Vrije Universiteit Amsterdam

Bachelor Thesis

# Scalability of RDMA transportation types in a Key Value store application

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

| | |
|---|---|
| *1st supervisor:* | Animesh Trivedi |
| *daily supervisor:* | Animesh Trivedi |
| *2nd reader:* | Jesse Donkervliet |

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

June 25, 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. A fast a reliable service is required to perform well under heavy workloads, with performance changes being unnoticed for the end user, such as a shopping customers at Amazon. The increase in number of customers and demand for such services, require data centers to scale resources, to maintain performance. Previously, KV-stores have maintained performance through improvements to the underlying data structure, typically a hash table. However, the increasing popularity of Remote Direct Memory Access (RDMA) networks in data centers, potentially gives rise to other advancements that can be made to KV-stores. The main advantage of such RDMA networks, is that lower latency can be achieved. This is crucial in latency sensitive applications, such as KV-store.

RDMA makes use of three so-called transport types: reliable connection (RC), unreliable connection (UC), and unreliable datagram (UD). Each transport type has its advantages and disadvantages. Previous work has not explicitly focused on transport type choice in an RDMA KV-store application. This paper focuses on this gap, evaluating multithreaded scalability performance of RDMA transportation types. These findings will be used to give recommendation for RDMA KV-store implementations.

This paper made use of multithreaded KV server, with a worker thread for each additional client. Clients threads are created by a benchmark application, and perform a set number of random KV-store operations, in total 10 million operations. These operations are generated randomly, with a 95% probability of a GET operation, and 5% probability of SET operation. This has been used as a realistic workload, to perform macro-level benchmarks. Additionally, two designs have been evaluated, with and without thread waiting. This has been done, as large number of threads results in high CPU usage.

After conducting experiements on the DAS-5 computing cluster,the results have shown that UC performs best out the three transport types. With wait, UC stabilizes around 273 thousand ops/sec. Without wait, UC reaches a maximum throughput of 370 thousand ops/sec, at 32 clients. After 32 clients, the maximum physical threads have been reached, and performances drops below TCP without waiting. This loss in maximum performance has been shown to be caused by context switching. UC achieves lower latency compared to TCP, in both blocking and non-blocking. However, the outliers in latency have been to shown to be significantly higher without blocking. Therefore, without any optimizations, it is recommended to use UC with blocking when exceeding maximum number of physical threads. UD performance close to UC without blocking, and does allow for more optimizations, thus UD is recommended with optimizations.

All relevant project files can be found on Github[9].

# 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 a variety of services: product recommendations, user preferences, and advertisements[13, 16]. Amazon's Dynamo is an example of such a highly available and used KV-store. Dynamo is used by Amazon to provide its core services, such as session storage and shopping cart[13]. At peak, these services handle tens of millions customers a second. This requires Dynamo to process requests in the span of a few milliseconds, to provide a consistent and fast user experience.

Remote direct memory access (RDMA) networks have increasingly become more popular in data centers, due to their high throughput, low latency, and lower cost of RDMA capapble network interface cards (RNIC)[12, 23]. RDMA and RNIC provides a lower latency networking interface compared to classical network cards, which is advantageous for latency sensitive applications, such as a KV-store, deployed in data centers.

## 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[12, 23]. Additionally, in a higher level sense, there are design choices to be made, in regard to transportation types and so-called verbs. RDMA networks can handle of various transportation types: reliable connection (RC), unreliable connection (UC) and unreliable datagrams (UD). RDMA makes use of multiple verbs, which can be seen as network operations. There are four main verbs: *SEND*, *RECV*, *READ*, and *WRITE*. Each transportation type and verb come with advantages and disadvantages, which will be discussed further in this thesis. However, 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[22, 24, 27].

Scalability is an important factor for tech giants such as Amazon. With the constant growth in customers and demand, for example for Amazon, services must be scalable to maintain performance with growing number of clients.

## 1.3 Research Question

This paper will explore to what extent RDMA transportation types affect the performance and multithreaded scalability of KV-store, under two designs: blocking and non-blocking. For this, research will be conducted to answer the following questions:

**RQ1** What is the multicore scalability of the RDMA transport types: reliable connection (RC), unreliable connection (UC), and unreliable datagram (UD)? Scaling up by adding more resources to a system, in this case number of threads, is a more cost-effective strategy, compared to scaling horizontally by adding more nodes. For this reason, multithreaded scalability of these RDMA transport types will be investigated, with a blocking and non-blocking design.

**RQ2** What are the advantages and disadvantages of RDMA transport types when implementing a RDMA KV-store? This question aims to aid with design process of future RDMA KV stores. By applying known ramifications of RDMA to that in a KV store use case. With supporting results, recommendations can be given, and possible design issues can be foreseen.

## 1.4 Research Methods

In this thesis the network performance of KV store, with varying network protocols, will be studied. First an understanding of KV-stores is established. Along with discussing known issues with traditional networking, RDMA will be introduced. Experiments will be performed to measure overall throughput and latency, with increasing number of clients. With this the possible scalability of each transport type, in both a blocking and non-blocking design, will be shown.

A prototype[17, 20, 31] and experimental[19, 21, 28] 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[9].

**M2** To investigate the performance and scalability between the various networking types, a workload realistic[10] 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 this is shown in table 3.1 and discussed in section 3.3.1

## 1.5 Thesis Contributions

This thesis presents scalability performance of RC, UC, and UD. These 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, and which design choices are available to improve scalability. These findings can be used to further examine possible optimizations or RDMA verb choices, or aid in future RDMA KV store implementations. The prototype and all project files can be found on Github[9].

## 1.6 Plagiarism Declaration

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

For this thesis, the basic structure of KV store from course Operating Systems, at the VU, has been used. Along with basic RDMA functions from Dr. Trivedi's RDMA example[7]. This has been used as a starting point for RC implementation.

## 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 socket, 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[29]. 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 has been the target for advancements in KV stores, using lower latency, memory efficient, and scalable data structures[14, 25].

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[10]. This figure will be used in the benchmarking design, see section 3.3.

## 2.2 Linux socket and TCP network stack

The Linux socket programming API, is versatile, and offers a simple interface to network communication. Behind this socket interface, the kernel is tasked with data and memory, and connection management[18, 33]. 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[18]. Memory copies have been found to cause delay in packet processing[15]. However, for small messages in the order of 100 bytes, Yasukata, and team, found that memory copies is insignificant to packet processing time[35].

Past attempts at improving TCP performance included offloading most processing to the network interface card (NIC)[18]. The socket API still requires system calls, among other limitations, thus requiring kernel trapping for management of socket queues, and consuming packets.

Figure 2.1 shows the path a receiving packet takes inside a Linux system. Packet is processed by CPU, and system calls and memory copies needed between stacks and queues.

## 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 [23], 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 **??** and figure 2.1 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, which can be seen in 2.2. Furthermore, RDMA offloads packet processing to the RNIC, freeing CPU cycles for other tasks. 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

