Vrije Universiteit Amsterdam

Bachelor Thesis

---

# RDMA Key Value store: Scalability of RDMA transportation types

---

**Author:** Jan Erik (Jens) Troost (2656054)

| | |
|---|---|
| *1st supervisor:* | Dr. Animesh Trivedi |
| *daily supervisor:* | supervisor name |
| *2nd reader:* | supervisor name |

*A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science*

June 4, 2021

# Abstract

Seemingly ever growing tech giants, such as Facebook, Amazon and Google, require fast, reliable and scalable key-value storage (KV-store) to serve product recommendations, user preferences, and advertisements. Past advancements of KV-stores have focussed on improving these underlying data structures. Remote direct memory access (RDMA) networks have increasingly become more popular in commercial and academic data centers. This relatively new technology offer lower latency and higher throughput performance compared to tradition sockets and network interface cards (NIC). Scalability performance of RDMA transportation protocols is less known, previous work focused on RDMA verb choice.

This paper focuses on this gap, evaluating scalability performance of RDMA transportation types. These findings will be used to give recommendation for RDMA KV-store implementations. Macro-level benchmarks have been conducted, with realistic KV-store workloads. It has been found that RDMA offers a significant improvements compared to TCP. UD has shown to perform best, with 94.8% throughput improvement over TCP, and 58.7% improvement over RC. All the while offering consistent and gradually increasing latency up to 30 clients, reaching a maximum average latency of 0.06ms. Additionally, scalability issues of RC have been shown, and is not recommended in a KV-store application.

# Contents

# List of Figures

## LIST OF FIGURES

# List of Tables

## LIST OF TABLES

# 1

# Introduction

## 1.1 Context

Seemingly ever growing tech giants, such as Facebook, Amazon and Google, require fast, reliable and scalable key-value storage (KV-store) to serve product recommendations, user preferences, and advertisements[10, 14]. A KV-store is a data storage type which aims to store many small values and provides a high throughput and low latency interface. KV-stores usually offer a simple interface using 3 commands: GET, SET, and DEL. A trivial implementation makes use of a hash-table which matches the hash value of a given key, to the value associated with that key. Past advancements of KV-stores have focussed on improving these underlying data structures[12, 23]. For example, Cuckoo and Hopscotch hashing[11, 20], have presented an alternative of an improved hash table algorithm. Nonetheless, with recent developments with RDMA networks, this has become a more attractive direction to improve KV-stores.

Remote direct memory access (RDMA) networks have increasingly become more popular in commercial and academic data centers, due to the increase in availability and decrease in cost for RDMA capable network interface cards (RNICs). RDMA offers a more direct connection between two machines, by using direct memory acces (DMA), without the involvement of host CPU or OS.

## 1.2 Problem statement

Making efficient use of RDMA networks is a difficult task, and requires in-depth knowledge on the hardware constraints present in the RNICs [**?** **?** ]. Additionally, in a higher level sense, there are design choices to be made. RDMA networks can handle of various

transportation types and so-called verbs, each with their own advantage and disadvantage. The impact of transportation types on scalability of an RDMA bases KV-store has not been explicitly examined. Previous work has focused on researching verb choices, less on transportation type[20, 22, 24]. Scalability is an important factor for growing tech giants. These require a scalable, high throughput and low latency solution[10], which RDMA could provide.

## 1.3 Research Question

This paper will explore to what extent RDMA transportation types affect the performance and scalability of KV-store. For this, this research will answer the following questions:

**RQ1** How scalable are these transportation types?This being an important factor for tech giants which make use of RDMA KV-store, and require a system to be scalable and reach multiple clients.

**RQ2** What are the advantages and disadvantages of RDMA transportation types on an KV-store? This question aims to apply the already known information on the RDMA transportation types, on a RDMA KV-store specific setting. Aiding the design processes of further RDMA KV-store implementation, and possibly related implementations.

## 1.4 Research Method

In this thesis the network performance of KV-store, with varying protocols, will be studied. First an understanding of KV-stores is established. Along with discussing known issues with traditional networking, a relatively new technology, RDMA, will be introduced. Network performance, in the form of throughput and latency will be measured and used to evaluate RDMA transport types.

A prototype[15, 18, 27] and experimental[17, 19, 25] approach is taken for this thesis:

**M1** A KV-store prototype will be implemented, with a flexible network interfacing, to include both traditional socket and RDMA interfacing. This, and all other relevant project files, are open source and can be found on Github[7].

**M2** To investigate the performance and scalability between the various networking types, a workload realistic[8] macro-level benchmark will be designed.

With the focus of this thesis being on the network implementation, a trivial KV-store will be used. This implementation does not offer strong performance and scalability, however these issues are minimized and kept consistent throughout experiments.

All measurements hereafter are recorded on the DAS-5 computing cluster. This allows for a consistent working environment. Further details on which are shown in table 3.1 and discussed in section 3.3.1

## 1.5 Thesis Contributions

This thesis presents scalability performance of RC and UD are shown and compared to the well established TCP protocol. Additionally, recommendation as to which transportation protocol is best suited for an RDMA KV-store application. These findings can be used to further examine possible optimizations or RDMA verb choices, or aid in future RDMA KV-store implementations.

