Universidade do Minho
Escola de Engenharia
Departamento de Informática

José Miguel Ribeiro da Silva

RDMA mechanisms for columnar data in analytical environments

January 2021

Universidade do Minho
Escola de Engenharia
Departamento de Informática

José Miguel Ribeiro da Silva

RDMA mechanisms for columnar data in analytical environments

Master dissertation
Intregrated Master's in Informatics Engineering

Dissertation supervised by
Professor Doutor José Orlando Roque Nascimento Pereira
Doutor Fábio André Castanheira Luís Coelho

January 2021

ACKNOWLEDGEMENTS

First of all, I want to thank my parents for allowing me to get an academic degree, without them this dissertation would not be possible (Obrigado aos meus pais por me darem acesso ao percurso académico, sem eles esta dissertação não seria possível). I also want to thank my supervisors, Dr. Orlando Pereira and Dr. Fábio Coelho, thank you for your dedication and patience.

It was also essential for this dissertation the availability of Minho Advanced Computing Center (MACC) and Rui Gonçalves. It was vital for this dissertations' success.

All the support from my girlfriend, brother, and friends was very important for this accomplishment, and for that, I thank you all.

STATEMENT OF INTEGRITY

ABSTRACT

The amount of data in information systems is growing constantly and, as a consequence, the complexity of analytical processing is greater. There are several storage solutions to persist this information, with different architectures targeting different use cases. For analytical processing, storage solutions with a column-oriented format are particularly relevant due to the convenient placement of the data in persistent storage and the closer mapping to in-memory processing.

The access to the database is typically remote and has overhead associated, mainly when it is necessary to obtain the same data multiple times. Thus, it is desirable to have a cache on the processing side and there are solutions for this. The problem with the existing solutions is the overhead introduced by network latency and memory-copy between logical layers. Remote Direct Memory Access (RDMA) mechanisms have the potential to help minimize this overhead. Furthermore, this type of mechanism is indicated for large amounts of data because zero-copy has more impact as the data volume increases. One of the problems associated with RDMA mechanisms is the complexity of development. This complexity is induced by its different development paradigm when compared to other network communication protocols, for example, TCP.

Aiming to improve the efficiency of analytical processing, this dissertation presents a distributed cache that takes advantage of RDMA mechanisms to improve analytical processing performance. The cache abstracts the intricacies of RDMA mechanisms and is developed as a middleware making it transparent to take advantage of this technology. Moreover, this technique could be used in other contexts where a distributed cache makes sense, such as a set of replicated web servers that access the same database.

Keywords: RDMA, cache, analytical processing, columnar data, distributed systems

RESUMO

A quantidade de informação nos sistemas informáticos tem vindo a aumentar e consequentemente, a complexidade do processamento analítico torna-se maior. Existem diversas soluções para o armazenamento de dados com diferentes arquiteturas e indicadas para determinados casos de uso. Num contexto de processamento analítico, uma solução com o modelo de dados colunar é especialmente relevante devido à disposição conveniente dos dados em disco e à sua proximidade com o mapeamento em memória desses mesmos dados.

Muitas vezes, o acesso aos dados é feito remotamente e isso traz algum *overhead*, principalmente quando é necessário aceder aos mesmos dados mais do que uma vez. Posto isto, é vantajoso fazer *caching* dos dados e já existem soluções para esse efeito. O *overhead* introduzido pela latência da rede e cópia de *buffers* entre camadas lógicas é o principal problema das soluções existentes. Os mecanismos de acesso direto a memória remota (RDMA - Remote Direct Memory Access) têm o potencial de melhorar o desempenho neste cenário. Para além disso, este tipo de tecnologia faz sentido em sistemas com grandes quantidades de dados, nos quais o acesso direto pode ter um impacto ainda maior por ser *zero-copy*. Um dos problemas associados com mecanismos RDMA é a complexidade de desenvolvimento. Esta complexidade é causada pelo paradigma de desenvolvimento completamente diferente de outros protocolos de comunicação, como por exemplo, TCP.

Tendo em vista melhorar a eficiência do processamento analítico, esta dissertação propõe uma solução de cache distribuída que tira partido de mecanismos de acesso direto a memória remota (RDMA). A *cache* abstrai as particularidades dos mecanismos RDMA e é disponibilizada como *middleware*, tornando a utilização desta tecnologia completamente transparente. Esta solução visa os sistemas de processamento analítico, mas poderá ser utilizada noutros contextos em que uma cache distribuída faça sentido, como por exemplo num conjunto de servidores *web* replicados que acedem à mesma base de dados.

Palavras-Chave: RDMA, cache, processamento analítico, dados colunares, sistemas distribuídos

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

A	
API	Advanced Programming Interface.
C	
CPU	Central Processing Unit.
CQ	Completion Queue.
D	
DDP	Direct Data Placement over Reliable Transports.
DISNI	Direct Storage and Networking Interface.
E	
ETL	Extract Transform Load.
F	
FARM	Fast Remote Memory.
H	
HDFS	Hadoop Distributed File System.
HPC	High-Performance Computing.
I	
IB	Infiniband.
IETF	Internet Engineering Task Force.
IOT	Internet of Things.
IWARP	Internet Wide Area RDMA Protocol.
J	

JNI	Java Native Interface.

L

LAN	Local Area Network.
LRU	Least Recently Used.

M

MPA	Marker PDU Aligned Framing for TCP.

N

NIC	Network Interface Controller.
NOSQL	Not only SQL.

O

OFED	OpenFabrics Enterprise Distribution.

P

PDU	Protocol Data Unit.

Q

QP	Queue Pair.

R

RC	Reliable Connection.
RDD	Resilient Distributed Dataset.
RDMA	Remote Direct Memory Access.
RDMAP	Remote Direct Memory Access Protocol.
RNIC	RDMA-Enabled Network Interface Controller.
ROCE	RDMA over Converged Ethernet.

S

SQL	Structured Query Language.

SVM	Stateful Verbs Method.

T

TCP	Transmission Control Protocol.
TMPFS	Temporary Filesystem.

U

UC	Unreliable Connection.
UD	Unreliable Datagram.
UDP	User Datagram Protocol.

1 INTRODUCTION

Nowadays, the information generated by information systems is growing exponentially, coming from a number of different sources. There is data generated by the casual use of the internet, for example, social networks, messaging services, cloud services, among others. Another source is the path towards the use of fully digital documents and processes by institutions and governments. Moreover, the growing *Internet of Things (IoT)* industry has the potential to generate a massive load of information from sensors and similar devices [10].

The purpose of analytical processing is taking accumulated information as input and introduce the analysis tools to output results reflecting all that data. For example, a university has all the information about its student's grades. It would be interesting to know how the students are performing over the years. Processing this information to get the average grades, filtering it by year and field of study, can help to understand if the institution is fulfilling its goals. It also provides a view of the university's evolution, which might be needed to understand if applied policies are beneficial for students.

The way systems store information is crucial because it determines how the read and write operations will perform. As such, when thinking about a storage solution for any analytical system, we need to know how the information will be accessed. There are diverse storage solutions which are developed for different use cases. Those solutions are often characterized in relational (SQL) or non-relational (NoSQL) storage.

Relational databases use *Structured Query Language (SQL)* to define and manipulate data. The SQL language is very powerful and widely-used, which makes it a safe choice and adequate for complex queries. These databases guarantee strong consistency. As for non-relational databases, the data model is more flexible, focusing on scalability and performance, but often compromising strong consistency.

In any database, the format used to store data in persistent storage is one of the factors that most harms performance. Usually, the formats used are row-oriented or column-oriented (among others such as document or graph oriented). Using a row-oriented format means that each row is stored in persistent storage contiguously. On the other hand, using a column-oriented format means that each column has the values for every row saved contiguously in persistent storage. Row-oriented formats are suitable for use cases where most operations read or save records. Relational databases usually use this format providing consistency guarantees with a transactional system. When this format is used, the use case should not require sequential accesses to specific columns of large sets of rows. A column-oriented format makes sense for aggregation problems as it avoids retrieving useless data from persistent storage. Accessing persistent storage is expensive and thus should be minimized.

As the information grows, analytical processing gets more complex and requires sophisticated algorithms in order to get results within a reasonable time span. These algorithms are developed thinking in a variety of factors that become increasingly important when dealing with more information. When it is possible, the work is distributed across multiple machines. That way, the processing is done distributedly, becoming agile, and efficient.

The analytical processing systems considered in this dissertation are systems similar to Apache Spark [9] or Apache Flink [6]. That is, systems composed of several distributed instances where processing is scattered among all nodes and storage requires remote data accesses in-between nodes. Getting data from a remote database adds overhead. When dealing with a large amount of information, this overhead becomes even more noticeable. Considering the rate of information growth, retrieving remote information must be done efficiently. For example, when data is being processed, it is common to need the same subset of information multiple times. It would be ideal to get this information as few times as possible. Mitigating this issue usually imposes the use of caching mechanisms.

There are multiple strategies to cache data, but the underlying concept is the same: storing a subset of the information close to the client or on hot standby for later use if needed. To make the most of a cache, there must be temporal and spatial locality. Temporal locality means that the same item of data is accessed multiple times in a short time span. Spatial locality means that nearby items of data are accessed in a short time span. In analytical environments, many times, there is a need to have these caches distributed across the nodes, and when accessing them remotely, there is an additional overhead.

The network implementations found in *High-Performance Computing (HPC)* clusters usually offer low latency and high throughput. Infiniband is a networking technology used for interconnecting servers and storage systems. It provides high-throughput, low latencies and facilitates data movement without *Central Processing Unit (CPU)* involvement, using *Remote Direct Memory Access (RDMA)*. RDMA is the capability to access remote memory (read or write) without interrupting the remote processor. With higher CPU efficiency, lower latency and higher bandwidth, Infiniband and RDMA can help deliver better performance in distributed computing environments.

1.1 THE PROBLEM

Nowadays, analytical processing is done by more than one machine or node, taking advantage of distributed processing capabilities or high-performance computing infrastructures. Even though each processing node could have its own individual cache, it is better if a distributed caching mechanism is used. This way, each node can access any value cached on any machine. This is especially relevant in distributed configurations, where processing a query requires data to be shuffled across instances, in order to compute a globally correct answer. There are already solutions for distributed caches. For example, Redis [25] or Memcached. Memcached [22] is an in-memory distributed key-value store, while Redis is very similar to Memcached and provides a variety of data structures including lists and sets.

Solutions for distributed caches usually have two main overheads: the network latency and object serialization. These solutions rely on connections between nodes, usually via sockets (*Transmission Control Protocol (TCP)* or *User Datagram Protocol (UDP)*). Network latency arises from having a connection between the nodes and having data flowing between them. Object serialization is required, in order to format the data in a way that allows it to be transmitted over a network. Serialization introduces overhead as it requires objects to be processed during the marshal and unmarshall stages. Network latency and object serialization overhead are still better than accessing the source of the data but has a negative impact on performance.

Distributed caching still has limitations. Ideally, network speed should be the only limitation, but that is not the case. The copy between memory buffers has a negative impact on the performance. Copies are made on mode switches between the kernel space and user space when receiving or sending network packets. The processing time takes a big hit with memory copies for every packet. These operations should be reduced to a minimum (zero-copy, if possible). As previously described, RDMA provides the capability to access remote memory without involving the remote processor. Thus, it makes sense to consider it when trying to mitigate this limitation because this mechanism is considered to be zero-copy.

1.2 OBJECTIVES AND CONTRIBUTION

This dissertation proposes a solution for distributed caching with low latency accesses (remote and local), focused on RDMA technologies, reducing the network latency to a minimum, aiming at analytical environments. At the same time, the proposed distributed caching mechanism abstracts the development and integration complexity of using network interconnect systems as Infiniband with RDMA, by integration of a software package and make use of a simple *Advanced Programming Interface (API)*.