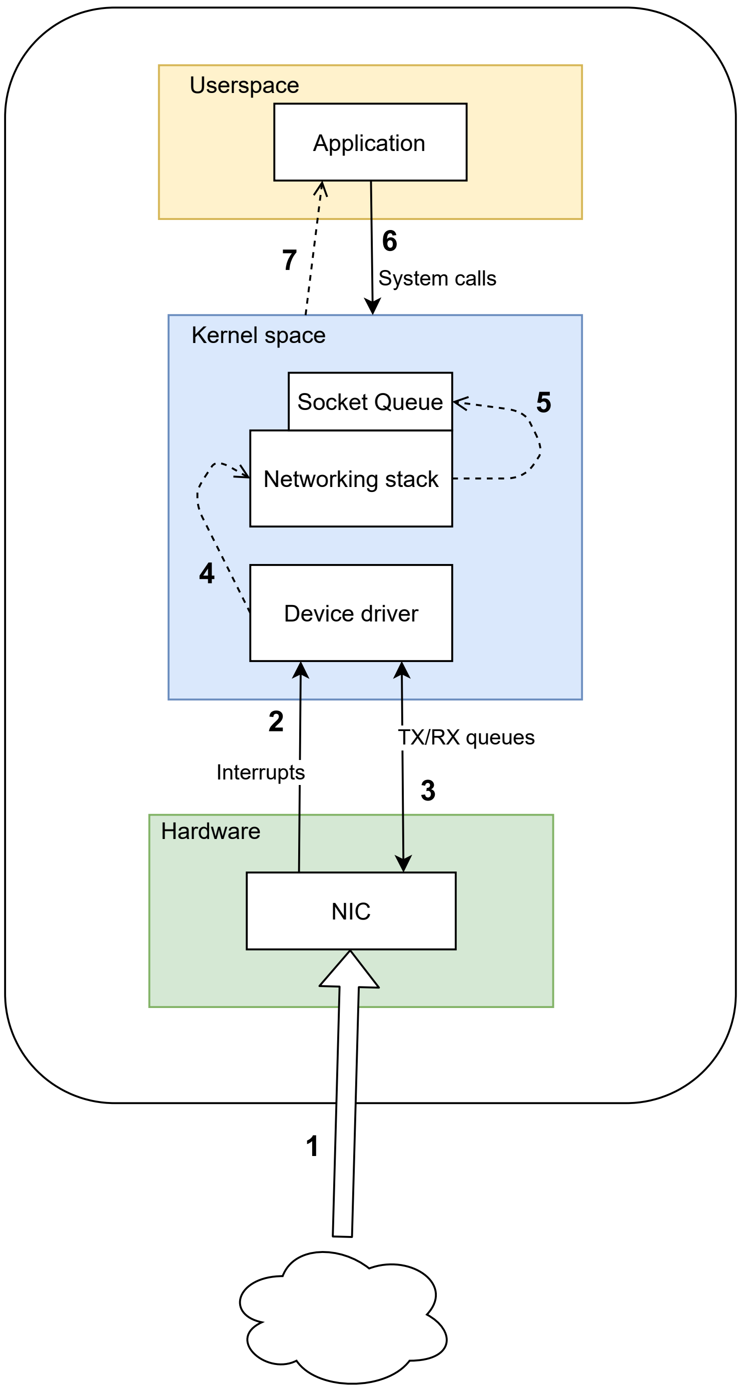


**Figure 2.1:** System overview of hardware to application for receiving a packet. Path is as follows: 1. remotely a packet is sent and recieved by NIC. 2. Hardware interrupt to the device drive, indicating new packet. 3. Device driver dequeues packet in RX queue. 4. Networking stack processes packet. 5. Packet is enqueued in queue assosiated with the socket. 6. Application performs a read system call, and performs *epoll_wait* if necessary. 7. Packet is copied and consumed by application.

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. 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.

For RDMA operations like *READ* and *WRITE* this would include the remote address to be accessed, with two-sided verbs such as *SEND* and *RECV* this is accompanied by a buffer for which DMA operations can take place.

### 2.3.2 Transportation types

RDMA can make use of several transport protocols, each having their trade-offs. Simply put, there are three main and supported transport 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[22, 24]. 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. Every QP has to be held by RNIC in cache, which is limited in space[32]. At large number of QP's, this might not be able to be cached, thus causing cache misses when switching between all QP's. This cache miss would diminish performance, as QP metadata must be retrieved from main memory. Data center workloads typically run with many connected machines[34]. If a connection oriented transport type is used in such use cases, performance could be limited[24].

UD requires an address handler (AH) to be passed along with its WR's. AH hold the relevant data for routing a packet to its destination. The destination does not need to be a QP, but could also be a multicast group. Multiple clients can listen on the same channel, this is can be applied in applications where the same data can be used by multiple clients.

| | **Application verb SEND** | **RECV** | **READ** | **WRITE** | **Operation 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

An AH can be created from a previous work completion (WC), or via metadata exchange, see 2.3.4.

UD is limited however in its 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[15].

### 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 (*READ* and *WRITE*) follow so-called memory semantics, while send and receive (*SEND* and *RECV*) 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 similar to socket, in the sense that the remote memory address does not need to be known. However, to perform a *SEND* operation, the receiving end must post a *RECV* WR before the *SEND* WR is sent, this is called pre-posting. 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.

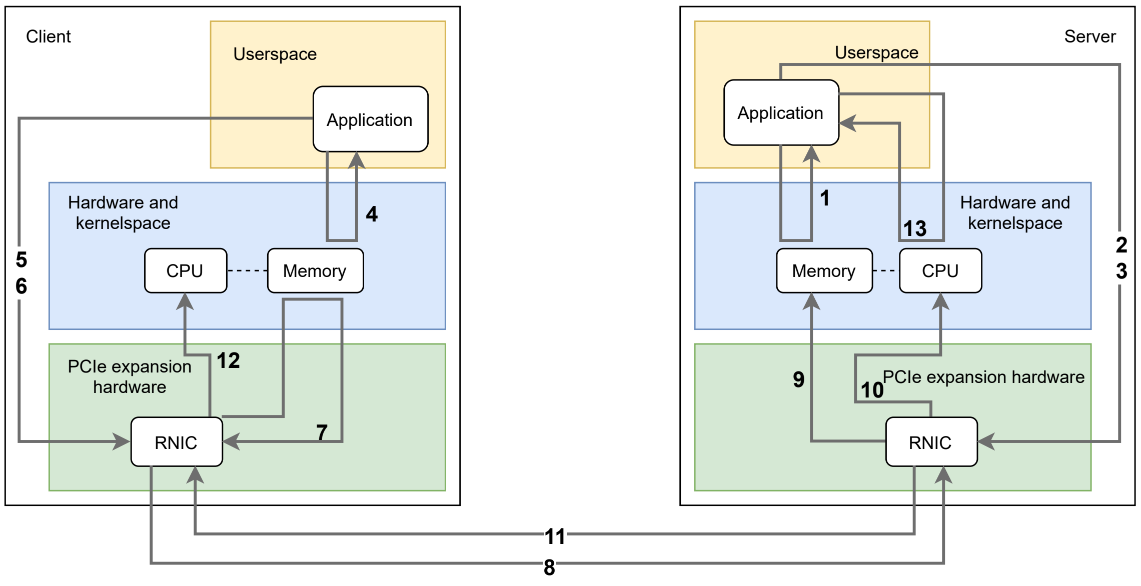


**Figure 2.2:** Process overview of pre-posting receive and send from client to server. Numbered arrows have a corresponding function described in 2.3.5. Looping arrows such as 1 or 7, indicate memory fetching/polling.

### 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 networks, using for example TCP. However, in some cases, a classical ethernet NIC is not available. In these circumstances, a known QP can be used. This is available by the Communication Manager (CM)[11, 26].

Unconnected datagrams also require metadata exchange. Data such as QP number, and physical port, which is are needed to create AH, or join a multicast group. This can be achieved in the same way as with connected protocols.

### 2.3.5 Programming with RDMA

Unlike traditional sockets, much of RDMA's programming API allows for more control, thus more detailed optimizations can be implemented. This interface is developed and supported by the OpenFabrics Alliance[2]. The cost of this is the need for more memory management in userspace, which otherwise would be done by the kernel. In this subsection, this open interface will be discussed, what functions are needed to be done to make use of RDMA, and what happens internally. For this, figure 2.2 illustrates the process of an RC QP connection, and *SEND* and *RECV* operation, from client to server. It is assumed that client and server have successfully made a protection domain (PD), QP, and connect or exchanged data for these QP's, see section 2.3.4 on how this can be done. Protection domains are used to separate resources, such as memory regions, QP's, and AH.

The numbered arrows in figure 2.2 correspond to the following:

1. The server application should allocate memory for its receiving buffer.
2. This buffer should be registered in the RNIC under the PD used for this application. By registering memory, a page table entry will be held in RNIC's cache, as shown with the corresponding dashed line.
3. Server pre-posts a receive. This is needed to be able to receive before the client will send. Usually this is done just before connecting, this way even if the receiving end lags behind, the RNIC will be ready to receive.
4. Client allocates the buffer that will be sent.
5. Also registering this buffer. This can be done while the server is completing steps 1 and 2.
6. Posting a *SEND* WR to the RNIC. This signals the RNIC that a given registered memory address will be sent via its QP.
7. The RNIC fetches this data in memory using DMA.
8. In step 8, this data is then sent over the network to the server.
9. The server's RNIC receives this data and performs an DMA operation to the buffer given by its *RECV* WR.
10. RNIC notifies with a completion event to the CPU, that it has a new event in its CQ.
11. RNIC also sends a completion event to the client, in the case of reliable connection.
12. This completion event is passed along to the CPU, to signal a waiting thread.
13. The server can poll on the CQ, this tells the application that the *RECV* request has been completed, and can now be read with the same data as was in the clients buffer.

