KVStore: A RDMA-based Distributed Shared Memory Key-Value Store

Shahid Khaliq
University of Waterloo
s3khaliq@uwaterloo.ca

Shuchita Singh
University of Waterloo
shuchita.singh@uwaterloo.ca

Siddhartha Sahu
University of Waterloo
s3sahu@uwaterloo.ca

ABSTRACT

Key-value stores are a very popular type of data storage systems used for a variety of workloads that make use of the simple interface mapping keys to arbitrary values. Generally such key-value stores operate completely from RAM to provide the maximum performance. However, such key-value stores become limited by the maximum RAM available on individual systems. Among the many different techniques to overcome this limitation and use multiple machines to service the key-value store, in this paper we implemented KVStore, a distributed shared memory architecture based key-value store that uses transparently mapped memory present across different servers to a single global address space. For the part of the address space on remote servers, we use Remote Direct Memory Access (RDMA) to directly perform reads and writes on the remote memory. RDMA atomic operations are used to ensure consistent read and writes to the global address space of the key-value store.

1 INTRODUCTION

Key-value stores are a simple but powerful type of NoSQL databases that allows arbitrary values to be stored in a database associated with specified keys. This simple data-model allows use in a wide range of use-cases, usually in the form of a non-persistent cache for data that needs speedy storage and retrieval, such as client sessions, shopping cart entries, various forms of data queues, among others.

For the best performance, key-value stores like Redis [15], memcached [14] and Aerospike [13] operate completely from memory. By eliminating the need to access data from disk, such key-value stores are able to provide orders of magnitude faster read and write performance for applications.

Application workloads in the modern cloud environment using such key-value stores often exceed the maximum RAM or CPU capabilities of a single server, thus requiring a distributed cluster of servers that together can service the entire workload. This requires sharding the data across the available memory on different servers.

One way in which such a distributed system is implemented is to use either server-side, such as Redis Cluster, or client-side, such as the API used in memcached, sharding mechanisms that uses techniques such as consistent hashing to distribute the data across the different machines. However, this approach has two key components that can be further improved to obtain an even higher performance capabilities.

First, the sharding technique is fully aware of the different server components of the distributed system and the fact that the entire memory space is divided into multiple discrete units spread across servers. The design of such systems involve explicit memory management of the discrete memory address spaces and deciding how to distribute the keys. This design can be simplified if the key-value store does not need to worry about the underlying distributed memory and is granted a unified address space that spans the total memory available in the distributed system. This idea uses a distributed shared memory (DSM) architecture where physically separated memories can be addressed as one logically shared address space. The key-value store uses the global address space directly to store the keys and the underlying system transparently maps the data to the different systems.

Second, the communication used in the above technique is performed over the traditional network using TCP and involves processes running on the individual servers that spend CPU resources in servicing the communication requests between the machines. However, upcoming networking technologies, such as Remote Direct Memory Access (RDMA), provide a faster mechanism for accessing in-memory data on remote servers. Key-value stores designed to use the faster networking stack would be able to take advantage of the enhanced speed in communication between servers for access data stored remotely.

In this paper, we present KVStore, which is a key-value store that uses a DSM architecture to transparently address memory that is distributed across different machines. For fast reads and writes to the memory space that is mapped to remote servers, KVStore uses Remote Direct Memory Access (RDMA) that bypasses communication with the remote CPU and directly operates on the data present in RAM. The in-memory data structures used in KVStore are designed to take advantage of the global memory address space, also taking into account key-value sizes and local accesses.

2 BACKGROUND AND RELATED WORK

In this section, we give a brief description of DSMs and RDMA technologies and describe other systems that have similar system designs.

DSM

In a distributed shared memory architecture system, there exists a shared memory address space that represents the total memory available in a distributed system consisting of multiple machines. The shared memory address space provides a transparent and unified mechanism to address memory present on the different machines. Apart from providing a large address space, such systems hide the communication mechanisms from the application code, provide mechanisms to protect simultaneous access to shared data, and scale transparently with a large number of machines.

Multiple systems [1, 2, 8, 17] implement a DSM architecture that provide a shared memory access mechanism. However, most such DSM are part of a larger distributed system such as a distributed operating system. KVStore uses the same idea to provide a global address space that stores the key-value data.

RDMA

Remote Direct Memory Access (RDMA) [12] provides an alternative networking stack as compared to the traditional protocols such as TCP or UDP. RDMA completely bypasses the kernel on the remote machines and allows direct data transfers between local and remote memory without having to communicate with the CPU or copy data around. This ensures that RDMA has a lower overhead than the traditional protocols for read and write operations on remote memory data.