The main goal is to develop a distributed cache that relies on a mechanism suitable for sharing large amounts of data. The mechanism used should allow data to flow across the nodes in an analytical environment avoiding the conventional I/O circuit. This requirement encourages exploring Remote Direct Memory Access mechanisms. A study exploring to what extent this technology can assist in the development of a distributed cache is provided. Moreover, understanding the limitations can help to mitigate them during the development process.

After studying the technologies, it is necessary to develop a distributed cache prototype that exploits them. This development has to be supplemented by test scenarios, to understand which factors impact performance the most. At an advanced stage of the prototype, it makes sense to compare its performance to distributed caches that do not take advantage of this type of technology. Then it will be possible to infer its viability based on the results.

In order to achieve the proposed objectives, this dissertation will present a set of contributions. Designing an architecture for a distributed cache relying on RDMA will be the first step. The following contribution is a prototype of a distributed cache based on RDMA mechanisms, supporting the usual cache operations. Lastly, the results from the performed tests during the development will be analyzed. This is important as it will be the basis to conclude in what scenarios this prototype could be useful. It will also support a comparison between what latencies could be achieved, in theory, and the ones that were accomplished, in practice.

1.3 THESIS STRUCTURE

This dissertation includes 6 chapters, beginning with an introduction, presenting the problem, motivation and objectives. Chapter 2 provides a solid background to understand the problem and the proposed solution. This chapter also details related work, distributed caching middleware systems, and distributed query processing engines.

Chapter 3 presents the use-case, describing the initial architecture, and introducing the problems we are facing during development based on initial tests.

The core solutions' architecture and specification are described in chapter 4, with every decision made based on some findings made in chapter 3 and evaluations shown in chapter 5.

The performance analysis is outlined in Chapter 5. It includes a comparison between two versions of the solution to support one of the main decisions of the implementation. There are also results comparing the solution to an existent cache to evaluate the performance difference with this solution.

Chapter 6 includes a retrospective of the results obtained and concludes the thesis. It also discusses learned lessons and future work.

2

BACKGROUND

Direct access to remote memory is a concept that is motivated by the prospect of accessing remote memory without interrupting the remote CPU. Different protocols support RDMA and each one has different requirements in terms of hardware. Even though there are multiple protocols, the RDMA operations are available through the same verbs specification.

In analytical processing environments, middleware systems reside between the data source and the process execution to provide better performance. Some of the performance gains are accomplished by caching information in the cluster, avoiding redundant data accesses. As such, we must analyze the existing caching mechanisms and confront them with RDMA-based solutions.

This chapter consists of four main sections: RDMA Networking, Distributed query processing engines, Distributed caching middleware, and RDMA in database engines. The "RDMA Networking" section details Remote Direct Memory Access and tools that aim at mitigating remote accesses' overhead. The "Distributed query processing engines" section details existing computing frameworks and their caching mechanisms. The "Distributed caching middleware" section focuses on middleware designed to improve performance when computing data in a cluster environment using caching. Finally, the "RDMA in database engines" section focuses on projects that have goals similar to this dissertation. These projects will be detailed to understand them in-depth and identify their strengths and weaknesses.

2.1 RDMA NETWORKING

RDMA introduces the capability to access remote memory (read or write) without interrupting the remote processor. This means that one machine can access another machine's memory like if it is its own. For this to be possible, specific hardware is required and it varies depending on the protocol being used.

Currently, there are three network protocols that support RDMA: InfiniBand [3], RDMA over Converged Ethernet (RoCE [1] and RoCEv2 [2]) and *Internet Wide Area RDMA Protocol* *(iWARP)* [24]. All these protocols share the same API, desiganted as Verbs [16]. With this API it is possible to use RDMA from userspace.

The functions present on the verbs API allow userspace programs to access the *Network Interface Controller (NIC)*. The NIC must be ready for RDMA operations and because of that, usually is referred to as *RDMA-Enabled Network Interface Controller (RNIC)* in contexts like this. These operations are posted to queues present on the RNIC. There are two queues, usually referred to as a *Queue Pair (QP)*. One send queue and one receive queue. Each QP has a *Completion Queue (CQ)* that is filled by the RNIC upon fulfillment of operations.

There are two kinds of operations:

One-sided - These operations are READ and WRITE. Using memory semantics to specify the remote memory address to write to or read from, they access remote memory directly. They are considered one-sided because the remote CPU is unaware of it. By not involving the remote CPU, these operations can achieve high throughput.

Two-sided - These operations are: send (SEND) and receive (RECV). The SEND operation transmits a payload that is written to a previously specified buffer on the remote machine with a RECV operation. They are considered two-sided operations because the remote node needs to post a RECV for another machine to perform the SEND operation.

RDMA messages can be transmitted in a connected or unconnected manner. Connected messages require a connection between two QP that only communicate to each other. There are two types of connection: *Reliable Connection (RC)* and *Unreliable Connection (UC)*. The difference in UC is that there are no acknowledgments for package reception and, as a consequence, UC connections produce less network traffic.

Unconnected QPs can communicate with an unlimited number of QPs. There is only one type of unconnected transmission: *Unreliable Datagram (UD)*. RNICs need to sustain state for all the active QPs and for applications where there is a server, having a connection for all clients is unsustainable, as the cache for this information in the network card is limited. Unreliable Datagram makes sense for such applications, but one must consider the unreliability that comes with it.

The operations supported on each connection varies. Table 1 [19] shows the operations supported by each queue-pair connection type.

Operation	RC	UC	UD
SEND/RECV	YES	YES	YES
WRITE	YES	YES	NO
READ	YES	NO	NO

Table 1: Supported RDMA operations by connection type.

The RDMA technology has numerous advantages. Reading and writing directly to remote memory avoids copying data between multiple software layers, thus RDMA is considered a zero-copy mechanism. The data is sent and received in the same context without context switches, which means that the kernel is bypassed. This is one of the biggest advantages, as it allows circumventing the classic I/O circuit on each connection. Sending and receiving data uses dedicated hardware, decreasing the usage of the CPU, as it does not do any active work. A short message can be transferred with really low latency. In current hardware and on current servers, the latency for sending up to tens of bytes can be a couple of hundred nanoseconds [26]. Assuming that Infiniband is used, the bandwidth is very high (from 2.5 Gbits/sec up to 120 Gbits/sec), in contrast to a standard Ethernet device, which is lower (10Mbits/sec or 40 Gbits/sec) [26].

Infiniband (IB) is a network communication standard introduced by the Infiniband Trade Association [3]. One of the key features is the fact that it supports RDMA natively. Infiniband includes specific hardware components for the network physical layer. InfiniBand features low latency and high-throughput data transfer which is ideal for high-performance computing and many other use cases. Using this protocol, in current hardware and on current servers, the bandwidth can be up to 56 Gbits/sec [26].

The *RDMA over Converged Ethernet (RoCE)* protocol was introduced by the Infiniband Trade Association [1]. It is a standard for RDMA over Ethernet, as it substitutes the physical InfiniBand layer with Ethernet. One of the main advantages of this protocol is the fact that it can be implemented on top of an existent Ethernet-based infrastructure without needing to replace the hardware completely as Infiniband requires.

The Infiniband Trade Association also released a second version (RoCEv2) that runs on top of the UDP/IP protocol. This makes it not suitable for lossy networks. But, on the other hand, it is routable on IP networks and it does not add the overhead that a protocol as TCP/IP adds to achieve reliability.

The bandwidth for these protocols depends on the Ethernet technology being used. Ethernet's bandwidth can go from 10Mbit/sec to 40Gbits/sec.

The iWARP protocol is an RDMA implementation that runs on top of the standard network and transport layers, which means it works on any Ethernet infrastructure. The TCP/IP protocol provides flow control and congestion management, meaning that a lossless network is not a requirement. The bandwidth for this protocol depends on the Ethernet technology being used.

Although TCP/IP is used, it is not the host's CPU that runs the protocol. An RNIC is used to handle traffic. The extensions to the TCP/IP added by iWARP were standardized by *Internet Engineering Task Force (IETF)* in 2007 and can be seen on Figure 1 [17].

The *Remote Direct Memory Access Protocol (RDMAP)* [24] layer provides data transfer operations. The protocol specifies seven operations, but the main ones are: Send, Receive,

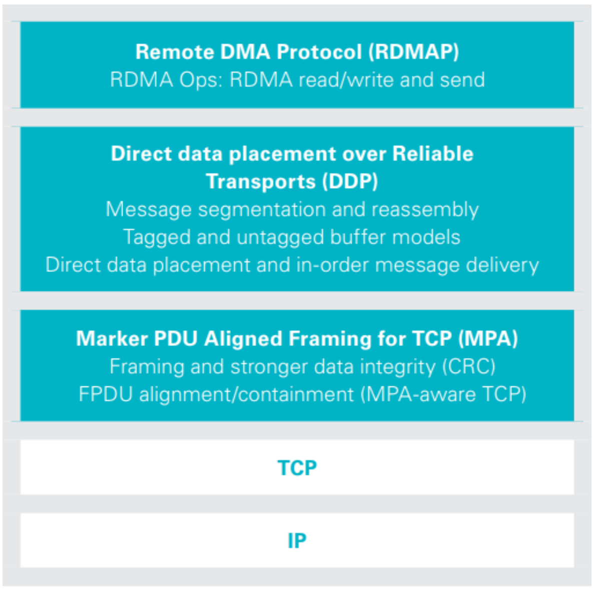
Remote DMA Protocol (RDMAP)
RDMA Ops: RDMA read/write and send
Direct data placement over Reliable Transports (DDP)
Message segmentation and reassembly
Tagged and untagged buffer models
Direct data placement and in-order message delivery
Marker PDU Aligned Framing for TCP (MPA)
Framing and stronger data integrity (CRC)
FPDU alignment/containment (MPA-aware TCP)
TCP
IP

Figure 1: The iWARP protocol.

Read and Write. All other operations are overloads of the ones enumerated except for one: Terminate. The terminate operation sends a message to a remote peer reporting an error that occurred at the local peer.

The *Direct Data Placement over Reliable Transports (DDP)* [28] layer handles all the process of writing to a data buffer. The protocol takes the payload and placement information and writes the data in the appropriate location.

The *Marker PDU Aligned Framing for TCP (MPA)* [14] layer stands between the TCP and DDP layers to preserve the reliable, in-order delivery of TCP adding record boundaries that are required by the DDP protocol.

Figure 2 [17] shows an overview of the RDMA stack for the different protocols that were described on this section.

Examining the Infiniband stack, it is possible to perceive that it requires a network infrastructure that is completely different from a commodity configuration. We perceive that because of the protocols being used. Infiniband provides high bandwidth but also implies a total replacement of the network infrastructure.

The RoCE stack (v1 and v2) is similar to Infiniband but uses the Ethernet protocol for the link layer. Changing the link layer solves the problem of the network infrastructure. But, there is the limitation of the Ethernet's bandwidth. RoCEv2 also uses the UDP/IP protocol, meaning it is routable.

The iWARP stack changes almost completely in comparison to the others. It has its own protocol for RDMA on top of TCP/IP and uses the Ethernet link layer. Using TCP/IP makes it suitable even for lossy networks.

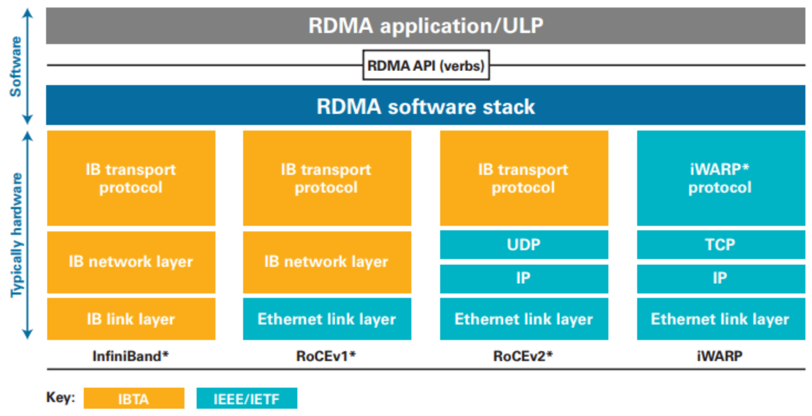

Figure 2: An RDMA stack overview.