## 1.6 Plagiarism Declaration

I herby declare that this thesis work, all data collected and findings are my own.

Note: permission incoming for use of KV store structure from OS course.

## 1.7 Thesis Structure

Section 2 will provide the necessarily background knowledge of KV-store, linux sockets, and RDMA. Next, section 3 will go over the design of the KV-store, networking interfacing, and benchmark. In section 3 this design will be taken and described the implementation in a more technical prospective. The benchmarking result will be presented and analyzed in section 4. Section 5 compares findings with that of previously done work. Future prospects will be given in section 6. Closing off the thesis, section 7 will go over the conclusion and provide recommendations.

# 2

# Background

In this section, background information on key value stores, which is the backbone of this thesis, will be given. Also, the issues with commonly used linux sockets, will be explained. Further, RDMA is thoroughly explained, as this is crucial to the understanding of this paper, and how it addresses issues facing sockets.

## 2.1 Key Value Store

Key value stores are extensively used to offer low latency look ups. These have a wide variety of use cases, most notably as cache systems, such as Memcached[1], and as remote DRAM storage, such as RAMCloud[26]. In its simplest form, KV store use a set of commands, commonly SET, GET, and DEL, to perform tasks on a server. These commands interact with a fast data structure, like hash table or similar. Typically, this was the target for advancements in KV stores, using lower latency, memory efficient, and scalable data structures[12, 23].

It has been found that typical work loads consist mostly of GET requests. Atikoglu et. al. analysed workloads on Memcached systems, and have found that on average a 30:1 (95%) GET/SET ratio[8]. This figure will be used in the benchmarking design, see section 3.3.

## 2.2 Linux sockets and TCP network stack

Linux sockets are versatile programming API, offering for many types: files, IO, and lastly networking. Behind this socket interface, the kernel is tasked with data and memory, and connection management[16, 29]. This results in CPU cycles being used to process

incoming packets and data copies, while these cycles could be used for other tasks. Linux takes an extensive route when dealing with packets, as shown by Hanford et. al. detailed investigation[16]. Memory copies have been found to cause delay in Frey and Alonso's work[13], however for small message sizes, Yasukata, and team, found this is insignificant[31]. TCP processing accounted for more than half of the overhead, after subtracting unavoidable PCIe latencies.

Past attempts at improving TCP performance included offloading most processing to the network interface card (NIC)[16]. Higher end NICs use an TCP offload engine (TOE) and DMA. The socket API still requires system calls, among other limitations, thus requiring kernel trapping for management.

## 2.3 RDMA

Remote Direct Memory Access (RDMA) address the issues of linux sockets, providing lower latency, less CPU overhead, and potentially higher throughput. RDMA has been used in super computers for many years, however recently have seen significant improvements. RDMA capable network cards (RNICs) have seen lowering in cost [21], which made this an appealing improvement to data center networking.

RDMA achieves this low latency by offloading the network processing onto the RNIC, bypassing CPU and kernel entirely. As shown in section 2.2 above, the linux kernel has an extensive route, from application to NIC to network, including system calls and memory copies. With RDMA, a zero-copy memory access can be realised through DMA and programmable IO (PIO) operations. This makes RDMA a compelling technology for latency sensitive workloads, like KV-store, making remote memory operations possible and perform near to local memory operation speeds.

### 2.3.1 Queue Pairs

RDMA is largely based around the notion of a queue pair (QP). These consist of a send and receive queue, these are the essence of performing network operations. These queues can be seen as the receive and transmit queue (RX and TX respectively) in classical NIC, however are bound per QP instead of being shared system wide. A QP is similar to a linux socket, as these are used to send and receive data. A work request (WR) is a type of command that tells the RNIC what task to perform and which memory locations are needed, these are passed to the RNIC via PIO operations. For RDMA operations like READ and WRITE this would include the remote address to be accessed, with two-sided