RDMA is either available as Infiniband [10] hardware deployments or over traditional Ethernet networks using the iWARP [11] or RDMA over Converged Ethernet (RoCE) [3] protocols. While RDMA networks have not become a common feature of server deployments, RDMA-enabled servers are becoming increasingly commonplace in datacenter settings, for instance in Microsoft Azure public cloud offering.

Communication over RDMA is performed using the verbs interface, consisting of one-sided or two-sided verbs. Two-sided verbs needs CPU interaction on both the local and remote servers. One-sided verbs on the other hand allow direct access to pre-allocated memory regions of a remote server, without interacting with the remote CPU. The three common operations used in our system are READ, WRITE, and COMPARE_AND_SWAP (CAS). READ and WRITE are used to get and set the contents of the remote memory, respectively. CAS is an atomic operation that is used to compare and set 64 bits of data.

We use an asynchronous programming model in which RDMA operations are posted to a send and receive queuing pair (QP). RDMA work requests allow associating a distinct 64-bit identifier with each request, which we use to associate the responses with the original request object, thus implementing a call-back system instead of using polling.

RDMA Key-Value Stores

Multiple key-value system use a RDMA-based system design to store and access key-value pairs across a distributed cluster of machines. We describe the key ones below.

Pilaf [7] used RDMA to implement a cuckoo hash table [9] and uses one-sided remote gets and sets to retrieve and set data. Pilaf protects its shared data from simultaneous access by using checksums in its hash table data.

FaRM [4] implements a RDMA based key-value store and uses hopscotch hashing [5] to distribute the keys. FaRM uses CAS operations to protect simultaneous accesses to the data. KVStore also uses CAS operations to prevent simultaneous writes to shared memory locations.

HERD [6] is another system that implements a RDMA-based key-value store and uses set-associative indexing. HERD has an interesting design where clients post requests to the server using RDMA and the server services those requests using polling. The server-side request handling also implicitly provides protection from simultaneous accesses.

Finally, Nessie [16] is another unique RDMA-based key-value store which does not implement a server for servicing requests. Instead, the clients themselves use RDMA requests to perform GET and PUT operations on the key-value store. KVStore uses some key ideas of the in-memory data structure of Nessie for its own index.

3 ARCHITECTURE

Our RDMA KV store typically runs over multiple servers. In the current design, each server has an RDMA process running on it. This process is responsible for allocating memory for the key value store and making this memory available to the Network Interface Card (NIC). One or more key value store processes are also running on these servers. This decoupling of the RDMA process and the key value store process allows us to have servers with no key value processes running on them, if such a setup is required. Each key value store process can directly access the memory on the server it is running on and can send network requests to access memory residing on other servers. The NICs on each server bypass the operating system and directly access the requested local memory for the remote key value store process.

Memory Distribution

A naive implementation of a key value store would have fixed sized data blocks. A key would be hashed to an index corresponding to one of the data blocks and the data would be stored there. It would be a simple exercise to implement such a store and it would perform very well, but it would