2.2 DISTRIBUTED QUERY PROCESSING ENGINES

Usually, analytical processing takes advantage of a computing cluster. Running an application inside a cluster involves many concerns, for example, task scheduling among the workers. Most of the time, a distributed computing framework is used to expedite the development because task scheduling is not an easy task and requires understanding complex concepts of distributed systems. As such, it is important to understand the caching mechanisms of some of the most used distributed computing frameworks to understand their limitations and how an RDMA solution could improve performance.

2.2.1 *Apache Spark*

Apache Spark [9] is a general-purpose distributed computing platform. It extends the MapReduce [11] programming model introduced by Google and popularized by Apache Hadoop [7].

The Spark project is composed of multiple integrated components, including Spark Core, Spark Streaming, Spark SQL, and others. Figure 3 illustrates the Spark architecture and its core components. The core is responsible for scheduling, distributing and managing the tasks running inside the cluster. There are three main elements in the architecture:

- **Driver Program:** the main application that manages the creation.
- **Cluster Manager:** an optional element that is necessary only when Spark is executed in a distributed environment. It is responsible for managing the Worker nodes.
- **Workers:** the nodes that execute the tasks sent by the Driver Program.

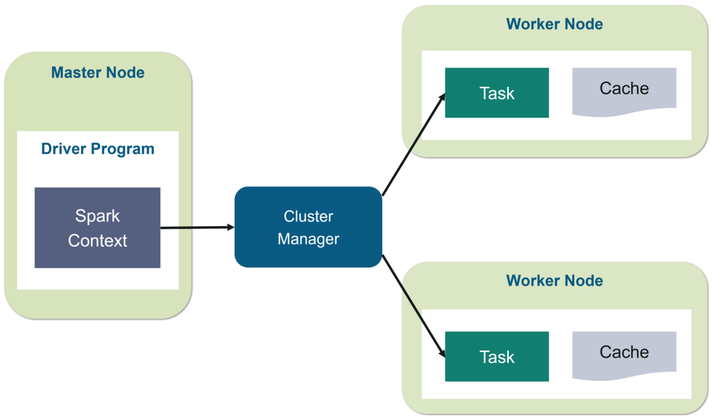

Figure 3: Apache Spark's Architecture.

The main abstraction in Spark Core is the *Resilient Distributed Dataset (RDD)* [29]. An RDD is a fault-tolerant collection of data objects which can be operated in parallel. It is a read-only collection of records that can only be created through deterministic operations (transformations) on data in stable storage or other RDDs. It is possible to control the persistence of a dataset specifying that a given RDD should be persisted in-memory or another storage type, for example, *Hadoop Distributed File System (HDFS)*. The partitioning of an RDD is also configurable based on a key associated with each record. Usually, a Spark program is defined as a sequence of transformations applied to a collection of records.

Apache Spark does not cache any of the results by default. But, it is possible to use some methods provided by the framework. These methods allow applications to cache/persist some intermediate values. The problem of these methods is the need to handle everything at the application level, that is, decide what is kept and what is discarded.

2.2.2 *Dremio*

Dremio [13] is a data lake engine, providing a platform that unifies storage layers with query interfaces. It unifies the storage and interface layers by creating a data-as-a-service platform, as depicted in Figure 4. With this layer between the storage and interface layers, there is no need to implement complex *Extract Transform Loads (ETLs)* which relaxes the need for a classical Data warehouse architecture.

Data has become more complex with time and as a result, in most cases, the data is managed by multiple technologies. Many modern storage solutions include multiple data sources, relational, and non-relational. As such, accessing those data stores involves using multiple interfaces and a solution like Dremio simplifies querying the data, offering polyglot capability.

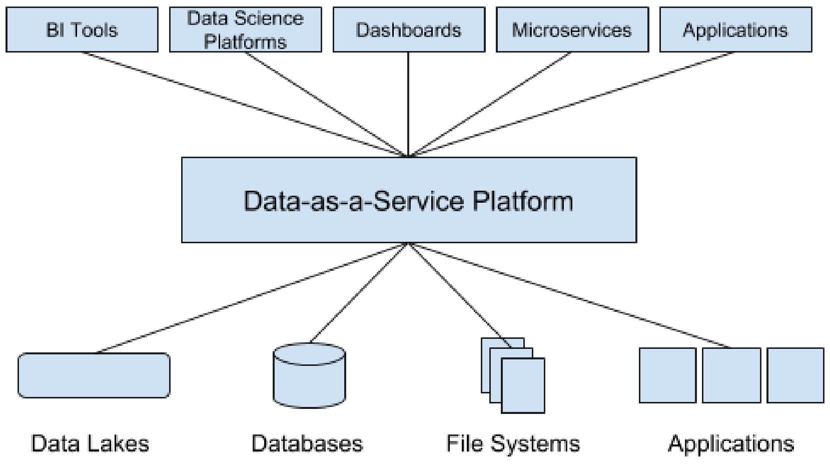

Figure 4: Data-as-a-service platform architecture.

There are two main node types in Dremio, that are typical in distributed computing frameworks:

- **Coordinators:** Responsible for coordinating the Executors, the "map" and "reduce" phases of queries and keeping metadata.
- **Executors:** The workers. These nodes are responsible for executing the actual task.

Similarly to Apache Spark, Dremio is independent of the used storage solution. The executors have a persistence layer implemented on top of Apache Arrow [5]. Apache Arrow is a development platform for in-memory data and uses a columnar memory format for efficient analytic operations.

Reflections are a very important part of Dremio. Reflection is an optimized physical representation of the source data. The query optimizer can use Data Reflections to assist queries. These Reflections might be used for only a fraction of the query or all of it. There are three types of Reflections:

- **Raw:** contain the raw data in the original data source but partitioned and ordered to improve query efficiency.
- **Aggregation:** contain pre-aggregated data, useful to improve analytical query performance (group by followed by sum, min, max, etc).
- **External:** stored outside the distributed storage of Dremio, mapped to another storage system.

Reflections can be saved to multiple distributed storage solutions, for example, HDFS. The archives are stored in the Parquet format [8], a columnar storage format which is ideal when combined with Apache Arrow that is used in Executors as described above.

In Dremio, there are two caching mechanisms. The first one was already detailed because the Reflections described above are cached information closer to where it is going to be used. Even though it is not in memory, it is in Dremio's distributed storage (lower latency than the data source type) and can have pre-calculated data. The other mechanism is the metadata cached in the Coordinator that includes dataset details useful for query planning.

2.2.3 *Apache Flink*

Apache Flink [6] is a framework designed for distributed processing of stateful computations. It is very similar to Apache Spark but optimized for working with streams. The framework is designed to work with bounded and unbounded streams:

- **Bounded streams:** When dealing with bounded streams, there is a known start and end. So the data in the stream will be read from start to end and processed accordingly.
- **Unbounded streams:** As the denomination suggests, an unbounded stream does not have an end. The data in the stream must be processed continuously.

Figure 5 depicts an overview of the Flink architecture, it has a single processor that treats all the input as a stream, and the Streaming Engine processes the data in real-time. In this architecture, both batch and streaming data are processed through the same stream processing engine. This data is fed to the Serving layer that works very similarly to other analytical processing frameworks, for example, Apache Spark. Briefly, it is composed of a Job Manager, that is responsible for scheduling work, and multiple Task Managers that execute the work.

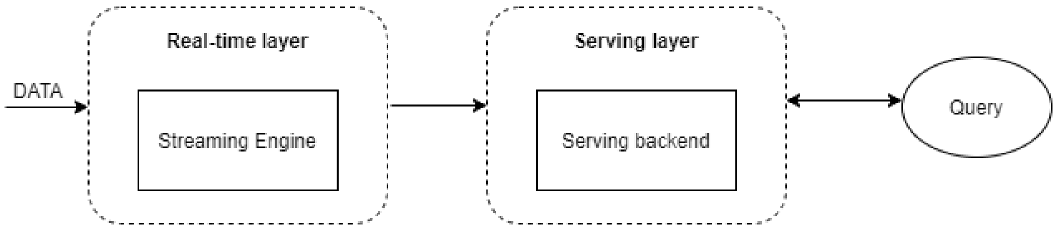

Figure 5: Apache Flink architecture.

Apache Flink offers a distributed cache solution. It is called a distributed cache but not for the reasons one expects. When using this cache what happens is storing a copy of the cached object in each worker machine. The access times are great but memory-wise it is not very efficient.

2.3 DISTRIBUTED CACHING MIDDLEWARE

The middleware systems we are analyzing in this section are designed to improve performance when computing data in a cluster environment. The main concern in this kind of middleware is caching the information closer to the processing machines. As accessing the data source is expensive in terms of performance, each access must be optimized. With that purpose, these systems usually devise a solution to keep the data in the processing machines while it is needed.

2.3.1 *Alluxio*

Alluxio [4] is a virtual distributed storage system. It builds a layer residing between storage systems and computation applications. With Alluxio there is a central point of access, making it easy for applications to access multiple sources of data. Briefly, Alluxio provides an abstraction that hides the complexity of creating connections for all the storage providers and manages available memory, keeping cached values.

The storage systems are referred to as under storage systems in this context. An under storage system is a data source where the actual data is stored, for example, AWS S3, Azure and others. Alluxio hides the integration process of all the under storage systems and therefore can be seen as a unifying layer when multiple storage systems are mounted. With this design, an under storage system can provide data for all the applications running on top of Alluxio.

Alluxio is composed of three types of components: master, workers, and clients. Alluxio's master server is accountable for managing file and object metadata, while the workers manage the respective node space. The clients are used by the applications to connect to the master and worker servers. The master node is a crucial component as it manages the global metadata and, as such, standby replicas should be used to provide fault-tolerance capabilities. If standby masters exist, their job is to keep their copies of the master state up-to-date.

Workers are responsible for managing the local resources allocated to Alluxio on each machine. Random Access Memory has a limited capacity and because of that, the user can configure the available resources, including persistent storage. A worker processes the client requests saving and serving the data as blocks. As the metadata is only stored on the master server, the workers only need to manage the blocks. The workers decide where to store the blocks, using tiered storage. Using tiered storage means that the fastest memory available will be used. Accesses to the under storage are also handled by the workers. The architecture is depicted in Figure 6.

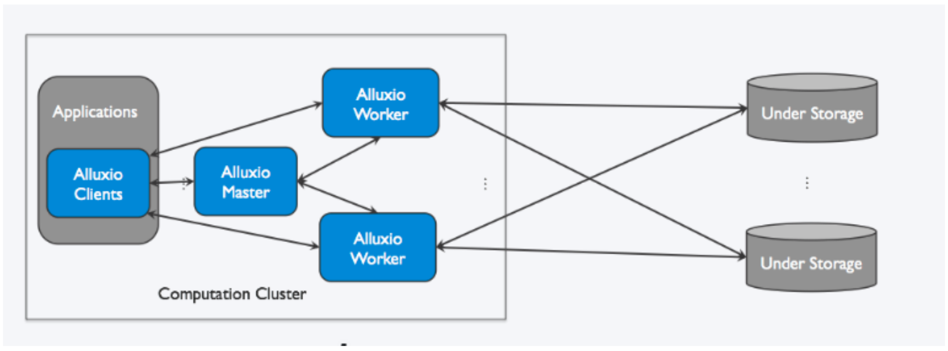

Figure 6: Alluxio's architecture.

For computing applications, this tool uses the storage of the machines where the applications are running. That way Alluxio can serve the information at memory speed if data is local. If data is not local but is present on the computation cluster, it is served at the cluster network speed. Alluxio caches the information that is read, accelerating the data access process.

Using object storage as one of the data sources for data analytics is increasingly adopted when processing is performed with frameworks like Apache Spark. In such architectures, deploying Alluxio alongside Spark, configuring it to persist from object stores, can benefit the applications. Although an object store is indeed easier to scale and maintain, it lacks some capabilities of a filesystem. There is a lack of filesystem-level caching, meaning that different tasks that access the same data cannot benefit from caching frequently accessed data. There is also a lack of node-level data locality because data is always read remotely. Furthermore, some providers for object stores can limit the throughput per computing node, thus the constant remote reads can be problematic. To address these problems, Alluxio caches data locally in the worker nodes, managing the corresponding metadata to optimize data accesses and cache frequently used data.