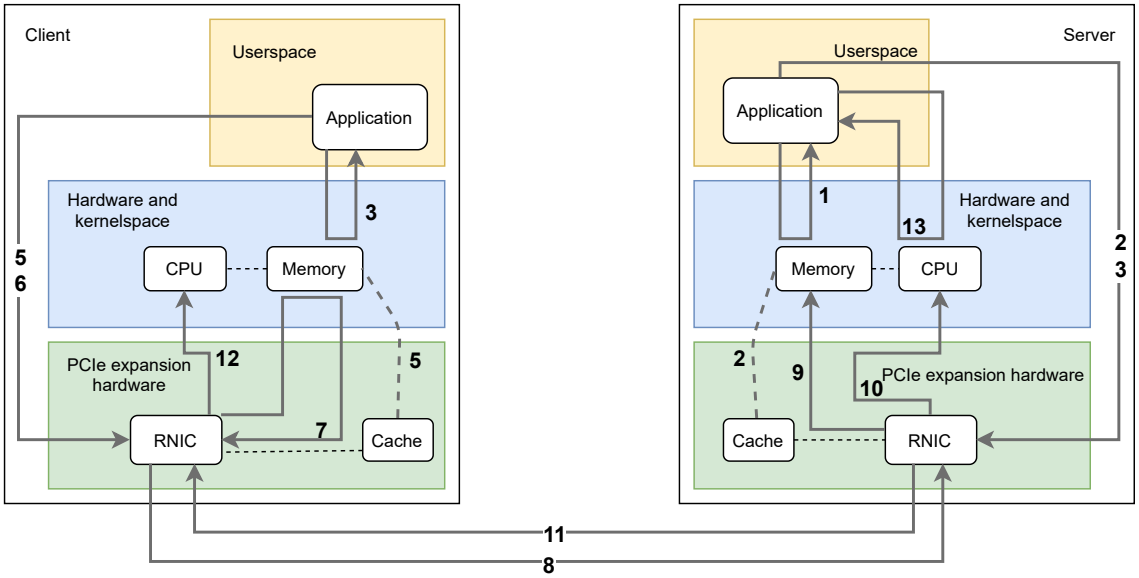


**Figure 2.1:** Process overview of pre-posting and send from client to server. add more explaination

verbs such as SEND and RECV this is accompanied by a buffer for which DMA operations can take place. In this paper, the focus will be on SEND and RECV verbs, more can be read in section 2.3.3 below and in 3.2. Every WR placed either in the send or receive queue will be consumed in the same order as placed in the queue, similarly for completions of WR's. This is important when dealing with unreliable transportation, as will be explained in 2.3.2.

Queue pairs are also linked with a completion queue (CQ). This queue is used as a notification queue, with the status of WR's. For signalled WR's, a completion event will be passed from RNIC to CPU, thus minor CPU involvement.

### 2.3.2 Transportation types

Much like traditional networking, RDMA can make use of several transport protocols. Simply put, these can be split into two types, connection and datagram, and reliable and unreliable. There are 3 main transportation types: reliable connection (RC), unreliable connection (UC), and unreliable datagram (UD).

Firstly, unreliable protocols do not make use of ACK/NAK packets, while this is used for reliable, this could result in packet loss and unordered packets. However, it has been shown that this rarely occurs[20, 22]. Additionally, this could require application retransmission handling. With reliable protocol this is process is done by the RNIC, without OS or

application involvement.

As stated in the name, RC and UC need a connection between queue pairs (QP). Only one QP can be connected to another QP. Contrasting this, unreliable datagram (UD) do not require a connection. This meaning that a UD QP can communicate with any other UD QP. UD therefore can make efficient use of a one-to-many network topology or application, and RNIC cache. Every QP has to be held by RNIC in cache, which is severely limited[28]. A cache miss or many context switching in RNIC, would diminish performance. With the fan-in and fan-out effect present in many data center workloads[30], the potential scalability of connection oriented QP's is limited[22].

UD is limited however in the maximum transmission unit (MTU). As can be seen in table 2.1, UD has a maximum message size of 4KB, beyond this and the message is divided into several packets. This strongly contrasts the 2 GB available for RC and UC. This has to be taken into account on a per-application bases. In the case of this thesis, 4 KB is sufficient message size for KV-stores. Frey and Alonso have comprised a list of application and if it is suitable for RDMA[13].

### 2.3.3 Verbs

To interact with RNICs, RDMA uses so-called "verbs" to execute specific types of instructions. Some of which are: read, write, send, and receive. Read and write follow so-called memory semantics, while send and receive follow channel semantics. Memory semantics require the destination memory address to be known. This meaning, to be able to perform a RDMA read of a remote memory location, the memory address of the requested memory needs to be known.

Channel semantics are simpler in the sense that the remote memory address does not need to be known. However, to perform a send operation from client to server, the server needs to first post a receive. This tells the server's RNIC which memory location the application expects the next incoming message to be place.

Not all verbs are available to every QP type. Table 2.1 summarizes the transportation type and which verb is available.

### 2.3.4 Connecting Queue Pairs

Establishing a connection between QP's involves exchanging metadata related to the QP. This can be done via a known QP, or via traditional communication like sockets. The latter is simplest, since it is a well established. However, in some cases, a classical ethernet

| | SEND | RECV | READ | WRITE | MTU size |
|---|---|---|---|---|---|
| *RC* | YES | YES | YES | YES | 2 GB |
| *UC* | YES | YES | NO | YES | 2 GB |
| *UD* | YES | YES | NO | NO | 4 KB |

**Table 2.1:** Verbs available to each transportation type

is not available. In these circumstances, a known QP, Communication Manager (CM), can be used. This is a QP which is always available when using RNIC, and can be used to communicate between RNICs.

### 2.3.5 Programming with RDMA

Unlike traditional sockets, much of RDMA's interfacing is open to the programmer, allowing for more detailed optimizations. The cost of this is the need for more memory management, which otherwise would be done by the kernel.

Each program should allocate a protection domain (PD) before any other RDMA operation, as this acts as the address space within an RNIC. Any queue or buffer is associated with a PD. A program can allocate many PDs, however cannot be used interchangeably. A buffer registered within one PD cannot be accesses by a queue pair from another PD, even within the same program.