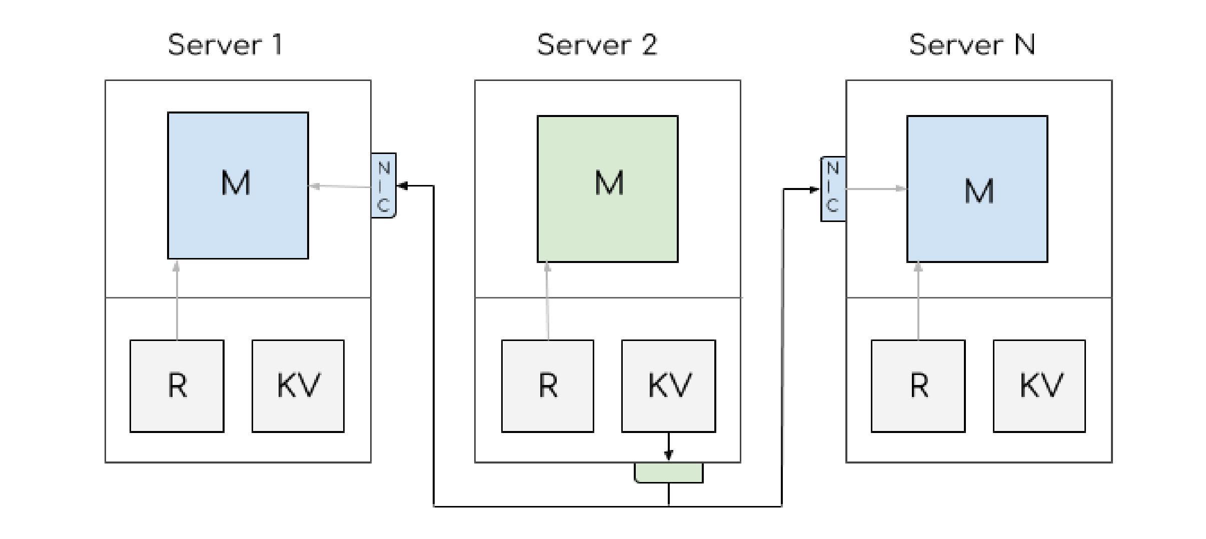

Figure 1: KVStore running on a cluster

result in a lot of wasted space. Our implementation uses data blocks of varying sizes. This results in a need to store metadata in addition to the actual data in the memory. The key value store process maintains a logical distribution of the allocated memory space by dividing it up into three portions, as shown in Figure 3:

- Allocation bits (0.2% of the allocated space)
- Index table (12.4% of the allocated space)
- Data table (87% of the allocated space)

The allocation bits portion allows us to keep track of used and unused data blocks on the respective server. One bit is used for each data block, which is set to 1 for used and 0 for unused. The atomic compare and swap RDMA operator is used when flipping bits to counter against concurrency issues.

The index table portion holds an 8 byte row for each data block on the server. This 8 byte row contains general information about the data block and is divided into four portions, as shown in Figure 2:

- Computer-ID (8 bits): The ID of the computer holding the data block.
- Computer-Offset (32 bits): The offset on the data block on the computer.
- Size (16 bits): The size of the data block.
- Owner (8 bits): The current owner of the data block (use for exclusive write access)

This distribution of bits served us well in our experiments but we discuss how it can be improved later in the paper. The atomic compare and swap RDMA operator is used here too, when updating an index row.

The data table portion holds the actual data and is divided into equal sized sub portions. Each sub portion contains data blocks of the same size. Different sub portions hold blocks of different sizes. So, for example, 12.5% memory in the data table will consist of data blocks of size 16 bytes and 12.5% will consist of data blocks of size 2048 bytes. Such a distribution ensures that more data blocks of smaller size exist in the system and since we expect to see smaller values (close to 16, 32, 64 or 128 bytes) more often than bigger values (close to 256, 512, 1024 or 2048 bytes), a nice memory distribution is formed for the stored data. Such a design limits the amount of wasted memory.

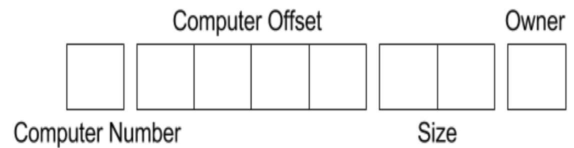

Figure 2: Index Row Bytes

4 IMPLEMENTATION

In its current form, the key value store is implemented as a C++ library. The library contains a KVStore class which exposes some common key value store functions like get, put and delete. RDMA functionality is abstracted out into a wrapper we wrote over the librdmacm library by mellanox.

Initialization

The key value store is initialized with some basic information about the number of servers in the cluster and the size of memory available for the store on each server. Using this information, it creates the previously discussed logical memory distribution. This allows different key value store processes

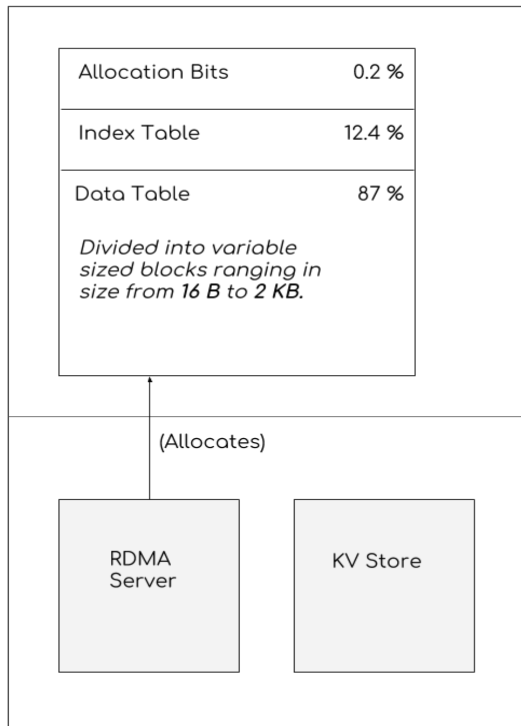

Figure 3: Memory distribution in KVStore

to store information in a consistent manner and hence retrieve it directly without any sort of communication between each other.