Unfortunately, Alluxio does not support RDMA for accesses inside the cluster.

2.3.2 *Memcached*

Memcached is an in-memory distributed key/value cache store for arbitrary data. It was originally intended to speed up dynamic web applications by relieving database load. The goal was to take advantage of unused memory on web servers and use it as a logical storage layer to store arbitrary objects associated with a key as depicted in Figure 7.

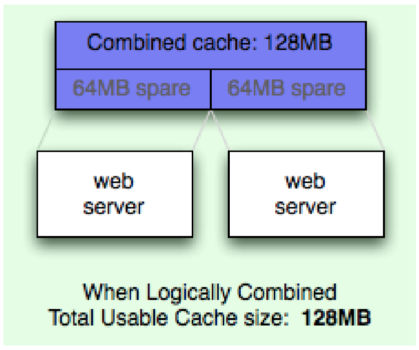

Figure 7: Memcached logically combined memory.

The keys are strings, but the values are handled as raw bytes thus Memcached is unable to parse data structures. The eviction strategy used to manage the available memory space is *Least Recently Used (LRU)*. When there is insufficient space available to add new values or when there are values that were used only a few times, then the least recently used items are discarded.

The servers are independent as there is no communication between them. Because of that, the clients decide which node will store a given key/value pair using a hashing mechanism. The used mechanism is the one published by Bob Jenkins, developed in 1996 and last updated in 2006 [18].

All the commands available are executed in constant time (O(1)) and they can be split into two categories: The ones to save data and the ones to retrieve data.

Even though Memcached does not support RDMA, it is described in this section because it has some similarities to what we are trying to accomplish in this dissertation. There are many solutions for distributed caching, such as Redis and NCache, but for this context, Memcached is more suitable. The reason for that is the fact that the objective of this dissertation is also to develop a distributed cache that handles raw data, using RDMA. Thus, we may compare it to the developed cache, considering it provides an equivalent solution.

2.4 RDMA IN DATABASE ENGINES

Many projects rely on RDMA mechanisms to improve performance in various contexts. But, considering the current context, only projects with similar goals to this dissertation are considered. In this discussion, we analyze work that relies on RDMA mechanisms to develop a performant key-value store or that alleviates the programmer from managing RDMA-level operations directly. The latter is important because RDMA development is not intuitive when the programmer is used to develop TCP-based solutions. Furthermore, this kind of project has the potential to support the development of a key-value store.

2.4.1 *Pilaf*

Pilaf [23] is a key-value store that uses one-sided RDMA reads to be a CPU-efficient solution. This project aims at implementing a key-value store exploiting RDMA operations to reduce overhead.

Pilaf's architecture and operation flow are represented in Figure 8. The main components in this configuration are the Client and Server, relying on an Infiniband network. The client and server, in this case, do not implement the classic client-server configuration, because the client also performs some work.

Using RDMA for all operations without the verbs send/recv abstraction would lead to complex problems (write races, for example) in the solution design, that comes from the fact that it is a one-sided RDMA configuration. Infiniband supports atomic operations but locking over the network affects the performance.

The major design decision is managing all the write operations on the server and implementing all read-only operations on client-side (using RDMA one-sided reads). Write-write races are not a problem with this decision because all the write operations are managed by the server. What can happen is a write-read race but in that case, there is no risk of corrupting data.

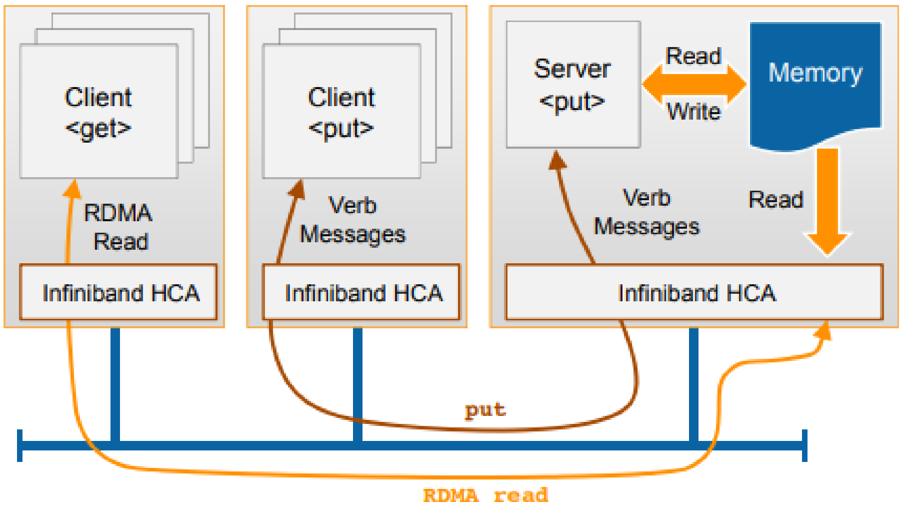

Figure 8: Pilaf's architecture.

The get operation is performed by the clients using one-sided RDMA reads. The server exposes the data structure in two memory regions registered within the network card. One of the memory regions is an array with a fixed size of hash table entries (each one contains a bit indicating if it is in use) and another with the actual keys and values. The client uses linear probing [27] to look for a key in the hash table array. Linear probing is used to eliminate the risk of overwriting data on collisions. If there is a collision, the closest following free location is used, preserving locality for the lookup operations.

Having the server handling all write operations can be underperforming, but that is not problematic as real-world workloads consist of mostly reads. As it was already mentioned, read-write races can happen and Pilaf presents some solutions to cope with it, a self-verifying data structure [23].

The solution presented on this project uses RDMA one-sided operations to offload the CPU. But it is a centralized solution which means the server holds all the information. Having a distributed solution would be interesting for caching scenarios on high-performance computing.

2.4.2 *Herd*

Herd [19] is a key-value store developed for RDMA-capable networks. The main focus lies in checking if one-sided operations are better for performance. The solutions for this kind of system usually use one-sided operations to bypass the CPU. However, the multiple RDMA reads needed, may impact the performance.

Herd introduces a hybrid solution, using both one-sided and two-sided operations. The clients use RDMA writes to post their requests to the server (unreliable connection) and then the requests are processed and completed using RDMA sends (unreliable datagram). Transport-level reliability is sacrificed to improve common case performance, assuming that the need to handle that problem at the application level will be rare. The experiments support the usage of the write operation, revealing that writes have lower latency than reads.

The decisions on this project are focused on network-level improvements. With that in mind, the design to develop the key-value store was borrowed from MICA [21], a key-value store/cache for classical ethernet use cases. It uses a lossy index to associate keys with pointers and stores the values in a circular log. Herd uses multiple server processes (same machine) and each process creates an index and a circular log with a fixed size of 4GB.

Clients write their requests to a memory region on the server, depicted in Figure 9. All the server processes have access to this request region. The request region is divided into 1KB slots which means that the maximum size for a key-value pair in Herd is 1KB. A GET request contains a 16-byte hash of the corresponding key. A PUT request has the key hash, a 2-byte field representing the length of the value and the actual value with a maximum size of 1000 bytes.

Herd provides an important in-depth study of the network level impact of some design decisions for RDMA-based key-value services. For practical usage, Herd is very limited

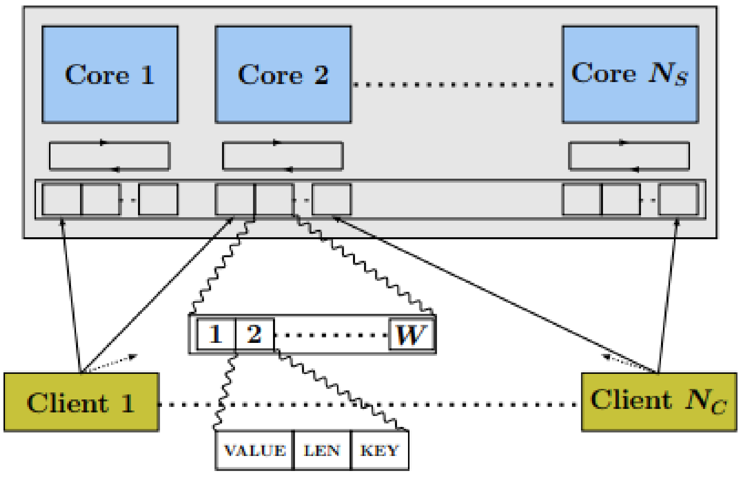

Figure 9: Herd's request region.

in terms of the size supported for key-value pairs. In a distributed cache scenario Herd shows limitation as it is implemented using multiple server processes on one machine only. That is a limitation because to use it in a distributed manner, it would be required to add a logical layer to synchronize all the nodes.

2.4.3 *FaRM*

FaRM [12] is a main memory distributed computing platform that exploits RDMA. The platform creates a shared address space, exposing the main memory of all nodes. Farm nodes can also execute application threads. It was designed this way because it would be wasteful not to use the CPU's power as it is not used for RDMA operations.

The platform uses RDMA reads to access remote data directly and implements a message passing primitive using RDMA writes. This primitive is a substitute for the send/recv verbs. The implemented primitive is based on a circular buffer (Figure 10) to implement a unidirectional channel. There is one buffer on each receiver per sender/receiver pair.

The receiver needs to detect new messages and for that purpose, unused portions of the buffer are zeroed. By polling the head position, it is possible to detect new messages because any non-zero value L symbolizes a new message with length L. The message buffer is zeroed and the head pointer advances after delivery to the application layer.

The sender uses RDMA to write to the circular buffer of the receiver. A message is sent by writing to the buffer tail and advancing the tail pointer. A sender keeps a local copy of the receiver's head pointer and never writes messages past that limit. The receiver makes processed space available to the sender by writing the current value of the head to the sender's copy using RDMA. This write operation is not executed for every message

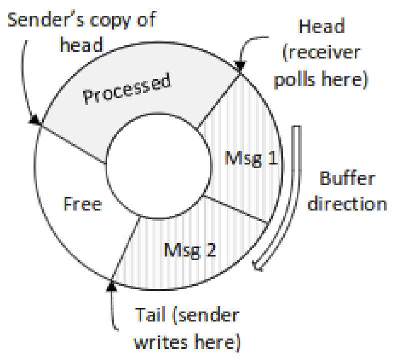

Figure 10: FaRM's circular buffer for messaging.

processed because that would lead to performance overhead. The operation is executed after half the buffer is processed.

FaRM's communication primitives are slower than accessing the main memory and, because of that, it is possible to execute threads on the node that contains the data. The programming model provided by FaRM is event-based and the operations that require polling take a handler as an argument, that is invoked when the operation is complete. Examples of such operations are: read and write. FaRM also provides operations for handling transactions to ensure consistency.

The advantage of Farm is that it provides a framework for developing applications based on RDMA in a cluster. Abstracting the actual RDMA operations makes the development easier. If the development is easier, developers can focus on complex problems without worrying about network-specific concerns.

2.5 DISCUSSION

After analyzing the caching solutions on analytical processing frameworks, it is noticeable the mechanisms can be improved. The cache solutions always have a particular feature missing. In Apache Spark, the management is non-existent and the user must define what is cached and what is evicted. In Apache Flink there is too much redundancy, making the cache very inefficient in terms of memory usage. In Dremio, the cache is not in memory and implementing a cache between Reflections and Executors could benefit performance.

Even though every studied framework includes a solution for caching, all of them could benefit from a more sophisticated one. As such, the development of a cache with memory management on top of an RDMA-capable network could improve these frameworks' caching mechanisms. It is important to note that Apache Flink's methods are more exposed to the development level. That particularity allows the integration of a new cache, like the one presented in this dissertation, more transparently.