To create a "connection", first a QP should be created. QP's differ per transportation type, see section 2.3.2, and should be created accordingly. Along with send and receive queues, a completion queue is needed to inform the application when needed. To check for new WC's, the program can either block or poll on this CQ. The latter is preferred for lower latency. Checking the CQ is a thread safe operation, and a WC can only be consumed once and only once. This is important for the UD implementation used in this thesis, as multiple threads can poll CQ without conflicting.

Before performing any DMA operations, memory should be registered to make this accessible by the RNIC. This is done within the PD, and creates a memory region (MR). This memory can then be used by the RNIC with DMA. Any given memory address can be registered multiple times. Memory registration is an intensive operation and should be used sparingly.

After the setup processes, the application can perform RDMA operations. A WR should be created. This holds scatter-gather elements (SGE), used to identify which memory to access and what size, and the type of operation being performed, be it RDMA READ,

WRITE, or SEND, RECV. A WR can be a linked list of multiple WR's. Along with this WR, a so-called "bad WR" should be included, for failed requests. A small payload can be attached to this WR, this is called inlining. This reduces the need for potential DMA operations, since some of the payload can be given along with the WR in one PIO operation. Program can post the WR and bad WR, on the send or receive queue of a QP. This operation is non-blocking, meaning the program can continue doing other instructions.

The completion queue can be polled, or blocked, to review the status of the WR's. Once a WC is enqueued the status of the WR can be seen, either successful or unsuccessful. A successful WC for a receiving WR would entail that a SEND has been received, and the requested buffer has been filled.

# 3

# Designing a RDMA Key Value store

In this section, the design of the used key value store is shown, along with design decision made for RDMA. Further, benchmarking strategy will be explained, used to evaluate the performance of various RDMA transport types. Lastly, an overview of DAS-5 is given, which will be used to run benchmarks. This can be used when comparisons are made with other findings.

## 3.1 Key value store

This thesis is focused on the scalability using RDMA, and will not focus on advancing KV-stores. Therefore, a simplified KV store has been used. For this KV store, SET and GET instructions are mainly used, a long with other commands for testing purposes.

### 3.1.1 Requests

For a client to interact with a KV server, the client has to send a request towards the server. A request is structured as shown in figure TODO MAKE FIGURE. Along with the command type, a key must be given. This key is used to identify the correct field within the hash table. A field for payload is always sent, however only used for SET commands. This is to have a constant data size.

### 3.1.2 Response

After processing a request the server will always respond. The structure of which is shown in TODO MAKE FIGURE. In case of a successful execution, an OK code will be given. For GET requests a payload is given along with this OK code, this is the value requested.

| Request | |
|---|---|
| **int** | client_id |
| **method** | method |
| **key[64]** | key |
| **Msg[256]** | message |
| **size_t** | key_len |
| **size_t** | msg_len |
| **int** | connection_closed |

(a) Request structure

| method |
|---|
| UNK |
| SET |
| GET |
| DEL |
| PING |
| DUMP |
| RST |
| EXIT |
| SETOPT |

(b) Values for method

**Figure 3.1:** Structure for request with accommodated method

| Response | |
|---|---|
| **response_code** | code |
| **msg[256]** | message |
| **size_t** | msg_len |

(a) Response structure

| method |
|---|
| UNK |
| SET |
| GET |
| DEL |
| PING |
| DUMP |
| RST |
| EXIT |
| SETOPT |

(b) Respond codes

**Figure 3.2:** Structure for response with accommodated response codes

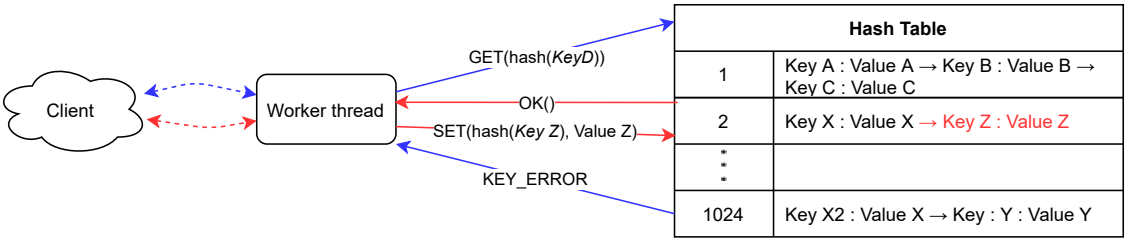


**Figure 3.3:** Key value server layout. Process worker thread takes with incoming requests, and what response is sent. A successful SET request is shown in red, adding a value "Value Z" to the internal linked list, with key "Key Z". Blue is an unsuccessful GET where the key is not known.

Upon error, an error code will be sent back: PARSING_ERROR, or UNKNOWN (UNK), latter of which is when the given command is unknown.

### 3.1.3 Hash table

Internally, a hash table with 1024 buckets is used. Each bucket contains a linked list of key-value pairs. Figure 3.3 shows the structure of the KV store used. It should be noted, a linked list approach, as used here, does not scale well. As buckets are filled, either due to a hash collisions or extensive use, GET requests can cause for significantly delay, especially when encountering a key miss. This can be partially solved with a large hash table, this however will not be a long term solution, along with strong hash function. However, with a 95% GET request workload, the scalability issues will not show a significant performance loss. More over, in section 3.3 this issue will be kept constant throughout experiments.