For more insight in which functions are avaliable, see RDMA programming guide[26] or the OpenFabric Alliance[2] which develop kernel bypassing API for RDMA aware networks.

3

# Designing a RDMA Key Value store

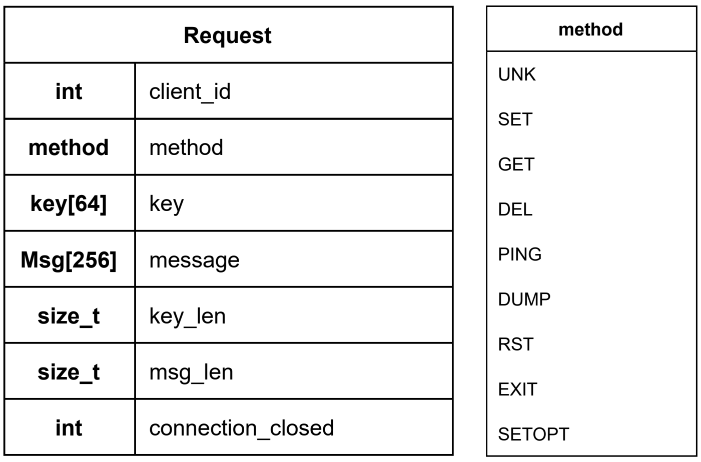
In this section, the design of the used key value store is shown, along with design decision made for RDMA. Further, benchmarking strategy will be explained, used to evaluate the performance of various RDMA transport types. Lastly, an overview of DAS-5 is given, which will be used to run benchmarks. 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 trivial KV store has been used. For this KV store, *SET* and *GET* instructions are mainly used, a long with other commands for testing purposes. All possible commands can be seen in figure 3.1(b).

### 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 3.1(a). This key is used to identify the correct index within the hash table. A field for payload is always sent, however only used for *SET* commands. This is to have a constant packet size, along with fixed sizes for key and message arrays: 64 bytes and 256 bytes. Constant packet size is required for memory registration with RDMA, and the size needs to be known at the destination when posting *SEND*.

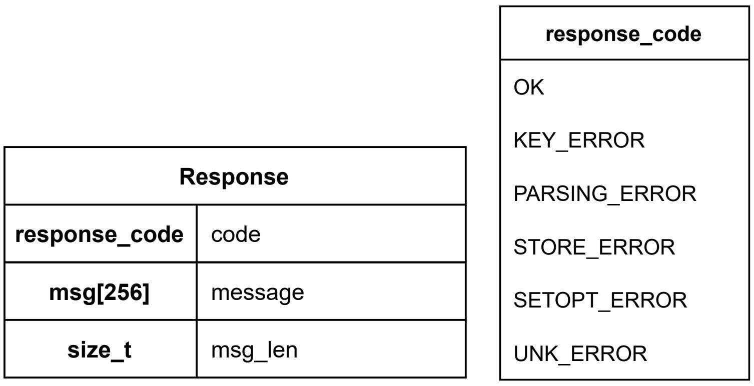


| Request | |
|---|---|
| **int** | client_id |
| **method** | method |
| **key[64]** | key |
| **Msg[256]** | message |
| **size_t** | key_len |
| **size_t** | msg_len |
| **int** | connection_closed |

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

(a) Request structure (b) Values for method

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



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

| response_code |
|---|
| OK |
| KEY_ERROR |
| PARSING_ERROR |
| STORE_ERROR |
| SETOPT_ERROR |
| UNK_ERROR |

(a) Response structure (b) Respond codes

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

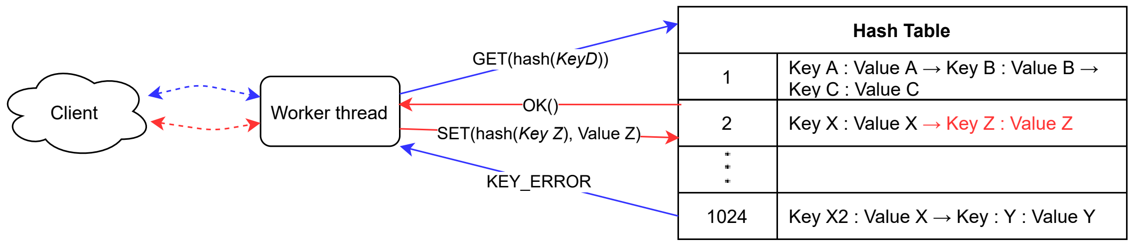

Client
Worker thread
GET(hash(*KeyD*))
OK()
SET(hash(*Key Z*), Value Z)
KEY_ERROR
Hash Table
1
Key A : Value A → Key B : Value B → Key C : Value C
2
Key X : Value X → Key Z : Value Z
1024
Key X2 : Value X → Key : Y : Value Y

**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.

### 3.1.2 Response

After processing a request the server will always send an respond. The structure of which is shown in 3.2(a). 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. Upon error, an error code will be sent back: *PARSING_ERROR*, or *UNKNOWN (UNK)*, latter of which is when the given command is unknown. Other response codes can be seen in 3.2(b)

### 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 with number of elements. Worst case, this approach to hash table has a search time complexity of $O(n)$, where $n$ represents the number of elements in the hash table. This can be partially solved with a large hash table, as this decreases the average size in each bucket. For this thesis, 1024 number of buckets has been found, by trial and error, to be a sufficient size. Additionally, using a constant number of key-value operations, the scalability issues, with respect to increasing number of elements in buckets, is also kept constant with increasing number of clients. More over, in section 3.3 this issue will be kept constant throughout experiments.

### 3.1.4 Multithreaded

Since this thesis focuses on the multicore scalability, the KV server is implemented with multithreading. 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.

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 [24, 32], however is required to provide isolation for the *ACID* principle.

## 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.

To establish a baseline improvement and scalability, no optimizations are implemented. In section 6 possible optimizations are discussed.

### 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.

#### 3.2.1.1 Work completion processing

With large number of threads, exceeding the 32 physical threads present per node of DAS, CPU usage would increase when directly polling. Therefore, two methods are used in this thesis: waiting for completion event and directly polling. In both cases, polling on completion queue is done by consuming a WC one by one.

## 3.3 Benchmark design

To evaluate the performance and scalability of the RDMA KV store, a multiclient benchmark has been made. Each client is run on its own thread, with no shared memory. In total, 10 million key-value operations will be divided among clients. This number is constant to lessen the scalability issues with the underlying hash table. A task involves two network operations, sending a request and receiving response. Only once a response is received from the server, can the client continue to the next task, thus running sequentially. 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 on a per-task level.

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. Time will be gathered with the *gettimeofday* function, and as accuracy achieves measurements at $\mu$sec scale. 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[6] is used, along with pandas[5] and matplotlib[4].

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 a realistic workload of 30:1 *GET*/*SET* ratio, this has been found by Atikoglu et al.[10]. A *SET* request has a 5% chance of being generated. This is done by generating a random number as follows: $rand() \bmod 100 <= 5$. 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[3]. 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. The dual CPUs used per node, result in a total of 32 physical threads.

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.

| 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

# 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 multicore scalability of these transportation types, with increasing number of clients, and therefore threads. For latency three figures have been used to evaluate: average latency, boxplot without outliers, and cumulative distribution function (CDF). The same raw data is used for both measurements, thus can be compared alongside each other. The two designs are analyzed, with and without waiting for completion event.

In short, the results show:

- Without blocking, throughput reaches an peak, with all transportation types, at roughly 32 clients. UC performs best, with a maximum throughput of 370185 ops/sec at 33 clients. Further details are given in section 4.1.

- With blocking, throughput reaches an equilibrium up to 70 clients, on all types: TCP, RC and UC. UC, again, performance best, stabilizing at roughly 272717 ops/sec above 32 clients.

- Latency increases across all transportation types, with increased number of clients, and both with and without blocking. Results are inversely comparable to throughput, again with UC performing best. Without blocking: UC has an average latency of roughly 49.2 $\mu$sec at 32 clients and 179.7 $\mu$sec at 60 clients. The spread also increases: standard deviation ranges from 46.6 $\mu$sec at 32 clients, to 1628.2 $\mu$sec at 60 clients. UC also performance best due to the stable 95th percentile, which is 110 $\mu$sec for both 32 and 60 clients.