3

USE CASE AND CHALLENGES

Distributed caching applications usually rely on TCP sockets to get and put values. It is obvious that when accessing other machines with cached data, there is an overhead added by the network communication. But, when the information is cached on the machine that needs to access it, network communications should provide only minimal impact on the access time. Knowing that distributed caching systems usually rely on TCP sockets, even when data is on the same machine, a test was conducted to understand the impact of this approach.

RDMA can provide several benefits in this context and, as such, an RDMA-based proof of concept cache was developed and tested to understand the challenges of this technology and conceive solutions. As discussed in the last chapter, a cache can be very useful in analytical environments. Figure 11 depicts the role and location of this cache in analytical environments.

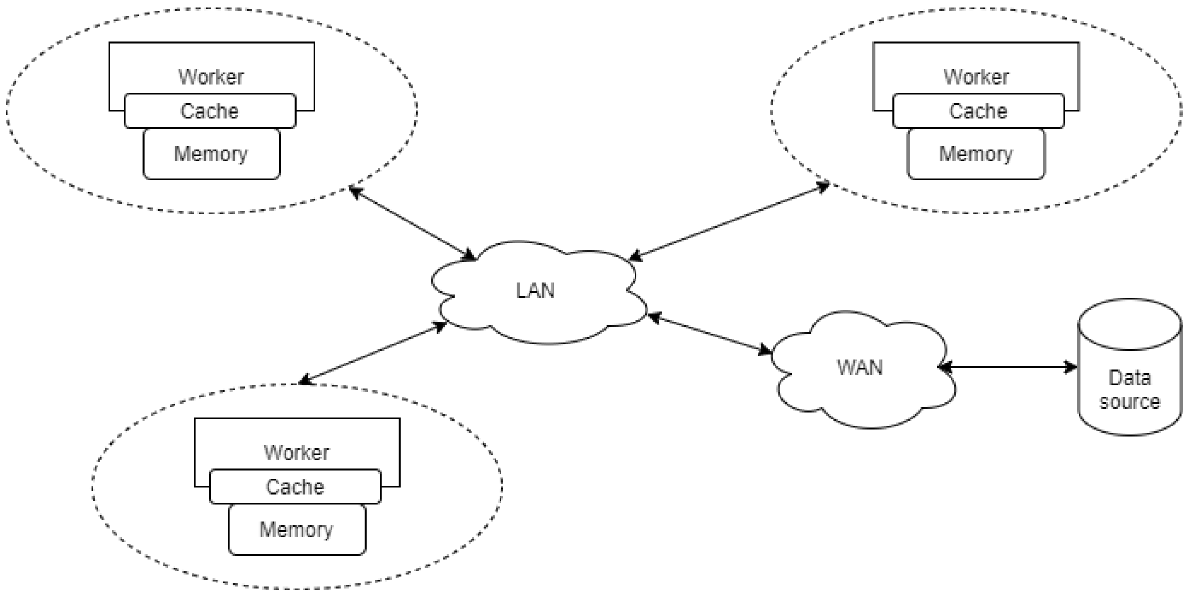

Figure 11: The cache in analytical environments.

In Figure 11, we have the cache module that is used in Worker nodes as a middleware. This way, the number of accesses to the Data source is reduced, which is possibly in a remote server as illustrated. These cache modules are connected through *Local Area Network (LAN)*, making the data in the cluster accessible to any worker.

3.1 USE-CASE

Two scenarios were considered. The purpose of the first scenario is to understand the impact of local accesses through a TCP connection. The second one is meant to supplement the development of a proof of concept. As already addressed, distributed caching systems usually rely on TCP sockets, even for local connections. As such, it is important to devise an experiment to help understand the impact of this communication strategy having a server on the same machine as the client. As there are no standard benchmarks to evaluate this trade-off, a micro-benchmark was developed for that purpose. The scenario designed for this experiment includes a micro-benchmark, deployed from a single machine with a Linux distribution installed and a Memcached server running. Memcached is a distributed key-value store, as detailed in the previous chapter. The system holds the following hardware:

- Intel ® Core ^{TM} i7-3537U @ 2GHz with 2 cores (4 logical cores with hyperthreading)
- 8GB of RAM
- 240 GB of storage (SSD)

In a Linux filesystem, the /tmp folder usually has the *Temporary Filesystem (tmpfs)* that keeps files in-memory (with swapping). This composition is represented in Figure 12, showing how the components interact with each other.

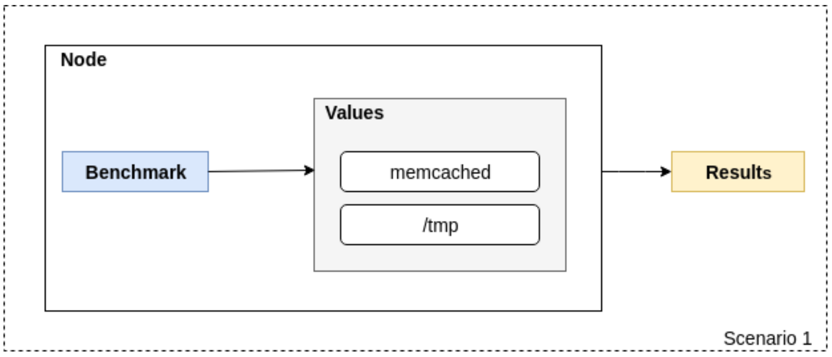

Figure 12: Local accesses experiment scenario.

The micro-benchmark developed for this experiment starts by generating 100 values and keys. The values generated are random strings that are 4 Megabytes long. These values are saved on a Memcached local instance and also in the /tmp folder of the machine. In the folder, keys correspond to the name of the file. After the values are created, all the values from both sources are retrieved by the benchmark. This process consists of getting the values 10 times and outputs the time it took to get and read them. The process is reproduced for Memcached and the /tmp folder. For Memcached, the values are retrieved using a socket. For the folder, the program uses memory mapping and, to make sure every byte is read, it runs through every byte in a cycle. The results produced by this benchmark are depicted in Table 2. It shows that having a socket connection has a very negative impact on performance when compared with accessing local memory. With the Memcached's socket connection, the execution time is three orders of magnitude higher.

Environment	Execution time(ms)
Memcached	1090.72
/tmp	3.99

Table 2: Local accesses overhead analysis.

The initial tests to assess the sockets' overhead motivate that designing a solution using RDMA mechanisms can provide a performance improvement. As an exercise, consider using the tmp directory as the data structure as in the initial tests. The problem with this is the fact that it is against the purpose of RDMA. The goal is to have a zero-copy system, and having the tmp folder as a data structure would imply constant copies from the memory region to the tmp and vice-versa. With all the memory copy, there would not be a substantial difference between using sockets and using RDMA.

As a first strategy, let's assume that all the nodes in the cache know where all the values are stored and, consequently, if they exist. For now, we will ignore how the clients choose which server to connect to retrieve a value for a given key. The focus is on how the data is stored and retrieved. The objective is to have an initial version of a cache supporting the usual operations: GET, PUT and DELETE.

Each server or node has a memory region that contains the data structure with all the keys and values. The server and client need three other memory regions, consisting of a send, receive, and data memory region. The send and receive memory regions are buffers used to send/receive requests, as the names suggest. These buffers are useful for two-sided operations where the client and server need to register the intention to send and receive. The data memory region is used by the client to perform a one-sided read when getting a value.

In this prototype, the usage is not abstracted, and it is necessary to initialize the server and clients explicitly on each node. Also, it is required to specify the server for storing and getting a key/value pair. Then, the clients have a public API for the PUT and GET operations. To get a value, the caller must specify the key, that must be a string. For the PUT operation, the caller must specify the key and value that is also a string in this prototype.

Comparing to the related work approaches, this is similar to Pilaf. The goal here is to use RDMA to read operations from the client without interrupting the remote processor. The Herd's approach indeed gets better results when comparing to Pilaf [19], as it takes advantage of the discoveries made about the efficiency of the write operation. The problem is that it is not trivial to use it for large values as Herd has a maximum size for each key-value pair, which is 1KB.

Clients know the memory region's address after connecting to the server and use that as a reference point. When a client needs to access a value, it uses the key to request it from the server that has it stored. When the server receives such a request, it returns the offset and the size that represent the value position in the memory region. The server returns this pair only when the key exists. In the current solution, to perform a PUT operation, a client sends the key/value pair to any server. But, all the nodes must know where any key/value is. For that reason, a deterministic algorithm or mapping service collects the server location.

Figure 13 depicts the prototype's architecture. It is worth noting that a server node can also be a client to other server nodes and vice-versa. This configures the expected pattern in an analytical context, where every node uses cached values across all nodes.

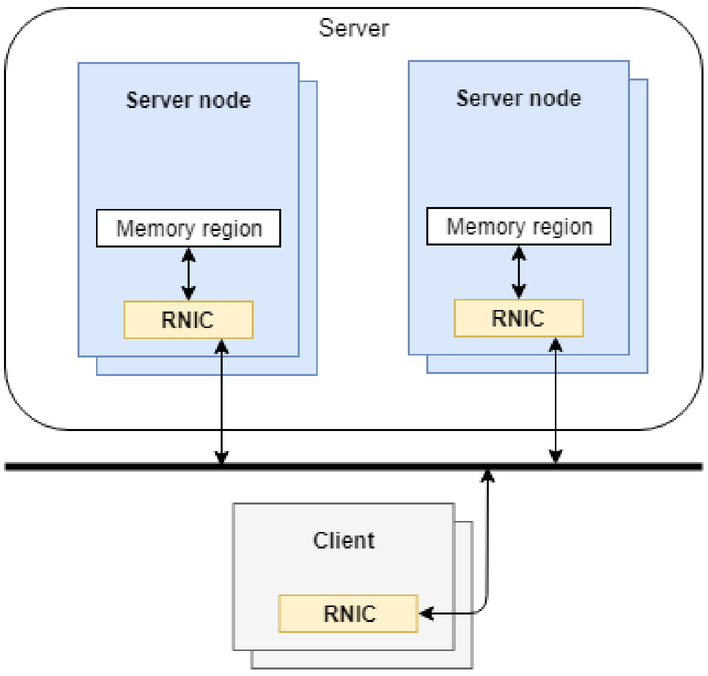

Figure 13: First approach architecture.

The second experiment provides the evaluation required to assess the proposed approach. It is meant to help understand the technology challenges and possible problems in the first approach.

The second experiment requires two nodes, and each node must have an RDMA-enabled network adapter. In this stage, we used software emulation, resorting to SoftiWarp, which is a software iWARP kernel driver and user library for Linux that implements the iWARP protocol suite in software, without requiring any dedicated RDMA hardware. SoftiWarp makes it possible for any machine to speak RDMA, as RDMA-capable network cards are expensive, and only available in particular nodes. It is costly for any of the standards described previously (IB, RoCE and iWARP).

In this experiment, the nodes are virtual machines with access to equivalent hardware resources:

- Intel Core i7 9xx @ 2GHz with 1 core
- 2GB of RAM
- 20 GB of HDD storage

The operating system installed in each one is Ubuntu 16.04 LTS with the 4.4.0-171-generic kernel version to support RDMA.

The first node contains a micro-benchmark very similar to the one developed for the first scenario. There are two main differences. The first difference is the fact that in this case, the Memcached server is running in another node. The second difference is the purpose. This experiment is intended to compare Memcached to the developed prototype. Therefore, the values are now fetched from remote servers and the /tmp folder has no role in the experiment. The servers run in the second node. This scenario is represented in Figure 14.