### 3.1.4 Multithreaded

The KV server implemented for this thesis is also multithreaded. For every client a worker thread process will be created. This thread will read requests, process accordingly, and send back a response, all the while the main thread is used to accept and set up for new clients. In this way, worker threads are unaffected by new clients that are connecting.

To ensure concurrent and correct operation, each bucket has a mutex lock, and each key-value pair a read-write lock. This could cause performance loss as been seen in FaSST [22, 28].

## 3.2 RDMA

For this research, the two-sided verbs SEND and RECV are used for RDMA. Looking at table 2.1, across all transportation types SEND/RECV is the only available verb. This might come at a cost of performance, due to the channel schematic and CPU involvement. It has been shown that in some use cases, one-sided WRITE operations perform better[20]. However, this difference is minimal, and the use of WRITE is not possible for this research.

### 3.2.1 Queue pairs

Connection based QP's will be handled similarly as in section 3.1.4 above. Every worker thread will have a client connected QP, which can only communicate with this client.

This differs for UD. As stated in section 2.3, any UD QP can communicate with any other UD QP. This meaning, that the server only needs one QP for all clients. A worker thread is still created for every client, however, in the case of UD, all threads use a shared queue.

TODO ADD MORE GRAPHICS

## 3.3 Benchmark design

To evaluate the performance and scalability of the RDMA KV store, a multiclient benchmark has been made. In total, 10 million tasks will be divided among clients. This number is constant to lessen the scalability issues with the underlying hash table. A task involves two operations, sending a request and receiving response. Only once a response is received from the server, can the client continue to the next task. This meaning that the full potential of RDMA is not being used, however it has been chosen for correctness, and such that latency can be evaluated accurately. TODO MAYBE I SHOULD CONTINUE?

During client benchmark execution, the time will be taken at two points: before sending request, and after receiving response. With this, the throughput and latency of every client and every operation can be traced back. These times will be kept in an array, and be returned after completing its tasks. The time will be later written to CSV files perform statistical analysis and graphs are drawn, as can be seen later on in the evaluation section 4. For this, Python 3.6[5] is used, along with pandas[4] and matplotlib[3].

The benchmark is designed to evaluate each transportation types under similar conditions, and will follow the same path. Once a client is setup and connected with the server, it will start performing its tasks. These tasks follow the 30:1 GET/SET ratio that found

| Cluster | Nodes | CPU type | Frequency (GHz) | Memory (GB) | Network |
|---|---|---|---|---|---|
| VU | 68 | dual 8-core | 2.4 | 64 | IB and GbE |
| LU | 24 | dual 8-core | 2.4 | 64 | IB and GbE |
| UvA | 18 | dual 8-core | 2.4 | 64 | IB and GbE |
| TUD | 48 | dual 8-core | 2.4 | 64 | IB and GbE |
| UvA-MN | 31 | dual 8-core | 2.4 | 64 | IB and GbE |
| ASTRON | 9 | dual 8/10/14-core | 2.6 | 128/512 | IB, 40 GbE, GbE |

**Table 3.1:** DAS-5 cluster specifications

by Atikoglu et. al.[8]. A SET request has a 5% chance of being generated. This is done by generating a random number as follows: $rand()$ mod $100 <= 5$. Else a GET request is generated. The client sends out the request, and waits for response.

### 3.3.1 Experimental Setup

All performance tests and results have been gathered on the DAS-5 computing cluster[2]. This distributed system of computers spread across the Netherlands, and is used by research groups from VU Amsterdam, TU Delft, Leiden University, and many more. Each cluster varies slightly in specifications, however each, is equipped with dual Intel E5-2630v3 8-core CPUs, a 56 Gbit/s Inifiniband (IB) RDMA networking, and 1 Gbit/s classical ethernet networking. The specifications of each cluster is as shown in table 3.1.

All experiments make use of the IB network card, and is also configured to run TCP/IP. Furthermore, at least two nodes are used, his ensures that the server and clients are separated.

# 4

# Evaluation

In this section, the experimental results are presented. The RDMA transportation types are compared within and against the baseline TCP implementation. Throughput and latency are mainly used to analysis the scalability of these transportation types. The same raw data is used for both measurements, thus can be put alongside each other.

In short, the results show:

- Throughput reaches an equilibrium with all transportation types up to 30 clients. UD performs best, with a maximum throughput of roughly 400 kilo-tasks/sec. Further details are given in section 4.1.
- Latency increases across all transportation types, with increased number of clients. Results are inversely comparable to throughput, again with UD performing best, with a latency of roughly 0.058 ms at 30 clients TODO ACCURATE. Section 4.2 delves deeper.

To recall the benchmarking setup: Ten million tasks are divided equally between $n$-clients. A task is defined as a request and response, and are equal to two operations.

## 4.1 Throughput analysis

We compare the average throughput between TCP, RC, and UD with up to 30 clients. From figure 4.1, the maximum average throughput is observed to be roughly 205 kilo-tasks/sec for TCP, 251 kilo-tasks/sec for RC, both at 30 clients, and 399 kilo-tasks/sec for UD at 25 clients. RC shows a mere 22.8% maximum improvement over TCP. UD in turn offers a significant gain of 94.8% over TCP.

