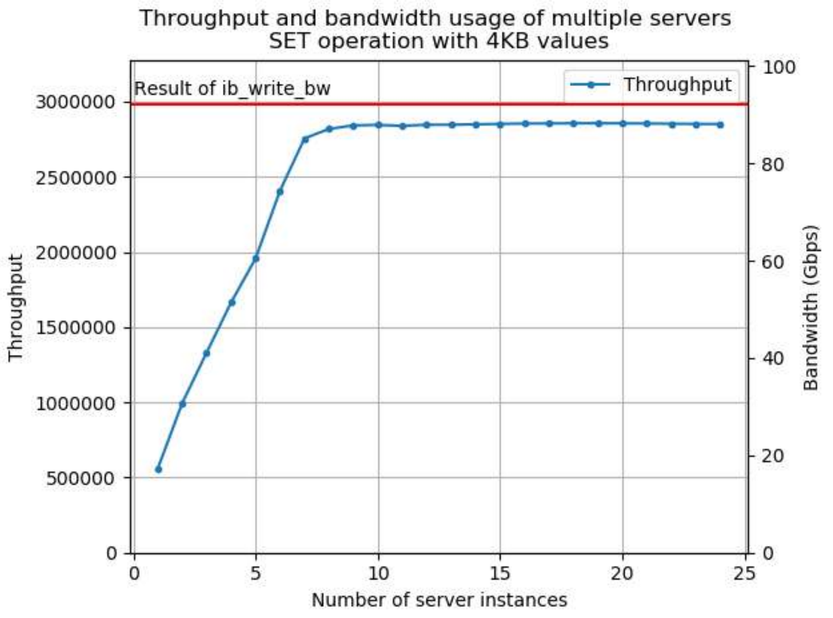

Fig. 4: Throughput and bandwidth usage of multiple servers SET operation with 4KB values

5.5. Cluster mode

In redis cluster, a response will be sent to the client before the master gets the response from its slave. As a result, Redis cluster shows no difference between with replica and without. We make the master send the response back to the client after receiving from slave. Thus, we can figure out how RDMA between master and slave optimize Redis.

Fig 5 plots the latency distribution of Redis cluster. Obviously RDMA between master and slave can make significant improvement. Redis cluster with RDMA between client and server has an average latency of 240.92us while Redis cluster with RDMA has an average latency of 45.23us. The average latency decreased by 81.23%.

On the other hand, Redis cluster on 3 masters has a throughput of 1449920, increased by 28.26% comparing to standalone Redis and increased by 745.26% comparing to Redis cluster with socket on 3 masters. Thus, RDMA can improve the performance of Redis cluster.

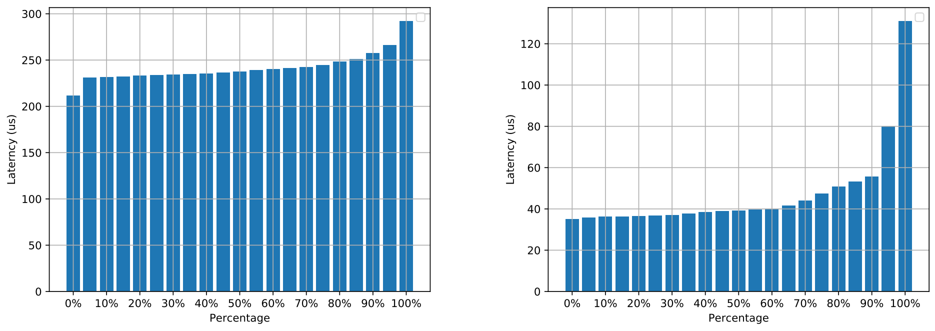

Fig. 5: Latency distribution of Redis cluster with socket and Redis cluster with RDMA

6. SUMMARY

As an in-memory database, high-speed network such as IB and RMDA operations are crucial to Redis. We substitute socket communication with RDMA both between servers and clients and among clusters. And we propose several significant optimizations for RDMA operations and Redis execution. In experiments, Our RDMA-based Redis outperforms original Redis by more than one order of magnitude.[]

REFERENCES

Proceedings of MobiSys 2003, the first international conference on mobile systems, applications, and services: May 5 - 8, 2003, san francisco, CA, USA. OCLC: 249708838.

Jian Huang, Xiangyong Ouyang, Jithin Jose, Md. Wasi-ur Rahman, Hao Wang, Miao Luo, Hari Subramoni, Chet Murthy, and Dhabaleswar K. Panda. High-performance design of HBase with RDMA over InfiniBand. pages 774–785. IEEE.

Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance RDMA systems. page 15.

Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. page 22.

Wenhui Tang, Yutong Lu, Nong Xiao, Fang Liu, and Zhiguang Chen. Accelerating redis with RDMA over InfiniBand. In Ying Tan, Hideyuki Takagi, and Yuhui Shi, editors, *Data Mining and Big Data*, volume 10387, pages 472–483. Springer International Publishing.