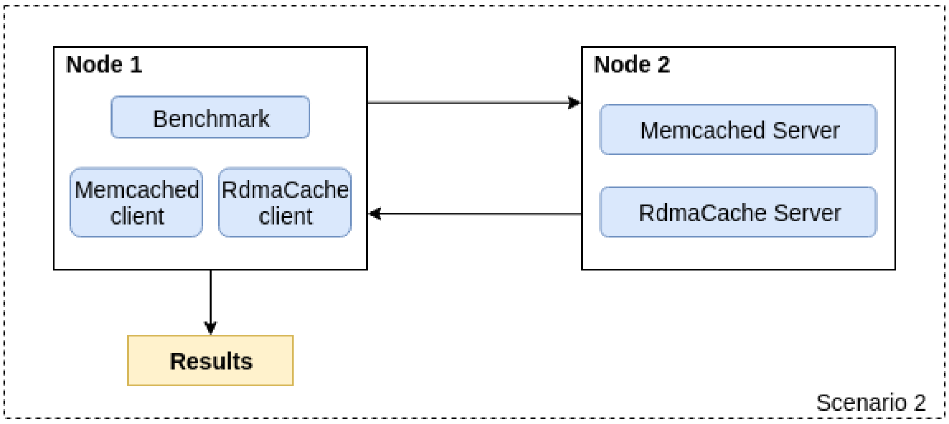

Figure 14: Second experiment scenario.

The results for this preliminary evaluation show a high impact on the execution time when using the prototype (Tables 3 and 4). The outcome of this experiment is due to the

Packet size	Execution Time (ms)
512 KB	626.17
1 MB	1110.55
2 MB	2137.83

Table 3: Prototype experiment - Memcached results.

Packet size	Execution Time (ms)
37 B	1475
185 B	1444
8 KB	13413

Table 4: Prototype experiment - RdmaCache results.

inadequate usage of the buffers that are registered as memory regions to perform two-sided RDMA operations, as described above. The strategy employed for these buffers is to register memory regions with a size that is capable of holding the biggest value supported by the cache. This approach is explained by the fact that the value is sent on the PUT operation, using RDMA send/recv. The length of the data can be defined before sending and that will avoid sending the complete buffer. But, as the memory registered for the buffer is larger, the memory mapping tables' performance is still affected.

This outcome is also influenced by inefficient buffer handling when reading and writing. In this prototype, the values and messages exchanged are always assumed to be a string. As such, the buffers are converted to a string for writing and reading, using a method that creates a char view of the buffer. This view shares the memory space but not the pointers (position and limit). With this approach, when writing and reading to the char view, the pointer of the original buffer can be different and leads to inconsistencies and data corruption.

These conclusions impact on the design of the middleware because it is imperative to rethink the memory region registration for send and receive buffers. Furthermore, it is necessary to improve the buffer handling by manipulating the buffers directly to make sure the position and limit pointers are correct and data is not corrupted. With this approach, the client can still fetch the value as a string instead of a char view of the buffer, for example, converting the byte array to string with a particular encoding.

3.2 DISCUSSION

One of the assumptions made on this prototype is that servers or nodes know where the value for a given key is stored and if it exists. One of the challenges to be addressed is finding a deterministic algorithm or service that returns the server that holds a given key.

Moreover, the eviction strategy for the cache is yet to be defined. An eviction strategy must be in place, as deciding what to do when the memory is full is not enough or when values are not accessed for a long time. When developing a caching system for an analytical environment, the complete dataset will not fit in memory. The memory is limited, and the cache must manage it efficiently to maximize cache hits.

After the second experiment, it is clear that the buffer handling needs improvements. The first step is to consider values as bytes and provide an option to get and put as a string but always converting from string to byte using the buffer directly and not the char view, because of the problems detected during the experiments. The second improvement is rethinking the memory regions registered for two-sided operations because it also had an impact on the collected results.

The cache presented in the next chapter is a middleware, and one of the main motivations for that is how challenging taking advantage of RDMA mechanisms is, due to its different protocol and flow of information when compared to other network protocols. Currently, the main obstacle in using RDMA is the fact that it is hard for a programmer to adopt RDMA technologies due to the different characteristics of the protocol and concepts. Thus, packaging the cache as a middleware that takes care of all the configuration and connections necessary is crucial, providing a simple PUT/GET API for the programmer while also ensuring all the performance produced by RDMA.

4 DISTRIBUTED CACHE

RDMA mechanisms have the potential to improve data transmission performance by using a zero-copy protocol. In the preliminary experiments, it is possible to realize that developing an application based on an RDMA-capable network communication is not simple.

The distributed cache abstracts the usage of RDMA for an application that requires a distributed cache with low latencies, high throughput, and low CPU usage. Using the distributed cache, the client application does not have to worry about the low-level intricacies of RDMA and buffer handling. The middleware manages the buffer handling taking advantage of RDMA features, and using an RDMA cache becomes as simple as a PUT and GET API.

The proposed distributed cache is implemented as a middleware layer, enabling client applications to create nodes and configure the available servers. When using the middleware, it is possible to configure the available memory in each node and also the maximum value size supported for data transmission.

4.1 SYSTEM ARCHITECTURE

The verbs API makes it possible to interact with the hardware and perform RDMA operations. In essence, verbs are an abstract description of the functionality that is provided for applications for using RDMA. In Linux, the verbs API is made available in the open-source user libraries developed by *OpenFabrics Enterprise Distribution (OFED)*.

The distributed cache uses the verbs API to execute RDMA operations. As the RDMA user libraries are developed in C, to use them in Java, it is necessary to use a *Java Native Interface (JNI)* layer. To achieve that, the *Direct Storage and Networking Interface (DiSNI)* library is used, which creates a thin JNI layer to bridge between Java and the RDMA user libraries.

The system architecture presented in Figure 15 reveals how the cache is packaged as a middleware to be used by external applications. This diagram shows the inner architecture of the middleware.

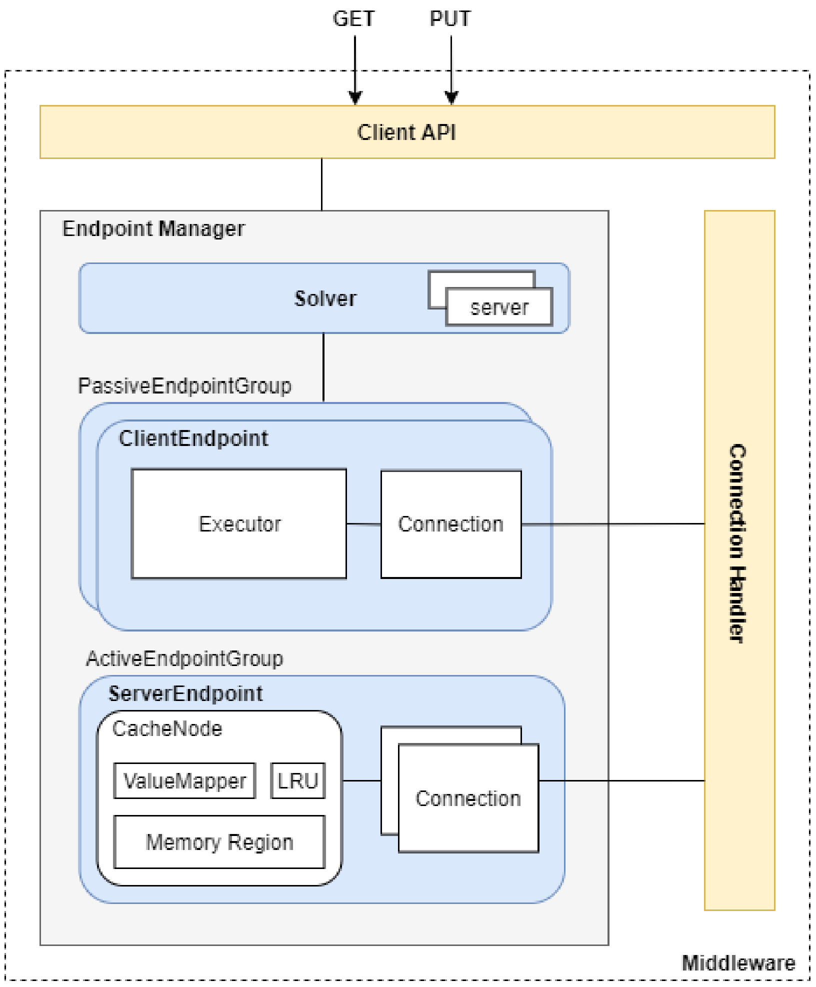

Figure 15: Middleware architecture.

Each node in a distributed environment creates an instance of the middleware and uses the Client API to initialize, configure, and interact with the cache. The API provided by the middleware includes two simple PUT and GET operations to store key/value pairs in the distributed cache. These operations support arbitrary bytes as the value and a string as the key.

The Client API acts as a proxy for the Endpoint Manager, which is responsible for managing all the components based on the provided configuration. This configuration defines the maximum value size, the node maximum memory, and the servers to connect to and will be detailed later in this chapter. The Endpoint Manager is composed of three major components: Client Endpoint, Server Endpoint, and Solver. The Client Endpoint and Server Endpoint components use the Connection Handler to establish connections and exchange data using RDMA. Briefly, when a request is submitted to the Client API, it is redirected to the EndpointManager. Then, the EndpointManager resorts to the Solver to map the key to the corresponding server. Knowing the server, it uses the corresponding ClientEndpoint to

GET or PUT, depending on the operation submitted to the API. The ClientEndpoint then marshalls the message in the buffers to send through the Connection Manager. On the other side, a ServerEndpoint will be listening and receives the request through the Connection Manager. In the case of a GET operation, the ServerEndpoint will fetch the location of the value from the CacheNode component and send it to the client to perform a one-sided read operation. For a PUT operation, the ServerEndpoint performs a one-sided read on the data buffer of the client, using the Connection Manager, and then stores it using the CacheNode.

The Connection Manager is the bridge between the distributed cache components and the underlying RDMA network. It is responsible for establishing and managing connections. As described above, the Client Endpoint and Server Endpoint components rely on it to use the RDMA network to transfer data.

The Solver component holds all the servers that are part of the cache, including the local one. When the Endpoint Manager needs to know which server to connect to, it resorts to the Solver to find out where a given key is stored, if it exists. The Solver applies a consistent hashing [20] technique and, based on the outcome, determines the server to store or get values. Consistent hashing reduces false cache misses when adding or removing servers, which is what would happen in mod-n hashing [15].

The Client Endpoint component holds the Executor that implements all the cache operations: GET, PUT, and DELETE. The Executor uses the connection to the server to perform RDMA operations to read and write data to a remote server. Based on the operation, the key-value creates the message to send to the server.

The Server Endpoint component includes all the connections to the clients and a Cache Node. The Cache Node is the module that handles the stored values in the current node. In this module, there is a ValueMapper that gets the value position based on a given key, and an LRU module, that is composed of several elements that make up the eviction strategy.

4.2 SPECIFICATION

This section introduces the information flows between servers and clients.

The GET operation flow is depicted in Figure 19. The first step is to determine which server holds the given key. After the Solver returns the correspondent server, the Client Endpoint that connects with that server marshalls the message to send and performs an RDMA send operation. The server then checks with the Cache Node if the key exists, and returns the necessary information for the client to perform an RDMA read, filling the data buffer with the value. If the key does not exist, the server returns zero, and then the client application decides what to do.