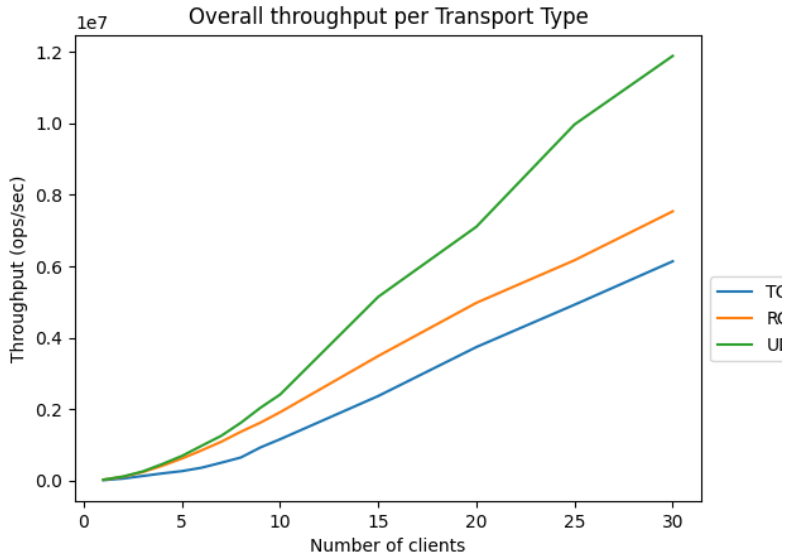


**Figure 4.1:** Throughput of clients executing 10 million operations

### 4.1.1 Scalability rate

It can be observed from figure 4.1 that all transportation types scale up to roughly 20 clients before settling. UD outperforms the other types, in both throughput and initial rate of scalability. A near linear scalability rate can be seen in the first 15 clients for UD. This is due the addition of threads for every new client. Since all these threads are actively working for all clients, this linear increase is to be expected. For the first 10 clients, UD increases its throughput by 23.4 kilo-tasks/sec per additional client. Compare this with 18.1 for RC and 10.9 kilo-tasks/sec for the baseline TCP. This shows strong scalability for UD with small number of clients. It is unclear from this graph if UD continues to scale beyond 30 clients.

### 4.1.2 Similarities between RC and TCP

It can also be seen in figure 4.1 that RC is similar to TCP in scalability. With both being a connection oriented protocol, each server thread being tied to a client, it shows similar performance. The benefits of RDMA do reflect in these results, with the throughput of RC being on average 59.7 kilo-tasks/sec above TCP. Both protocol stabilizing after roughly 20 clients, with less than 5% change for TCP and less than 2% for RC.

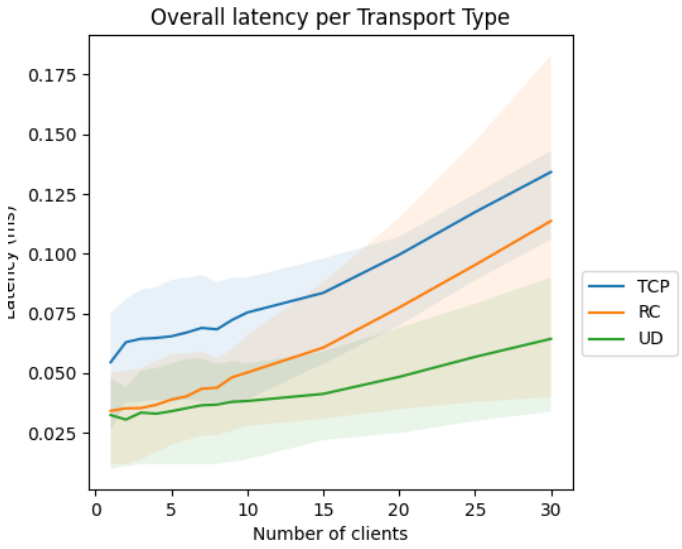


**Figure 4.2:** Latency of clients executing 10 million operations

### 4.1.3 Fairness between clients

TODO GENERATE GRAPH FOR THIS

## 4.2 Latency analysis

An important factor to KV store is their quick response time, and therefore this should ideally scale similarly, preferably better, than throughput. RDMA offers fast response time due to the lack of CPU and kernel involvement. To see how this is translated per transportation type, the average latency of all 10 million tasks were measured, from request sent to response received. Results can be seen in figure 4.2. Along with the average latency, the first and third quartile are shown, to analyze the variation of latencies.

The latency between protocols is an inverse reflection of the throughput graph, scaling and overall performance. Similarly, TCP performed worst, with an average latency of 0.135ms TODO ACCURATE NUMBER at 30 clients. RC following the same path as TCP and hits an average of 0.11ms, also at 30 clients. UD again performing best, reaching 0.06ms on average. This is an order of magnitude more than has been found in FaSST[22], HERD[20], and what should be theoretically possible. This could be due to lock contention with shared QP, and poor performance of KV store. A further examination of this could support this.

Similarly to throughput, UD scales relatively well, while RC and TCP scale linearly. A near uniform latency for the first 15 clients, afterwards a gradual slope, compared to TCP and RC.