With blocking: UC has a average latency of roughly 99.3 $\mu$sec at 32 clients and 158.3 $\mu$sec at 60 clients, and scales in a linear relation. The spread also increases: standard deviation ranges from 150.7 $\mu$sec at 32 clients, to 326.8 $\mu$sec at 60 clients.

Section 4.2 delves deeper with statistical analysis.

To recall the benchmarking setup: Ten million KV-store operations are divided equally between $n$-clients.

## 4.1 Throughput analysis

To compare the overall throughput between TCP, RC, UC, and UD with up to 70 clients. Both designs, with and without waiting for completion event, are analyzed.

### 4.1.1 Throughput analysis without waiting for completion event

With non-blocking design, peak throughput performance is realised at roughly 32 clients. This can be seen in figure 4.1. For RC, maximum throughput is roughly 247782 ops/sec at 32 clients. While for UC, maximum throughput is roughly 370185 ops/sec at 33 clients, and for UD this is at roughly 357019 ops/sec at 31 clients. The maximum throughput TCP achieves in this number of clients range, is 206750 ops/sec at 33 clients. For RC, UC, and UD this is a percentage increase over TCP of: 19.85%, 79.05%, and 72.68%. However, after 32 clients, throughput has a sharp decline, with RC failing to connect properly above 60 clients, resulting in incomplete data at these points.

#### 4.1.1.1 Scalability rate

Multicore scalability of all RDMA transportation types, without blocking, is poor. Up to roughly 15 clients, throughput performance increases at a linear rate. For TCP, there is an average increase of 10246 ops/sec per additional client. For the RDMA transport types, for RC this is 16726 ops/sec per additional client, 21462 ops/sec for UC, and 21067 ops/sec for UD. This shows that unreliable transport types, UC and UD, scale better per additional client and worker thread, compared to RC and TCP. However, after 15 threads, performance begins to plateau for all transportation types and TCP, showing that the used KV-store is CPU bound. This is due to the pthread locking, used to provide isolation with concurrent threads operating on the KV-store.

After 32 clients and worker threads, multicore performance weakens, and dropping in performance. This is due to the non-blocked design, and 32 physical threads being the

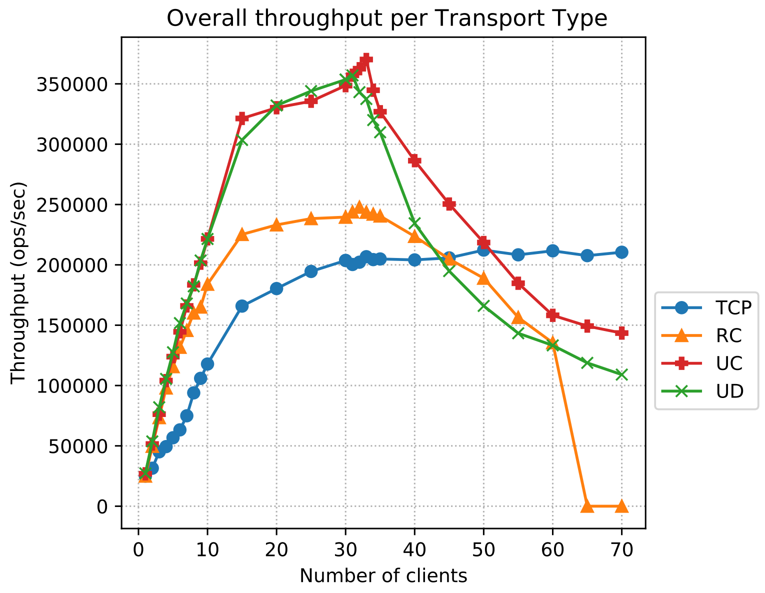


**Figure 4.1:** Throughput of clients executing 10 million key-value operations. Note: RC could not connect reliably above 60 clients, thus removed.

maximum on a single DAS-5 node. Without this blocking or scheduling, threads take CPU cycles while polling on the receive queue. With only a limited number of CPU cycles, this is not used efficiently, resulting in drop in performance for all worker threads.

### 4.1.2 Throughput analysis by waiting for completion event

With a blocking design, maximum throughput decreases to 285395 ops/sec, again with UC transportation type, and UD could not be measured. This maximum throughput is a 22.9% decrease compared to results shown in section 4.1 above. However, UC still achieves a 38.0% increase in throughput compared to TCP. Similarly, RC performance 15.9% better than TCP, with a throughput of 239600 ops/sec. Unlike the results shown without blocking above, multicore scalability has improved, without loss in performance with number of clients above 32. This can be seen in figure 4.2.

#### 4.1.2.1 Scalability of with waiting for completion event

Similar to without blocking, throughput performance scales linearly up to 15 clients. In these results, TCP has an average increase in throughput of 11065 ops/sec per additional client. For RC this is 13339 ops/sec per additional client, and 15770 ops/sec per additional client for UC. For RC and UC, this rate of change is less than seen without blocking, this is to be expected, as the overall performance has depreciated.

| Clients | TCP | RC | UC | UD |
|---|---|---|---|---|
| 1 | | | | |
| 2 | 6531.57 | 24686.53 | 24638.92 | 26502.61 |
| 3 | 13594.19 | 23770.88 | 24785.06 | 28301.23 |
| 4 | 4297.08 | 24568.10 | 27795.82 | 23489.22 |
| 5 | 7363.60 | 17665.62 | 19850.78 | 22090.11 |
| 6 | 6521.77 | 16064.32 | 20579.03 | 24167.81 |
| 7 | 11672.15 | 13975.28 | 21439.07 | 16246.35 |
| 8 | 18989.11 | 14401.89 | 17510.51 | 14201.32 |
| 9 | 12033.20 | 4957.15 | 17930.69 | 21432.35 |
| 10 | 11858.04 | 18916.31 | 20129.91 | 17827.71 |
| 15 | 9603.18 | 8252.11 | 19962.03 | 16410.47 |

**Table 4.1:** Change in throughput per additional client, up to 15 clients. Values are shown in ops/sec per additional client.

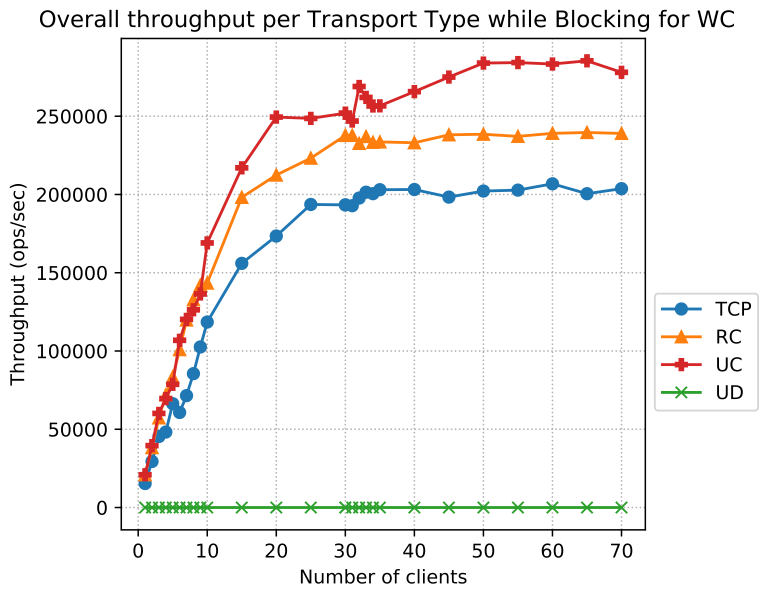


**Figure 4.2:** Throughput of clients executing 10 million key-value operations, waiting for completion event. Note: UD could not result in any reasonable time, or not at all. For this reason this is shown with no significant data.

After 32 clients, RC and TCP reach an equilibrium, while UC reaches equilibrium after 50 clients. This shows that this blocking method, despite lower maximum performance, can remain its throughput performance with high contention.

## 4.2 Latency analysis

An important factor to KV-store is their quick response time. Latency should ideally scale similarly, preferably better, than throughput, such that with an increased number of clients, latency changes are unnoticed to clients. RDMA should offer faster response time, by their bypassing kernel design. In this section, the extent of this is explored, for both methods of polling: with and without waiting for completion event by blocking.