The sizes of the allocation bits and the index table portions depend on the size of the data table portion but the size of the data table is affected by the sizes of the other two portions, and hence we run into a chicken or the egg problem here. Experimenting with different sizes of these portions, we have observed that allocating about 87% of the memory for the data table hits the sweet spot and results in minimal space wastage between the metadata and the data portions of memory.

Given the size of allocated memory on each server, the key value store is able to estimate the number of data blocks for each supported data block size in the system. It is also able to calculate where the index table and the data table start on each server.

Put

The put request expects two strings: a key and a value. The key is hashed and then a modulo is taken against the total number of index rows (or data blocks) in the key value store. This value is converted into a Computer ID and Computer Offset pair. An RDMA compare and swap request is made to set the current server as the owner of the index row to ensure exclusive write access for the corresponding data block.

If the index row is storing previous metadata, the corresponding data node is retrieved and the key is inspected. If the stored key is identical to the key in the request, the system determines if the same allocated block should be used to store the new key and value. If the new value will not fit in the current block or a smaller block could be used to store it, the system flips the corresponding allocation bit to deallocate the current block and looks for a different block.

If the key stored is different than the key in the current request, the function calculates a new hash for the previously stored key using a second hash function. The system migrates the previously stored index row data to the new index. This is a recursive process since the index row at the migrate destination could also be occupied and has to be dealt with in a similar fashion. This recursive algorithm forms the basis of Cuckoo hashing, which we have implemented in our system.

When looking for a new block for storage, the system begins by searching its own allocation bits portion. If it doesn't find a suitable block, it retrieves and searches allocation bits portions of other servers. For every 8 data blocks, the system needs to make only one comparison check.

Once a suitable block is found, the system writes the key and the value to it. It then updates the index table row to store the size of the stored data and the location of the data block and sets the owner to null.

Get

The get request expects a single string: a key. The key is converted into a Computer ID and Computer Offset pair using the same method as for put. The index table row is retrieved and, using the data block location stored in it, the actual data block is retrieved.

The key stored in the data block is compared with the key in the request. If the keys are identical, the stored value is returned. If not, the system computes the second hash for the key in the request. Using the same process as before, it obtains the data block corresponding to the new hash and returns the value stored in it.

Delete

Given a key, the delete request works in a similar fashion as the get request to obtain the index row. It uses the atomic CAS operator to reset the index row and calculates the corresponding bit for the data block this index row was previously pointing to and flips it, thus effectively deallocating the data block and making it available for future use.

5 PERFORMANCE EVALUATION

We evaluated KVStore on the SYN cluster of University of Waterloo. We configured 2 to 5 machines to form a cluster. A single KVStore process then performed a set of PUT and GET requests using random strings in batches starting from

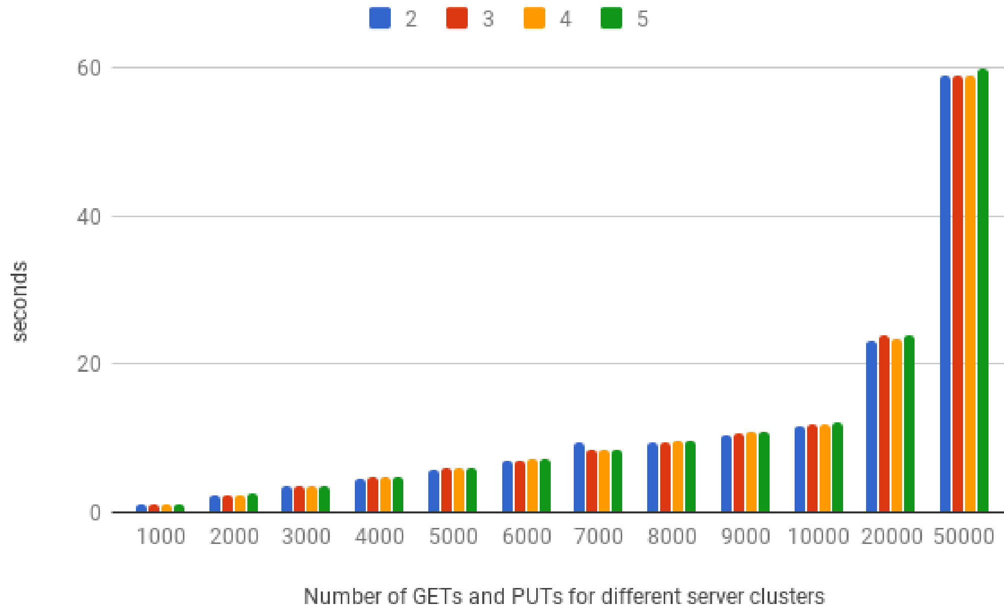

Figure 4: Execution time of KVStore on varying number of servers

1000 upto 50,000 requests. We noted the total time it took to complete executing the whole batch of requests.