### 4.2.1 Variation from average

In figure 4.2 the variation is given in form of first and third quartile. It can be seen that TCP and UD have a constant, a small, variation. TCP is skewed towards higher latencies at 30 clients, while UD, and RC, stay normally distributed throughout. RC has a rapidly increasing variation. This is concerning for scalability and consistency, and could possibly be due to context switching.

### 4.2.2 All 30 clients and per operation latency

Figure TODO ADD REFERENCE TO GRAPH OF ALL 3 PER CLIENT LATENCY delves deeper into the details of the variation in latency. Within all protocols there are outliers and spikes in latency. Surprisingly, TCP has the least varying results, despite the few initial spikes.

### 4.2.3 RDMA multiple QP's causing cache misses and context switching

In section 2.3.2 the issue with holding multiple QP's within the RNIC, has been discussed. The RNIC needs to perform context switching when changing QP. As the number of QP's increase, like with connection oriented transport types, the number of context switches needed increases. Along this, the chance of cache misses are also increased. This decreases the performance, with increased latency, and in throughput as shown here. With context switching, a large delay occurs, and harms the performance per client, but also of the overall network. For this reason, for large number of clients, a one-to-one QP is not advisable.

#### 4.2.3.1 Cache misses as increased variation for RC

Furthermore, cache misses occur more often at the starting phase of KV store operations, as can be seen in section TODO PER CLIENT LATENCY above. RC has consistently higher latency (0.75ms) for some clients at the start. Initially it was thought to be retransmissions, however for this was set to 67.1ms. Since this only occurs to some clients, and not all, this could be a result from the aforementioned context switching and RNIC cache misses. At the start of operations, worker threads are operating in sync. This would result in RNIC processing multiple WR's concurrently. Some QP's will be cached and experience little

cache misses throughout the starting phase, while others would encounter cache misses repeatedly. Once threads are out of sync and KV store operation latency increases (due to the increasing size of the hash table), the cache misses will occur for all QP's. This can be seen in figure TODO, as the latency in the final phase is randomized and less constant, compared to the starting phase.

With cache misses, latency per client varies, as some QP's would require further lookup after cache miss. This effect can be seen in figure 4.2, as increasing number of clients, and therefore QP's, increases the variation dramatically. Further investigation into cache misses would support these claims, this has not been done for this thesis.

# 5

# Related Work

There are several proposed RDMA KV store designs. One of which, HERD [20], uses a combination of transport types. HERD uses RDMA WRITE over UC for requests and SEND over UD for responses. Kalia et. al. have shown that WRITE offers lower latency and higher throughput compared to READ. Outgoing WRITEs have also been shown to not scale well with increase in number of QP connections. This confirms the poor scalability seen with RC in section 4.1.1.

HERD out performs the RDMA KV store presented in this thesis significantly. This could due to multiple factors:

- HERD has implemented some optimizations towards prefetching before posting a SEND WR. With this they have observed roughly 30% improvement, in the best case comparison (2 random memory accesses at 4 CPU cores.)
- Making use of one-sided WRITE for requests, bypassing the CPU.
- Using inlined data for key. Inlining small payloads decreases latency as there is no need for a DMA operation.

However, Kalia et. al. stated that performance should be comparable to SEND/SEND, given that inlining is possible. It was found that for a large number of clients and/or requests, the round-robin polling used, is inefficient. With 1000s of clients, a SEND/SEND design would scale better than the WRITE/SEND used in HERD.

# 6

# Future improvements

## 6.1 Experimental transport types

Current transportation types all have advantages and disadvantages. Reliable transport has advantages being predictable packet behavior. However, as has been shown in this thesis, connection transports (such as RC) have scalability issues, due to the RNIC caching issues with increasing number of QP's. Dynamically Connected Transport (DCT) is, currently, an experimental transport type which addresses these issues. DCT has a connection based design, while only requiring one QP. This is achieved by dynamically connecting and disconnecting with remote QP's, which would introduce additional latency with short-lived clients, or when dealing with concurrent large number of clients.

## 6.2 Optimizations

Currently, no optimizations are used to improve performance. Since this thesis focused on performance across transport types, this needed to remain consistent throughout. Optimizations, such as those used in HERD, improve performance significantly. Other optimizations can be found in Kalia et. al. paper on design guidelines and possible optimizations[21].

This thesis has shown the potential of using UD as scalable and high throughput transport type for RDMA KV stores, however this could be improved further. One improvement is to have multiple groups of clients, each with a single QP. This has been shown to be effective in ScaleRPC[9]. Grouping clients reduces RNIC cache contention, by limiting the number of worker threads competing for cache. TODO However, like with RC and UC, using multiple QP's increases context switching, which harms performance. ScaleRPC have

proposed solutions by "warming up" QP's.

## 6.3 Newer hardware

Performance benchmarks for this thesis have been ran on DAS-5. DAS-5 is becoming outdated, with the newer DAS-6 being available mid-2021[6]. DAS-6 will make use of 100 Gbit/s Infiniband RNIC, near 2x bandwidth compared to DAS-5.