### 4.2.1 Latency analysis without waiting for completion event

With a polling, non-blocking design, RDMA transportation types perform better than TCP up to 55 clients, after which UD performs worse, and RC following this trend. This can be seen in 4.3. At roughly 32 clients, ranking is similar to throughput as seen in figure 4.1. Results show that TCP has an average latency of 138.0 $\mu$sec at 32 clients. RC has an average latency of 115.1 $\mu$sec, a 16.61% improvement relative to TCP. UC performs best at 32 clients, with an average latency of 49.24 $\mu$sec, which is an 64.33% improvement over TCP. Finally, UD has an average latency of 65.5 $\mu$sec, which is an improvement of 52.5% over TCP.

It can also be observed that latency scales proportional with the number of clients for TCP, while this is only the case for RDMA transport types up to roughly 32 clients. The average rate of change for TCP, RC, UC, and UD up to 32 clients is: 2.31 $\mu$sec, 2.00 $\mu$sec, 0.37 $\mu$sec, and 0.72 $\mu$sec. This shows that UC scales favorably up to 32 clients, closely followed by UD. After 32 clients, RDMA transport types scale worse, with UD surpassing TCP's average latency at 60 clients. RC is trending towards surpassing TCP after 60 clients, however, this data could not be collected reliably. Lastly, UC still performs favorably compared to TCP at 70 clients, however, relative performance is now at 21.98%.

#### 4.2.1.1 Variation in Latency performance

Consistent latency is important for clients, such that performance is predictable, and provide reliable services to end users. To examine the variation in latency with increasing number of clients visually, two types of graphs are used: box plot and CDF graph. Graphs show the spread in latency results at three points: 5 clients, 32 clients and 60 clients.

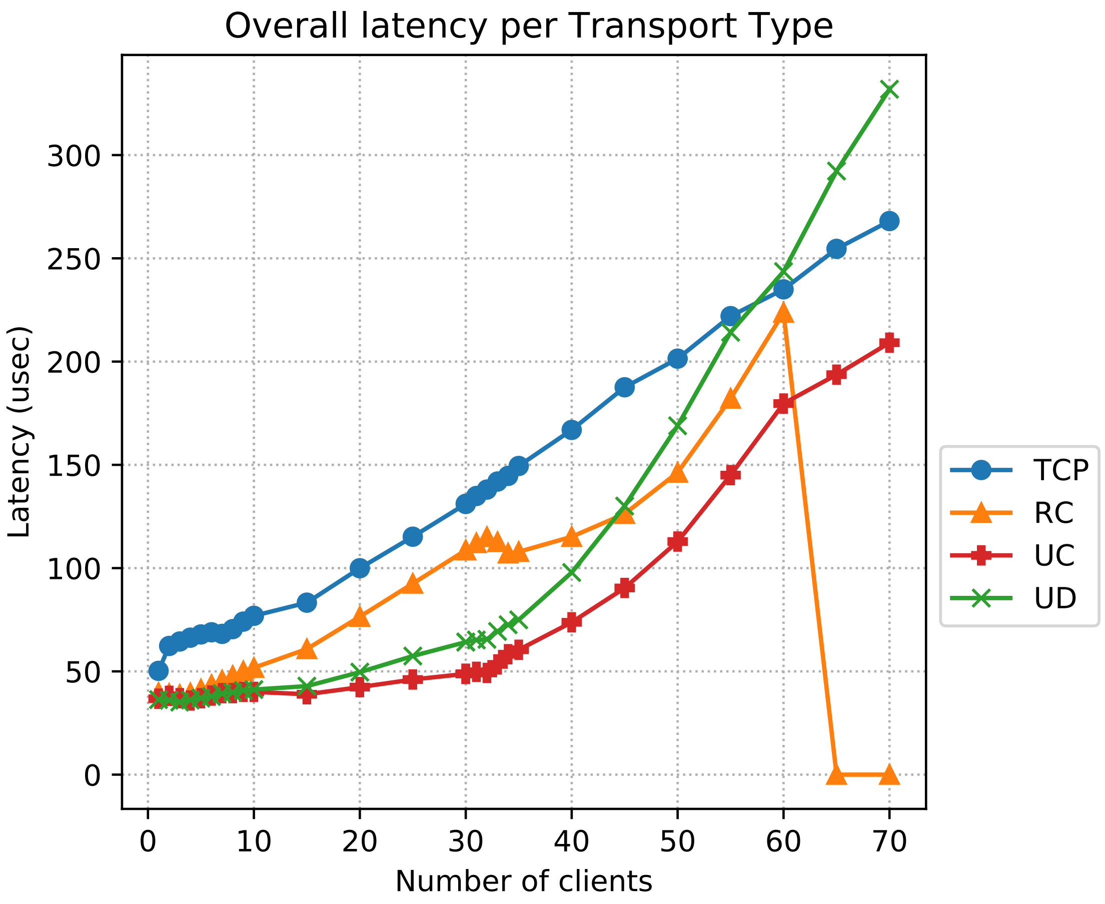


**Figure 4.3:** Latency of clients executing 10 million key-value operations

Additionally, the median, innerquartile range, 95th and 99th percentile, and standard deviation, is used to examine the spread numerically. This numeric data can be found in table 4.2, for the three sets of data. A complete table, with all number of clients up to 70, can be found in table INSERT TABLE in the appendix.

From table 4.2 it can be observed that for 5, 32 and 60 clients, standard deviation increases as number of clients increase. With RDMA, there is a significant increase in variation after 32 clients. The median latency and 95th percentile of UC and UD remains similiar between 32 and 60 clients, however the outliers or 99th percentile, increase significantly, this can be seen in 4.2. This is further supported by the CDF figure of 32 clients, figure 4.4(b), and 60 clients, 4.4(c). In these figures, 80% of the data is near 100 $\mu$sec and 90 $\mu$sec, for UD and UC respectively, deviating slightly between 32 and 60 clients. However, the top 1% of the data is not below the 500 $\mu$sec shown for 60 clients, while this is the case for 32 clients. These outliers are caused by polling with concurrent number of threads, which exceed the number of physical threads. With high CPU usage, the time until a new request is seen and can be processed, increases, causing a delay for the client. These outliers can be the cause of loss in throughput performance, seen in figure 4.1.

It can also be observed that RC suffers from a peak spread around 32 clients, which afterwards declines. This can be seen in table 4.2(b) and figure 4.5. Innerquartile range reaches a peak of 149 $\mu$sec at 32 clients, at a median of 112 $\mu$sec. It is unclear what causes this, however should be considered when reaching maximum number of physical threads.

### 4.2.2 Latency analysis by waiting for completion event

Figure 4.6 shows that latency performs worse below 32 clients, compared to non-blocking design, however scales better beyond 32 clients. Below 32 clients, RC has an average difference of 7.6 $\mu$sec compared to non-blocking, performing worse. UC has an average difference of 22.2 $\mu$sec. After 32 clients and up to 50 clients, this gap increases to on average 23.6 $\mu$sec for RC and 42.1 $\mu$sec for UC. Beyond 50 clients, waiting for completion events show to be favorable, compared to without. At this point, RC has an favorable average difference of -6.9 $\mu$sec, and UC -19.7 $\mu$sec.

Additionally, from figure 4.6, it can be observed that all three types, TCP, RC and UC, scale linearly. Out of the three types, UC has the least rate of change in latency per additional client, thus performing favorably. The rate of change of UC is roughly 1.60 $\mu$sec, while RC and TCP have a value of 2.39 $\mu$sec and 2.66 $\mu$sec, respectively. At this rate, UC and RC will remain providing lower latency compared to TCP, with an increasing gap.

| **Clients** | **Median** | **IQR** | **95th percentile** | **99th percentile** | **Standard deviation** |
|---:|---:|---:|---:|---:|---:|
| 5 | 62 | 55 | 126 | 152 | 33.50 |
| 32 | 129 | 40 | 192 | 233 | 268.86 |
| 60 | 238 | 65 | 318 | 368 | 275.85 |

(a) TCP

| **Clients** | **Median** | **IQR** | **95th percentile** | **99th percentile** | **Standard deviation** |
|---:|---:|---:|---:|---:|---:|
| 5 | 31 | 43 | 88 | 99 | 29.33 |
| 32 | 112 | 149 | 227 | 231 | 83.10 |
| 60 | 48 | 56 | 120 | 7852.41 | 1662.83 |