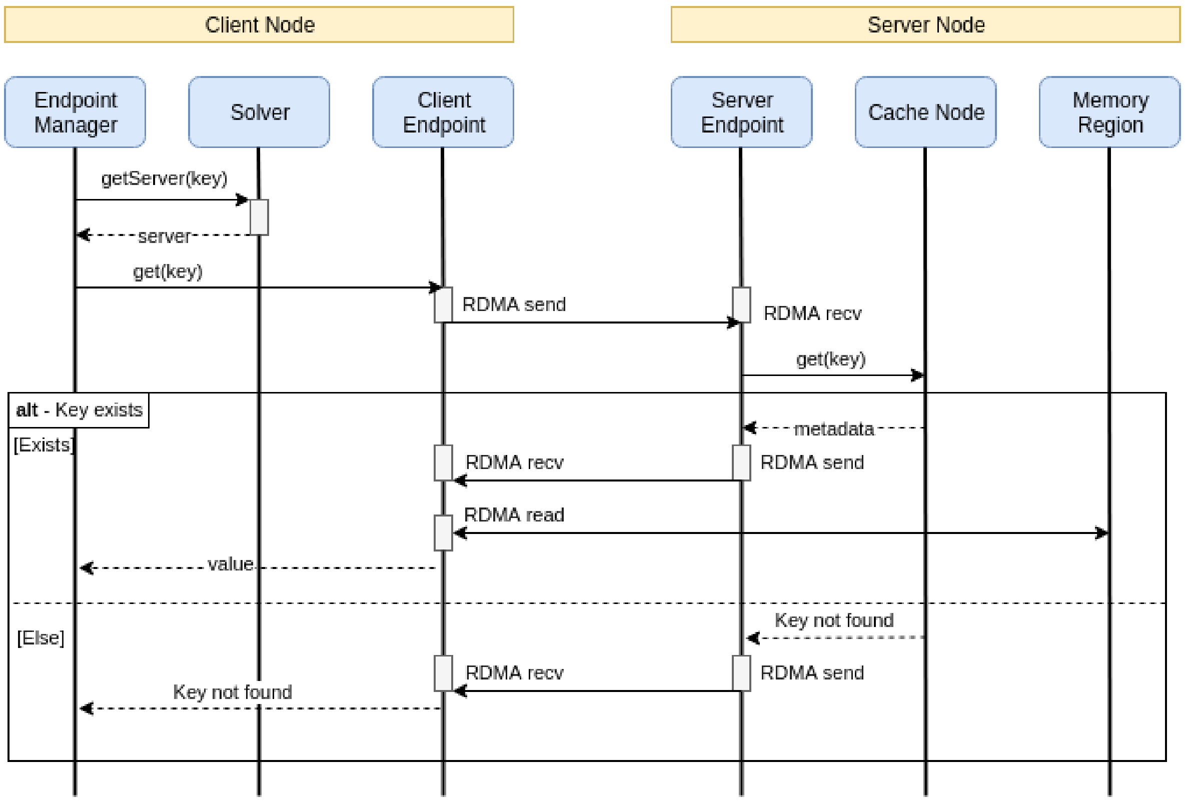

Figure 16: The GET operation flow.

As presented in Figure 20, when a PUT operation is requested through the Client API, the Endpoint Handler needs to resort to the Solver once more to figure out where the given key should be stored. Once the Solver returns the server, the Endpoint Manager proxies the work to the Client Endpoint that connects to that server. The message is built in the send buffer following the protocol to perform an RDMA send. Also, the value is kept in the data buffer for the server to read it later. The server uses the information in that message to perform an RDMA read operation and get the value from the client's data buffer. Then, the Cache Node writes the value to the memory region buffer after evaluating the available space. Finally, the server endpoint sends an acknowledgment to the client.

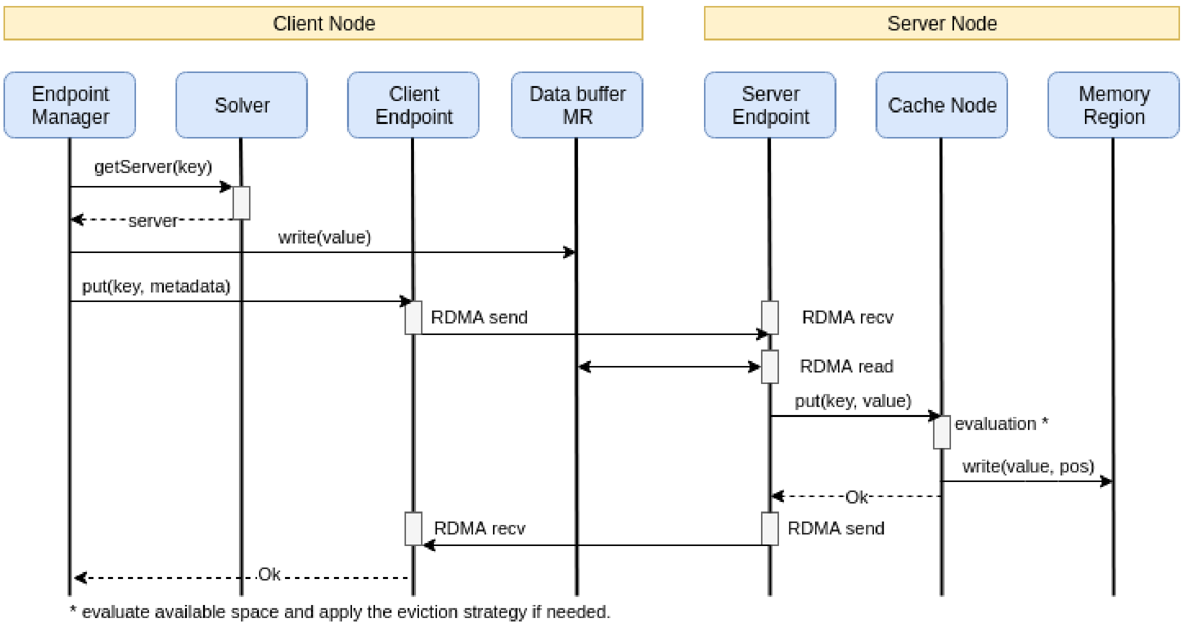

Figure 17: The PUT operation flow assuming success.

Finally, if a value needs to be explicitly deleted, the process is very similar to the PUT operation, the only difference is that there is no value for the server to get. The send buffer only contains the operation byte and the key. After receiving the request, the server simply deletes the value from the memory region and sends an acknowledgment.

4.2.1 *Assumptions*

The distributed cache excludes direct assumptions regarding fault tolerance of the proposed solution. That means that the cache does not have any redundancy and if the server that holds the value for a given key fails, getting that key will result in a cache miss. Usually, the value can be obtained from the data source and cached again. This does not preclude the introduction of fault-tolerance mechanisms through data replication, which we leave for future developments.

4.2.2 *Configuration*

When deploying the distributed cache, it is important to configure it based on the application requirements. There are three configurations, but only one of them is mandatory, as it is essential for the bootstrap process, which consists of connecting all the servers, ensuring that all servers have a connection with each other. The required configuration defines which servers the current node will connect to. The optional configurations are the maximum value size and the total node memory to allocate for cache storage.

- **Maximum value size** - The maximum value size influences the size of the data buffer in the Server Endpoint and Client Endpoint instances. If the cache is used for values bigger than the one that is configured in this parameter, it will result in errors. This configuration is optional and defaults to 2MB.
- **Total node memory** - The total node memory determines the allocated memory for the values in the Cache Node, registered as a memory region. In essence, it is the memory available to store values in the current node. This configuration is optional and defaults to 500MB.
- **Servers to connect** - This configuration is mandatory because it must include, at least, the local server. The nodes need to start execution in a specific order, depending on this configuration. The order is important because when a node creates a connection to another one, the destination server creates a connection back. As such, the first node to start execution is configured with the local server only, the second one is configured with the local and the first one and so on.

4.3 IMPLEMENTATION

After understanding the system architecture, it is essential to comprehend how the cache implements its core features.

The Connection Handler relies on DiSNI, which is a library for accessing the verbs API using Java. The DiSNI library uses a thin JNI layer to bridge between Java and the RDMA user libraries. To avoid performance issues due to the complex parameters and arrays for the RDMA calls, this library implements a concept called the *Stateful Verbs Method (SVM)*. With SVM, the JNI serialization state is cached per verb call to be reused many times. The Connection Handler uses the DiSNI library to manage the network connections. The DiSNI library follows a Group per Endpoint model which is based on three interfaces:

- **DisniServerEndpoint** - a listening server waiting for new connections, with methods to bind to a specific port and to accept new connections.
- **DisniEndpoint** - a connection to a remote or local resource (RDMA in this case). It offers non-blocking methods to read or write to the resource.
- **DisniGroup** - a container and factory for client and server endpoints.

There are two types of EndpointGroups in the RDMA API available in DiSNI:

- **RdmaActiveEndpointGroup** - Actively processes network events caused by RDMA operations.
- **RdmaPassiveEndpointGroup** - Provides a polling interface that allows the application to directly get the events from the completion queue.

These two groups have different advantages and, as such, work best in different contexts. The passive group provides lower latency but when using multiple threads for the same connection, there could be a contention problem. On the other hand, the active group is robust with a large number of threads but causes higher latencies. Passive endpoints are usually better when the application knows when messages will be received. In most cases, clients should use passive endpoints and servers active endpoints.

The Solver component uses consistent hashing to map a given key to the corresponding server. To implement consistent hashing, we have a ring of values, and the servers are placed on that ring based on their hash. Then, the hash of a key is also placed in the ring to figure out the closest server, and that is the chosen server. Still, consistent hashing can have a non-uniform distribution of data if we directly map the server hash to a location in the circle, mainly when there are few servers. To improve the distribution, we can create virtual nodes, that is, adding multiple entries for the same server in the circle. That way, the servers are more dispersed, and the keys will be better distributed. Using a TreeMap as a structure, here is the sample code to add a server to the ring with several replicas:

```
for (int i = 0; i < numberOfReplicas; i++) {
   byte[] data = StringUtils.getUtf8Data(node.toString() + i);
   circle.put(hashFunction.hash(data), node);
 }
```

To interact with the server, the client uses the buffers depicted in Figure 16 (the box sizes are not to scale). All the buffers are registered as memory regions to be accessible to the RNIC. The send and recv buffers are used to perform RDMA two-sided operations (send and recv) and the data buffer is used for one-sided ones. The Server Endpoint uses equivalent buffers because, for the two-sided operations, there must be registered buffers with the same size in both ends. The creation and management of these buffers is a crucial part of the cache, as the main goal is to take advantage of RDMA technology, providing a simple interface for other programs as a middleware.

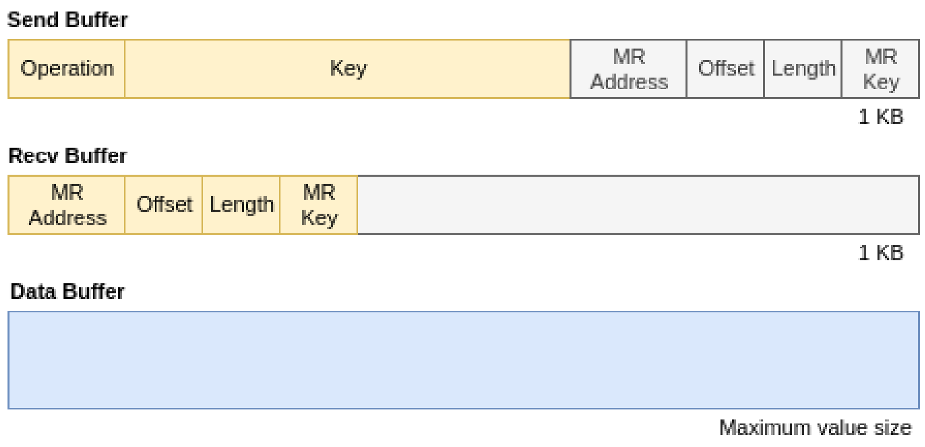

Figure 18: Client endpoint's buffers.

In the send buffer, the first byte is used to specify which cache operation is being issued. To execute a GET operation, the byte corresponds to the character ”g”, and for a PUT operation it is ”p”. The operation byte is always followed by the key, that must be a string, and other optional fields. The optional fields are used when a PUT is being executed. To avoid the issues encountered when developing the proof of concept, namely, the buffer size impact on the performance of send/recv verbs, the send and recv buffers are kept short and the values are never sent in the operation message. Instead, the PUT operation includes the memory region information required for the server to perform an RDMA read and get the value directly. This strategy circumvents using long buffers for very short messages.

The recv buffer is used to execute an RDMA recv when waiting for the server to respond. Although Figure 16 depicts the recv buffer as having all those parameters, that is only the case when the operation performed is a GET. In that case, the server sends the memory region's address and key, the offset of the value, and its length. With this information, the client can read the value directly by executing an RDMA read. Using the RDMA read operation, it is possible to obtain the value directly, with zero-copy among buffers in the remote server. In the Server Endpoint, the recv buffer is also used to execute an RDMA recv, but in this case, the server always registers a recv after every handled request, because it is always listening for requests.