# 7

# Conclusion

RDMA has been shown to be a promising advancement for key values stores. The low latency that RDMA brings with it, opens op new possibilities, not only for key value stores, but also for remote storage in DRAM. RDMA also requires more design decision to be made, which transportation protocol to use, which optimizations to use, and which verbs to use. In this thesis, the transportation protocols RC and UD have been compared, on performance, to TCP.

**RQ1:** In industry, KV stores require to be scalable and low latency. Connection based transportation (RC and UC) put stress on RNIC when using large number of clients, hindering performance with respect to throughput and scalability. RC has similar scalability as TCP, with a near 60 kilo-tasks/sec increase in throughput across all number of clients. Best performing transportation type is UD. It has been shown that UD offers a 58.7% improvement in throughput against RC, and 45.5% improvement in latency. For scalability UD is recommended to use, as this offers the best overall performance and sustains this up to 30 clients, and has the potential for optimizations to further improve performance.

**RQ2:** The advantages and disadvantages present with the different transportation types impact the KV store design choices. In context of KV stores, UD is a compelling transportation type, however does not offer RDMA verbs such as READ and WRITE, which could offer for improved performance. UD also allow for more optimizations, due to their one-to-many QP. UD performance in this thesis have found to be limited. Multiple UD QP's along with client grouping could continue scalability further. Connection based transportation types are limited by one-to-one QP, although also have some room for improvements.

# References


[1] memcached: free and open source, high-performance, distributed memory object caching system. http://memcached.org, 2003. Online: accessed 28 May 2021. 5

[2] Das-5: Distributed asci supercomputer. https://www.cs.vu.nl/das5/home.shtml, 2012. Online: accessed 28 May 2021. 15

[3] Matplotlib: Visualization with python. https://matplotlib.org/, 2016. Python graphing and visualization tool. 14

[4] Pandas. https://pandas.pydata.org/, 2016. Data analysis and manipulation tool. 14

[5] Python. https://www.python.org/, 2016. Python programming language. 14

[6] Das-6: Distributed asci supercomputer. https://www.cs.vu.nl/das6/home.shtml, 2021. Online: accessed 28 May 2021. 26

[7] Rdma based key-value store. https://github.com/jtro0/RDMA-KV-store, 2021. All relevant thesis project files are openly avaliable as of June 2021. 2

[8] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012. 2, 5, 15

[9] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable rdma rpc on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–14, 2019. 25

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007. 1, 2

## REFERENCES


[11] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. URL `https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}`. 1

[12] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 25–36, 2012. 1, 5

[13] Philip Werner Frey and Gustavo Alonso. Minimizing the hidden cost of rdma. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 553–560. IEEE, 2009. 6, 8

[14] Roxana Geambasu, Amit A Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M Levy. Comet: An active distributed key-value store. In *OSDI*, pages 323–336, 2010. 1

[15] Richard W Hamming and Roger Pinkham. The art of doing science and engineering: Learning to learn. *Mathematical Intelligencer*, 20(3):60, 1998. 2

[16] Nathan Hanford, Vishal Ahuja, Matthew K Farrens, Brian Tierney, and Dipak Ghosal. A survey of end-system optimizations for high-speed networks. *ACM Computing Surveys (CSUR)*, 51(3):1–36, 2018. 5, 6

[17] Gernot Heiser. Systems benchmarking crimes. https://www.cse.unsw.edu.au/ gernot/benchmarking-crimes.html, 2010. "Online: accessed 28 May 2021". 2

[18] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin Van Eyk, Lucian Toader, Vincent Van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. The atlarge vision on the design of distributed systems and ecosystems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1765–1776. IEEE, 2019. 2

[19] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. john wiley & sons, 1990. 2

[20] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 295–306, 2014. 1, 2, 7, 14, 19, 23

[21] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance {RDMA} systems. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pages 437–450, 2016. 6, 25

[22] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided ({RDMA}) datagram rpcs. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 185–201, 2016. 2, 7, 8, 13, 19

[23] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. {MICA}: A holistic approach to fast in-memory key-value storage. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 429–444, 2014. 1, 5

[24] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided {RDMA} reads to build a fast, cpu-efficient key-value store. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 103–114, 2013. 2

[25] John Ousterhout. Always measure one level deeper. *Communications of the ACM*, 61 (7):74–83, 2018. 2

[26] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010. 5

[27] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007. 2

[28] Haonan Qiu, Xiaoliang Wang, Tianchen Jin, Zhuzhong Qian, Baoliu Ye, Bin Tang, Wenzhong Li, and Sanglu Lu. Toward effective and fair rdma resource sharing. In *Proceedings of the 2nd Asia-Pacific Workshop on Networking*, pages 8–14, 2018. 8, 13

[29] Sameer Seth and M Ajaykumar Venkatesulu. *TCP/IP Architecture, Design, and Implementation in Linux*, volume 68. John Wiley & Sons, 2009. 5

[30] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G Andersen, Gregory R Ganger, Garth A Gibson, and Brian Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. *ACM SIGCOMM computer communication review*, 39(4):303–314, 2009. 8

[31] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. Stackmap: Low-latency networking with the {OS} stack and dedicated nics. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pages 43–56, 2016. 6