(b) RC

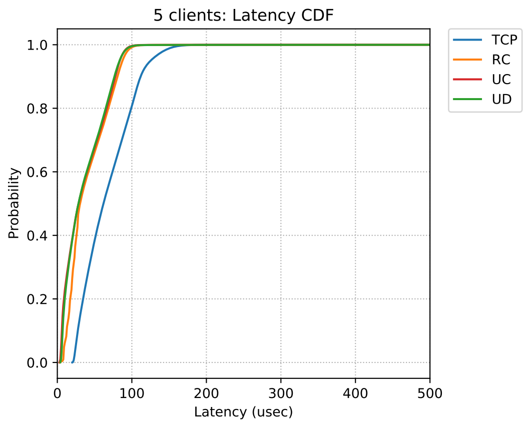
| **Clients** | **Median** | **IQR** | **95th percentile** | **99th percentile** | **Standard deviation** |
|---:|---:|---:|---:|---:|---:|
| 5 | 29 | 49 | 84 | 95 | 29.26 |
| 32 | 39 | 70 | 110 | 122 | 46.59 |
| 60 | 35 | 56 | 110 | 1296 | 1629.16 |

(c) UC

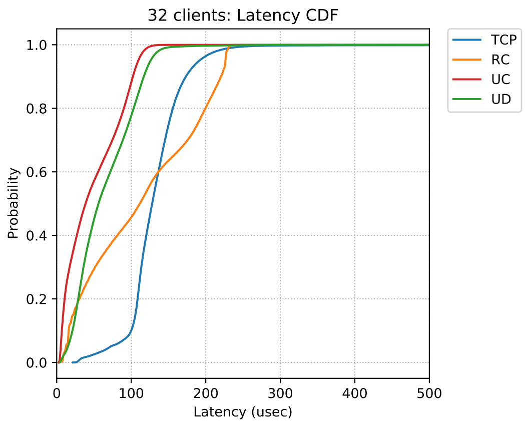
| **Clients** | **Median** | **IQR** | **95th percentile** | **99th percentile** | **Standard deviation** |
|---:|---:|---:|---:|---:|---:|
| 5 | 29 | 46 | 84 | 95 | 30.45 |
| 32 | 57 | 64 | 126 | 146 | 65.59 |
| 60 | 54 | 65 | 143 | 9036 | 1693.72 |

(d) UD

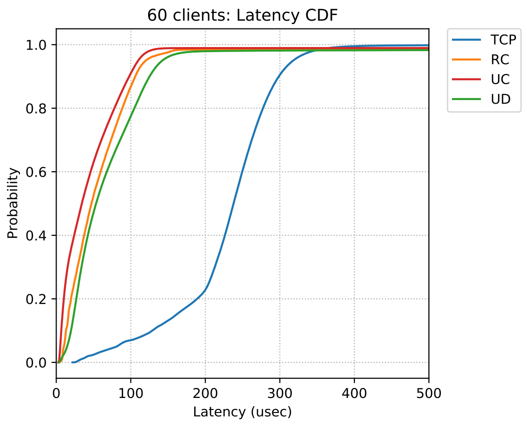
**Table 4.2:** Numerical statistics latency. All statistical values have unit $\mu$sec. IRQ is short for innerquartile range



(a) 5 clients

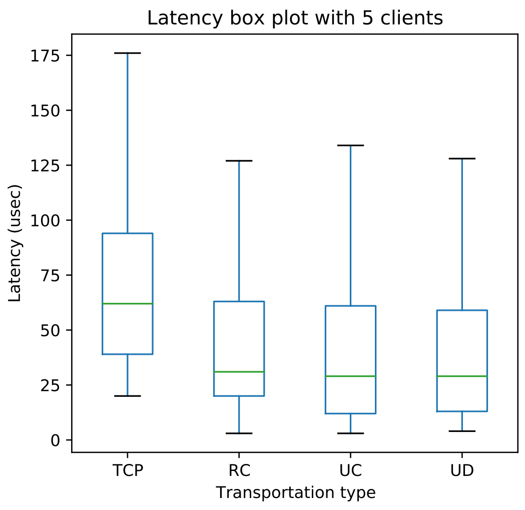


(b) 32 clients

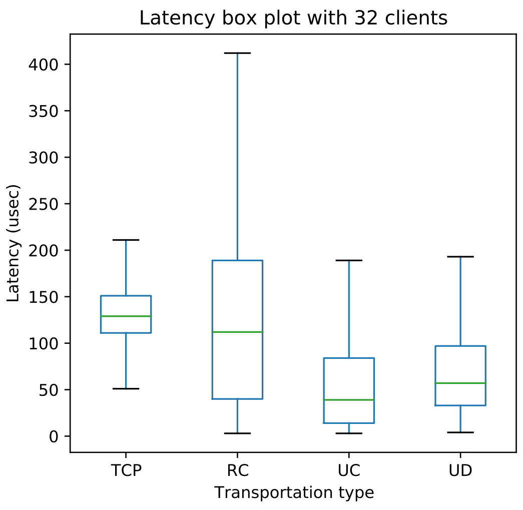


(c) 60 clients

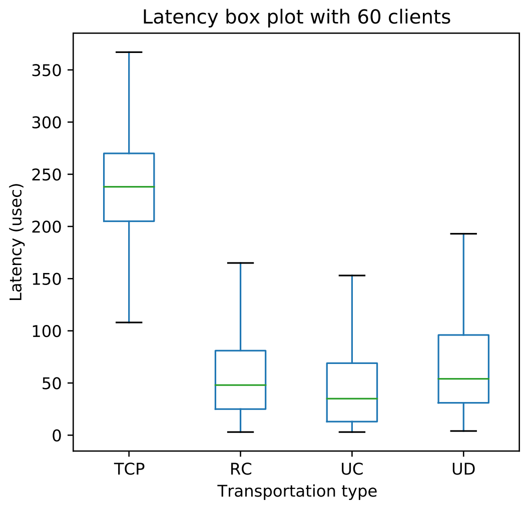
**Figure 4.4:** Latency cumulative distribution function (CDF) for 5, 32 and 60 clients.