Figure 4 shows the results. As shown in the figure, the execution time linearly increases based on the number of GET and PUT requests, and on the number of servers that are part of the cluster.

6 FUTURE WORK

Our key value store works fine in its current state but it leaves a lot more to be desired. A couple of areas which we could work on in the future might be the following.

Index Row Structure

The current index table row structure can be tweaked and experimented with to see what works best. A 32 bit computer offset limits the memory size in each system to 4 GBs. The 8 bit computer ID and owner fields allow the maximum number of machines to be 255. The 16 bit size field allows the maximum supportable key + value size to be 65535, which is an overkill, considering that the maximum allowed data block size in the current implementation is 2048 bytes. A couple of bits could perhaps be removed from the these three fields and shifted to the offset field to allow larger memory chunks.

Cuckoo Hashing

Our current implementation uses cuckoo hashing to counter hash collisions. This provides fast lookups but a problem with this is that, eventually, the hash table needs to be rebuilt. We have not completed our rehashing code at this stage and the system works without it. Using two hashing algorithms, as the system is currently using, we can expect to utilize up to half of the allocated memory space without having to rebuild the hash table. If we add a third hashing algorithm, we can expect to use up to 91% of the allocated space. It might also be fruitful to experiment with other hashing algorithms and see which one performs the best.

Allocation Bits

While looking for a free data block, the system has to sequentially search through data bits. We have optimized this so that the system can hop through 8 blocks at a time. It still does not seem to scale. If the memory size is large enough and the key value store is almost full with data, the system might have to sift through megabytes of allocation bits and make millions of comparisons before it is able to find a suitable data block. A better approach would probably be to have some communication between different processes and have each process maintain a list of free data blocks on its server.

7 CONCLUSION

In this paper we presented KVStore, an RDMA based distributed key value store. KVStore uses an all in-memory approach and supports variable block sizes to conserve space. We also implemeted the Cuckoo hashing algorithm in an

RDMA key value store and presented our initial performance evaluation and some ideas for future work.

REFERENCES

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[2] A. Barak, S. Guday, and R. G. Wheeler. *The Mosix Distributed Operating System: Load Balancing for Unix (Lecture Notes in Computer Science)*, volume 672. springer-verlag, 1993.

[3] M. Beck and M. Kagan. Performance evaluation of the rdma over ethernet (roce) standard in enterprise data centers infrastructure. In *Proceedings of the 3rd Workshop on Data Center-Converged and Virtual Ethernet Switching*, pages 9–15. International Teletraffic Congress, 2011.

[4] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 401–414, 2014.

[5] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *International Symposium on Distributed Computing*, pages 350–364. Springer, 2008.

[6] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 295–306. ACM, 2014.

[7] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[8] C. Morin, R. Lottiaux, G. Vallée, P. Gallard, G. Utard, R. Badrinath, and L. Rilling. Kerrighed: a single system image cluster operating system for high performance computing. *Euro-Par 2003 Parallel Processing*, pages 1291–1294, 2003.

[9] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[10] G. F. Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[11] M. J. Rashti and A. Afsahi. 10-gigabit iwarp ethernet: comparative performance analysis with infiniband and myrinet-10g. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[12] R. Recio, P. Culley, D. Garcia, J. Hilland, and B. Metzler. An rdma protocol specification. Technical report, IETF Internet-draft draft-ietf-rddp-rdmap-03. txt (work in progress), 2005.

[13] Aerospike. https://www.aerospike.com.

[14] memcached. http://memcached.org/.

[15] Redis. http://redis.io.

[16] T. Szepesi, B. Wong, B. Cassell, and T. Brecht. Designing a low-latency cuckoo hash table for write-intensive workloads using rdma. In *First International Workshop on Rack-scale Computing*, 2014.

[17] B. J. Walker. Open single system image (openssi) linux cluster project. *2005*, 2008.