(a) 5 clients



(b) 32 clients



(c) 60 clients

**Figure 4.5:** Latency box plot for 5, 32 and 60 clients.

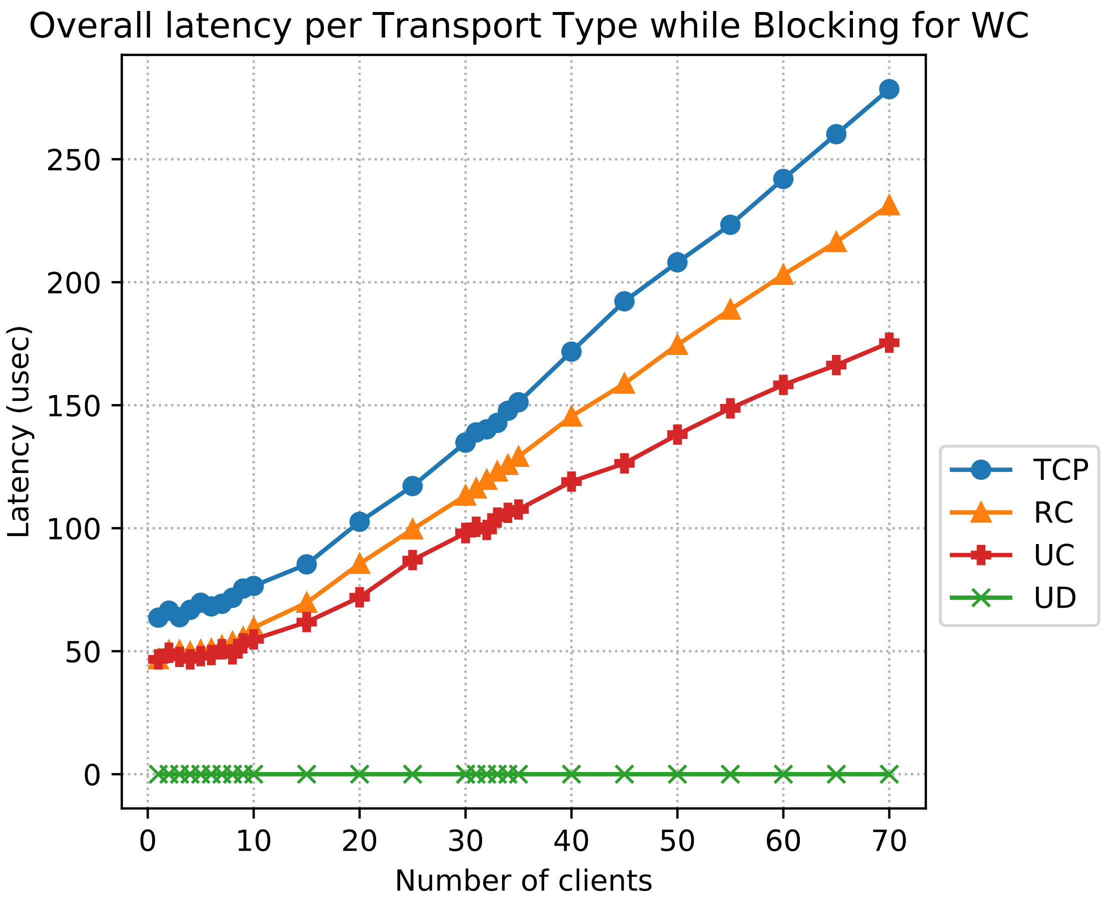


**Figure 4.6:** Latency of clients executing 10 million key-value operations, with waiting for completion event

| Clients | Median | IQR | 95th percentile | 99th percentile | standard deviation |
|---|---|---|---|---|---|
| 5 | 64 | 53 | 131 | 155 | 34.42 |
| 32 | 130 | 38 | 193 | 241 | 266.65 |
| 60 | 245 | 66 | 326 | 379 | 273.75 |

(a) TCP

| Clients | Median | IQR | 95th percentile | 99th percentile | standard deviation |
|---|---|---|---|---|---|
| 5 | 43 | 47 | 101 | 120 | 31.93 |
| 32 | 111 | 118 | 219 | 231 | 109.48 |
| 60 | 152 | 303 | 430 | 441 | 200.03 |

(b) RC

| Clients | Median | IQR | 95th percentile | 99th percentile | standard deviation |
|---|---|---|---|---|---|
| 5 | 41 | 47 | 98 | 112 | 31.64 |
| 32 | 90 | 35 | 152 | 245 | 150.74 |
| 60 | 158 | 55 | 249 | 325 | 326.77 |

(c) UC

**Table 4.3:** Numerical statistics latency, waiting for completition event. All statistical values have unit $\mu$sec. IRQ is short for innerquartile range

#### 4.2.2.1 Variation of latency using blocked design

Comparing the variance between the blocked design and without blocking, shows significant improvement in variation and spread of latency data. This can be seen in table 4.3, again with larger table given in appendix, table INSERT TABLE REF.

In table 4.3, it can be seen that median, IQR, 95th percentile, 99th percentile, and standard deviation, all increase proportionally to the number of clients. Compared with table 4.2, it can be seen that for median, IQR, and 95th percentile, are worse for non-blocking, at 60 clients. However, the outliers are significantly decreased. The 99th percentile for UC at 60 clients, is 325 $\mu$sec, which is significantly less than that found for non-blocking design, which is at 1296 $\mu$sec. Additionally, standard deviation is found to be significantly better for blocking design, 326.77 $\mu$sec at 60 clients for blocking UC design, compared to

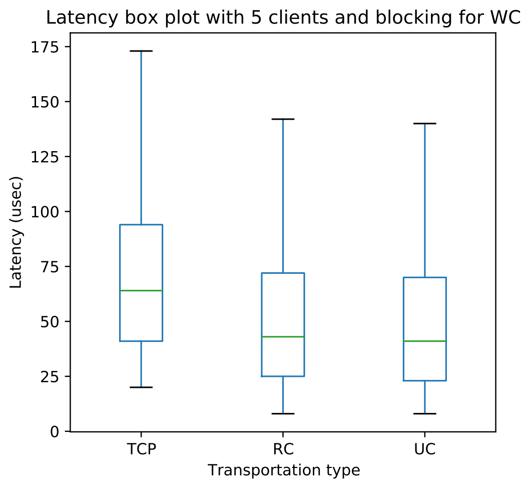
1629.16 $\mu$sec for similar non-blocking design.

Furthermore, RC can be seen to have a wide spread after 32 clients, compared to UC, in figure 4.7(b) for 32 clients, and figure 4.7(c) fr 60 clients. Innerquartile range is 118 and 303 $\mu$sec for 32 and 60 clients for RC. Compared with UC's innerquartile range of 35 and 55 $\mu$sec, shows a significant increase in spread. However, RC has less outliers, as its standard deviation is less compared to UC, as can be seen in table 4.3. This can be further seen in the CDF figures in figure 4.8. From the figures 4.8(b) and 4.8(c) it can also be seen that RC has a slight bimodual distribution at 32 clients, and a more prominent at 60 clients. Additionally, it can be seen in these figures, that UC is more normally distributed in 32 and 60 clients.

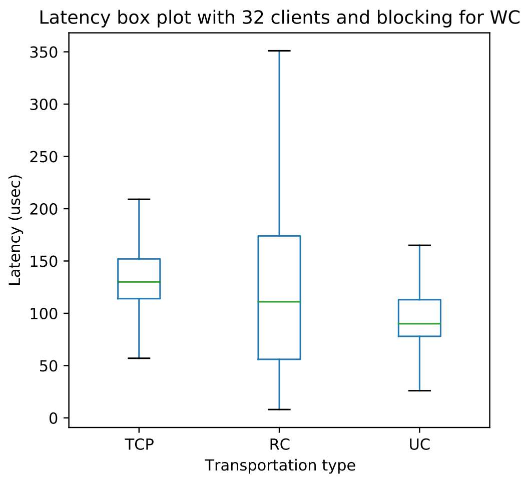
## 4.3 Effect of CPU context switching between blocking and non-blocking designs on performance

In the analysis above on throughput and latency performance between blocking and non-blocking design, has shown a loss in initial performance, but normalization beyond the physical thread count. With blocking, CPU utilization decreases, as threads are waiting to for completition event, without using CPU cycles. For this to happen however, CPU must use context switching to (re)store execution when blocking. This causes delay, as required data has to be loaded and unloaded inside the CPU. This delay is seen as a increase in latency, and decrease in maximum performance. However, as shown above, this only harms performance below 32 threads, as this is the number of physical threads. Beyond 32 threads, context switching causes throughput performance to remain equal, while without switching, contention would cause CPU cycle loss and drop performance. Table 4.4 shows the increase in number of context switches when blocking for completion event. It can also be seen that number of context switches is roughly equal to $2 * numberofoperations$, as server worker threads would block when receiving request, but also when sending response, for benchmark this would be when sending request and receiving response.

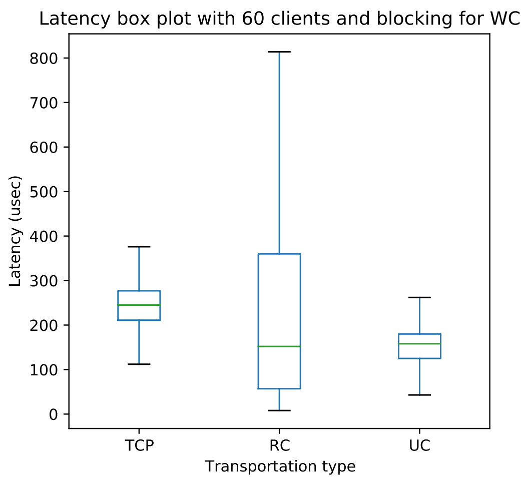
Also, the trend in throughput and latency, found when blocking, is similar to that of TCP. This is due to the fact that the TCP implementation as shown in this thesis, does a similar blocking and context switching.



(a) 5 clients

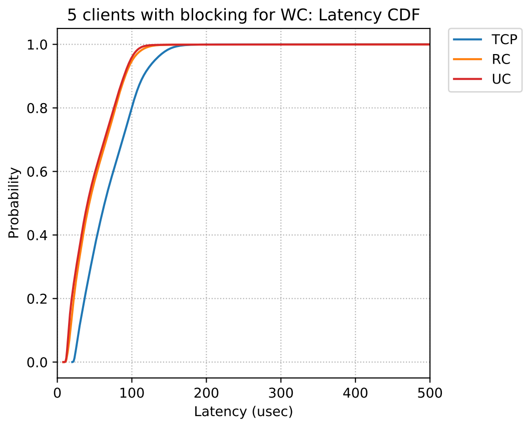


(b) 32 clients

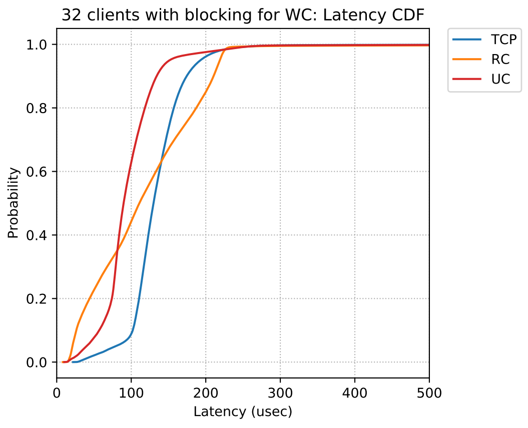


(c) 60 clients

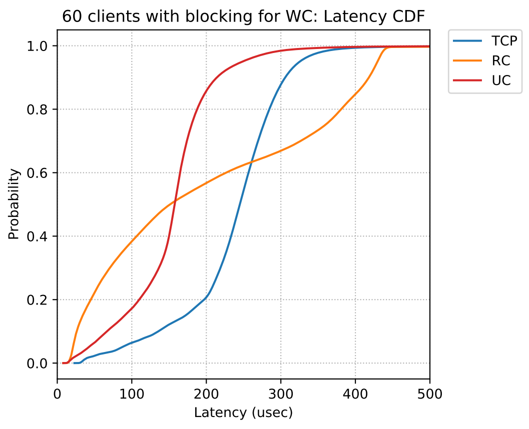
**Figure 4.7:** Latency box plot for 5, 32 and 60 clients with waiting for completion event.



(a) 5 clients



(b) 32 clients



(c) 60 clients

**Figure 4.8:** Latency cumulative distribution function (CDF) for 5, 32 and 60 clients, and with waiting for completion event.

| Clients | No blocking: server | No blocking: benchmark | Blocking: server | Blocking: benchmark |
|---|---|---|---|---|
| 5 | 1,453 | 47,120 | 19,547,566 | 19,813,183 |
| 32 | 7,208 | 716,238 | 20,055,568 | 19,972,089 |
| 60 | 191,667 | 574,049 | 20,077,619 | 19,980,293 |

**Table 4.4:** Context switching for UC with and without blocking. Data collected by *perf stat*, on both server and benchmarking side.

# 5

# Related Work

## 5.1 HERD

There are several proposed RDMA KV store designs. One of which, HERD [22], 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 incoming *WRITE*s offers lower latency and higher throughput compared to *READ*. Outgoing *WRITE*s have also been shown to not scale well, making this unadvisable for sending responses.

HERD has been shown to perform well with increasing number of clients. Factors which contributed to this:

- 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 with network operations. To know if a request has been received with *WRITE*, polling/checking on memory changes is required.
- 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* with UC, given that inlining is possible. It was found that for many 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.

## 5.2 ScaleRPC

ScaleRPC makes uses of RC and efficient resource sharing, to provide a scalable RPC[12], comparable to HERD[22]. Like HERD, ScaleRPC makes use of prefetching to decrease latency when receiving requests. Additionally, ScaleRPC groups clients to balance contention to the RNIC cache. With this performance and be held with increasing number of clients, and thus increasing number of connected QPs.

## 5.3 FaSST

FaSST shows comparable performance to HERD and ScaleRPC. FaSST is built on two-sided verbs with UD, and most makes use of batching to increase performance[24]. Kalia et al. have shown that packet loss is rare, even with UD, and can be ignored.

# 6

# Future improvements

## 6.1 Experimental transport types

Current transport types all have advantages and disadvantages. The transport types investigated in this thesis, RC, UC, and UD, are supported by Mellnox. New and experimental transport types, such as Dynamically Connected Transport (DCT), are under development. 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. However, could have the potential to combine the reliability of RC and the scalability of UD.

## 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, making use of prefetching and inlined data, improve performance significantly. Other optimizations can be found in Kalia et. al. paper on design guidelines and possible optimizations[23]. These are hardware focused, including batching, reducing the number of PIO and DMA operations.

This thesis has shown the potential of using UD and UC as scalable and high throughput transport type for RDMA KV stores, however this could be improved further. One improvement to UD is to have multiple groups of clients, each with a single QP. This has been shown to be effective in ScaleRPC[12]. Grouping clients reduces RNIC cache contention, by limiting the number of worker threads competing for cache. However, like with

RC and UC, using multiple QP's increases context switching, which harms performance. ScaleRPC have proposed solutions to this by "warming up" QP's.

## 6.3 Key-Value store

Results have shown that underlying key-value store was limiting network performance. A more advanced hashing table, such as Cuckoo[30], can be used to improve scalability. Additionally, a more efficient locking mechanism could improve performance of the KV store.

## 6.4 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[8]. 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 up new possibilities and improved performance. 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, UC, and UD have been compared, on performance, to TCP, using both blocking and non-blocking methods.

**RQ1:** In industry, KV stores require to be scalable and low latency. For scalability, UC has been shown to be the best performing transport type. When directly polling, UC reaches maximum performance of roughly 370 thousand ops/sec, with 33 clients connected. This is a 79% increase over TCP. UC also performed best in regard to latency, achieving roughly 49 $\mu$sec, which at 32 clients, is a 64% increase over TCP. However, much like all other transport types, above 33 clients, throughput and latency performance drops below TCP's throughput performance, while UC nears TCP's latency. This shows poor scalability for all RDMA transport types with contesting threads, causing high CPU usage.

Waiting for completion event, has shown proper improvements to both latency and throughput, when exceeding physical thread count. By blocking, CPU usage decreases, at the cost of context switching. Below 32 clients, context switching results in 23% decrease in throughput performance. Above 32 clients however, throughput is reaches an equilibrium, much like TCP. UC beats RC in performance, reaching a maximum throughput of roughly 285 thousand ops/sec. This is a 38% improvement over TCP. In context of latency, UC scaled well here too. Latency, and its standard deviation, remained proportional to number of clients. UC remained below TCP and RC, achieving an average latency of 99 $\mu$sec at 32 clients, and 158 $\mu$sec at 60 clients. Along with TCP and RC, variance on latency increases with number of clients, this is to be expected. However, it has been found that outliers are significantly reduced compared to the non-blocking method.

Therefore, for scalability UC is recommended to use. If number of threads will not exceed the maximum number of physical threads, then non-blocking achieves higher throughput and latency. However, with multithreading beyond physical thread count, blocking would be recommend, as throughput and latency performance remains consistent.

**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. An example optimization would to 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. However, UC is easier to implement with multiple worker threads, as there is one QP per client. Thus, without any optimizations, UC is recommended to be used.

# Appendix

# 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] Openfabrics alliance. https://www.openfabrics.org/, 2004. OpenFabrics Alliance: develops, tests, licenses, supports and distributes RDMA/Advanced Networks software. 10, 11

[3] Das-5: Distributed asci supercomputer. https://www.cs.vu.nl/das5/home.shtml, 2012. Online: accessed 28 May 2021. 17

[4] Matplotlib: Visualization with python. https://matplotlib.org/, 2016. Python graphing and visualization tool. 17

[5] Pandas. https://pandas.pydata.org/, 2016. Data analysis and manipulation tool. 17

[6] Python. https://www.python.org/, 2016. Python programming language. 17

[7] Rdma exmaple. https://github.com/animeshtrivedi/rdma-example, 2019. RDMA example implementation used as inspiration for RC implementation and RDMA helper function. 3

[8] Das-6: Distributed asci supercomputer. https://www.cs.vu.nl/das6/home.shtml, 2021. Online: accessed 28 May 2021. 38

[9] 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. iii, 3

[10] 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. 3, 5, 17

[11] Dothan Barak. Rdmamojo. https://www.rdmamojo.com/, 2012. RDMAmojo is a blog on RDMA technology and programming. Last accesed 28 May 2021. 10

[12] 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. 1, 36, 37

[13] 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

[14] 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. 5

[15] 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, 9

[16] 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

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

[18] 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

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

[20] 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

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

[22] 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. 2, 8, 35, 36

[23] 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. 1, 6, 37

[24] 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, 8, 16, 36

[25] 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. 5

[26] *RDMA Aware Networks Programming User Manual.* Mellanox Technologies, 350 Oakmead Parkway Suite 100, Sunnyvale, CA, USA, 1.7 edition, 2015. Online: accessed 28 May 2021. 10, 11

[27] 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

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

[29] 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

[30] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004. 38

[31] 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

[32] 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, 16

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

[34] 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

[35] 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