The data buffer is used when performing an RDMA read. That is why its size corresponds to the maximum value supported by the cache. As described above, an RDMA read is performed by the server when a PUT operation is requested. The client resorts to its data buffer when the GET operation is executed. In both cases, the data buffer gets filled with the specified remote data.

Lastly, the Memory Region buffer depicted in Figure 17 is dedicated to storing the actual values and is only allocated in the Server Endpoint. The values are stored contiguously and a data structure is maintained to identify the available memory space. When a value is deleted, the space occupied by that value will become available and, sometimes, that creates noncontiguous free space. Likewise, it is possible to know where the available spaces are. To do this, when the server endpoint is initialized, it creates a HashMap of free spaces with only one value. That value corresponds to the entire memory region because, at this point, all the space is available. In Figure 17 there is an example of the state of the memory region when a value is explicitly deleted.

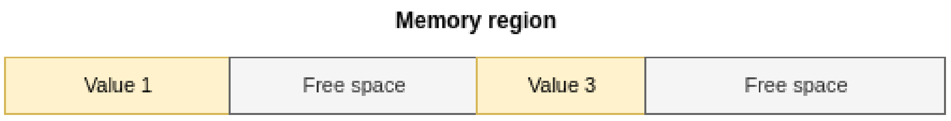

Figure 19: Server's memory region.

The CacheNode component implements an LRU eviction strategy because it only makes sense to cache something when it is being used multiple times in a short period. If a value is not used for a long time and is the least recently used, then it can be beneficial to delete it and leave room for a value that, possibly, will be used multiple times. To achieve the LRU eviction strategy, the Cache Node keeps two important pointers. The start pointer leads to the most recently used value and the end pointer corresponds to the least recently used one. As it is depicted in Figure 18, it is also necessary to keep a double-linked list for the values. These links help to update the start and end pointers when a value is evicted, deleted, added or retrieved. For example, when a value is evicted, the left pointer is used to determine the new least recently used value.

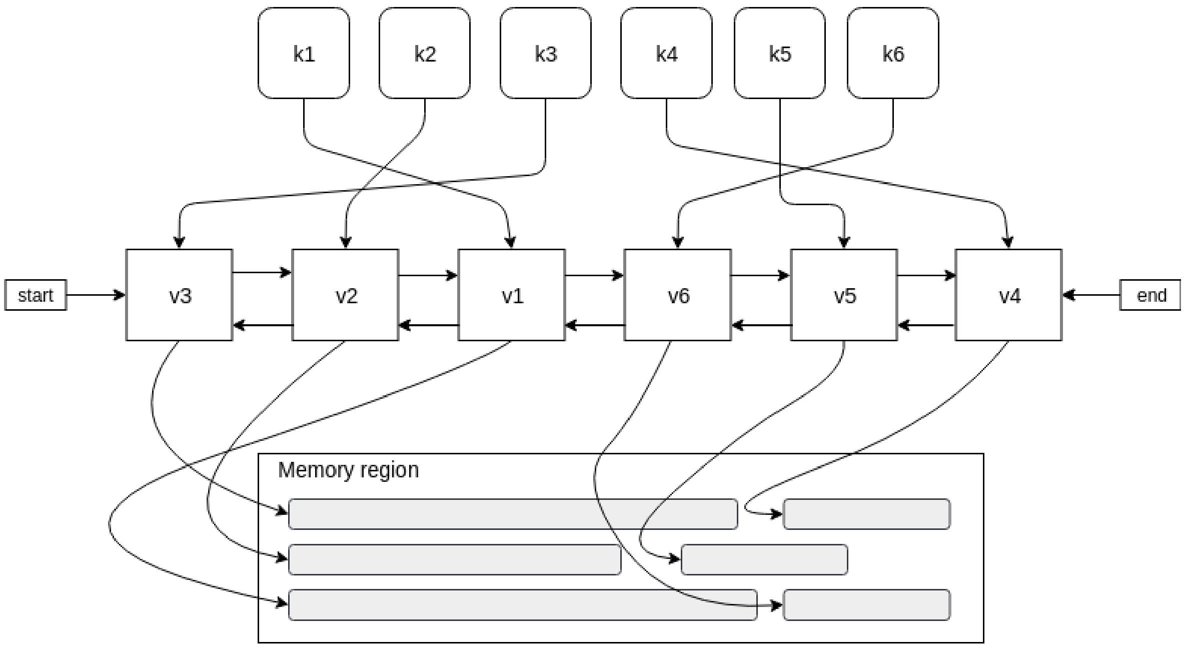

Figure 20: LRU cache implementation using an HashMap and double linked list.

5 SYSTEM ANALYSIS AND RESULTS

The distributed cache is evaluated considering a micro-benchmark specially tailored for this purpose. Two evaluation campaigns were then carried out, executing the same job for Memcached and the distributed cache introduced in this dissertation. The main goal is to examine to what extent the developed cache delivers performance wise, in comparison to one of the most used distributed caches, Memcached.

5.1 EXPERIMENTAL SETTING

The performance analysis relies on two micro-benchmarks, to compare the distributed cache to another cache with very similar environments, making sure no external factors influence the results. Standard benchmarks are not suitable as, generally, they target high level processing systems with abstractions at the application level. As the cache is completely agnostic to such high level abstractions, these benchmarks are not adequate for this environment.

The main purpose is to assess the performance when getting values from the cache by running the micro-benchmark in multiple configuration settings. As the main goal of a cache is to get values that were already fetched from the data source, it is important to understand to which point an RDMA-based cache can improve the latency and throughput during the GET operation. The micro-benchmarks are split into two stages. The first stage consists of populating the remote node with values of a given size or sizes. The second stage executes the GET operation continuously for a given period. During the execution, the benchmark gets the values from the remote server and registers the response time. In the end, it outputs the average response time and also the throughput (operations per second, and MBits per second).

The evaluation is deployed over two setups. The first setup is based on two nodes running in an emulated and virtualized environment. In the emulated environment, there is no RDMA-enabled hardware and software emulation is used resorting to Softiwarp. Moreover, there is only one physical machine and the nodes are virtual machines. The second setup is hosted in the Minho Advanced Computing Center and is also comprised of two nodes but, in this case, the network is based on Infiniband, which means RDMA operations can be executed without software emulation. Also, both nodes in this setup correspond to physical machines.

The first batch of evaluations is performed in the first setup and the physical machine's hardware is:

- Intel ® Core ™ i7-3537U @ 2GHz with 2 cores (4 logical cores with hyperthreading), 8GB of RAM and 240 GB of storage (SSD)

In this setup, two virtual machines are created and configured with 1GB of RAM and one CPU core each, with 20GB of SSD storage available for each virtual machine. The experiments performed in this environment are useful to determine the viability of the buffer strategy in the distributed cache, that will be referred to as RDMA Cache. As such, RDMA hardware is not mandatory and both nodes are setup with Softiwarp, to emulate RDMA-enabled hardware. In this environment, the comparison is performed between the RDMA Cache, Memcached and an early version of the RDMA Cache. Both nodes are setup with Memcached, RDMA Cache's final version, and RDMA Cache with the early buffer handling strategy.

The second batch of evaluations is performed in the second setup, which holds the following hardware:

- 8-core Intel Xeon E5-2680 @ 2.7 GHz, 32GB of RAM and 115 TB in a shared Lustre parallel filesystem.
- 8-core Intel Xeon E5-2680 @ 2.7 GHz, 32GB of RAM and 115 TB in a shared Lustre parallel filesystem.

The nodes are interconnected with an Infiniband FDR network (54 Gbps).

In this setup, both physical nodes are setup with Memcached and the RDMA cache. The goal of the evaluations performed in this setup is to assess the performance difference between using Memcached and the RDMA cache in an RDMA-capable environment. Softiwarp is no longer required, as this setup's network is based on Infiniband hardware.

5.2 CONFIGURATION

It is important to test the cache with different configurations to determine their impact. The configurations manipulated during the experiments are divided into two groups. The first group is the cache's configuration. For the RDMA Cache, the configuration consists of manipulating following values:

- Maximum value size.

- Available memory for values.
- Available servers.

For Memcached, the configuration is very similar, but it is necessary to provide the location of every server explicitly. The second group is the benchmark configuration that allows configuring the following parameters:

- Value size of the populated values.
- Benchmark running time.

5.3 RESULTS

This section focuses on analyzing the results produced by running the micro-benchmarks. Firstly, the RDMA Cache buffer strategy is analyzed to understand the impact of the decisions made in the final version when compared to an early version. The early version considered has all the buffer handling problems encountered in the proof of concept solved. After asserting the viability of the RDMA cache buffer strategy, the analysis is focused on comparing it to Memcached, which is one of the most used distributed caches.

5.3.1 *Emulated and Virtualized environment*

The first experiments of the micro-benchmarks are performed in the emulated environment. These results provided important knowledge for a more efficient implementation of the RDMA cache. The experiments are performed with two versions of the RDMA cache. The first version shares the same buffer strategy with the proof of concept but the buffer handling issues are resolved. The second version corresponds to the final version, with short buffers for RDMA send/recv messages and a data buffer for reading remote values, as described in the last chapter.

The one-sided experiment is the first evaluation performed in this environment, which is conducted by setting up the benchmarks to run in one of the nodes. As such, the values are only being retrieved in one way, which means that one of the nodes is acting only as a server and the other as a client. The benchmark is set up with different value sizes and always runs for 30 seconds. As the value size changes, the RDMA cache's configuration also changes accordingly. The server instances configuration is always the same, as there are only two instances for all benchmark runs. The value sizes considered in this experiment range from 16KB to 2MB to understand how the size variation impacts the performance. For small value sizes, the minimum is 16KB because we need small values that still can store some meaningful information. The maximum value size is only 2MB due to the context of analytical processing, as the data is, often, processed in shards.

The values represented in Table 5 correspond to the results obtained when running the benchmark against the RDMA cache with the first buffer method. As it is possible to notice, solving the buffer handling issues delivers results much closer to what was expected.

Packet size	Throughput (op/s)	Avg Response Time (ms)	Throughput (MBits/s)
16 KB	803.34	1.25	105.29
32 KB	450	2.23	117.96
64 KB	393.34	2.56	206.22
128 KB	265	3.81	277.87
256 KB	180	5.60	377.49
512 KB	103.33	9.85	433.39
1 MB	68.34	14.95	573.28
2 MB	33.33	30.11	559.18

Table 5: One sided results for the RdmaCache's first method.

The values depicted in Table 6 were collected by running the benchmark against Memcached. It is still noticeable that Memcached has better performance in this environment, but it is imperative to note that this is an emulated environment and there is no RDMA-enabled hardware.

Packet size	Throughput (op/s)	Avg Response Time (ms)	Throughput (MBits/s)
16 KB	881.44	1.13	115.53
32 KB	872.3	1.15	228.67
64 KB	865.46	1.16	453.75
128 KB	723.85	1.38	759.01
256 KB	450.31	2.22	944.37
512 KB	287.54	3.48	1206.03
1 MB	198.56	5.04	1665.64
2 MB	144.78	6.91	2429

Table 6: One sided results for Memcached.

The values described in Table 7 correspond to the results collected when running the benchmark against the RDMA cache's final version. The results provide better response times and throughput in comparison to the first version, especially as the value size increases. It is possible to infer that the new buffer strategy delivers better performance.

Packet size	Throughput (op/s)	Avg Response Time (ms)	Throughput (MBits/s)
16 KB	652.67	1.54	85.54
32 KB	625.33	1.60	163.93
64 KB	596.67	1.68	312.83
128 KB	462.67	2.17	485.14
256 KB	295.33	3.40	619.35
512 KB	178.67	5.67	749.39
1 MB	106.67	9.53	894.81
2 MB	57.50	17.67	964.69

Table 7: One sided results for the RdmaCache's new method.

To support the analysis of the results, the data is represented in two-axis graphs. Figure 21 depicts the throughput results for all the tests performed in this experiment and Figure 22 depicts the response time results for the same tests. Both figures provide a visualization of the performance difference between the caches.

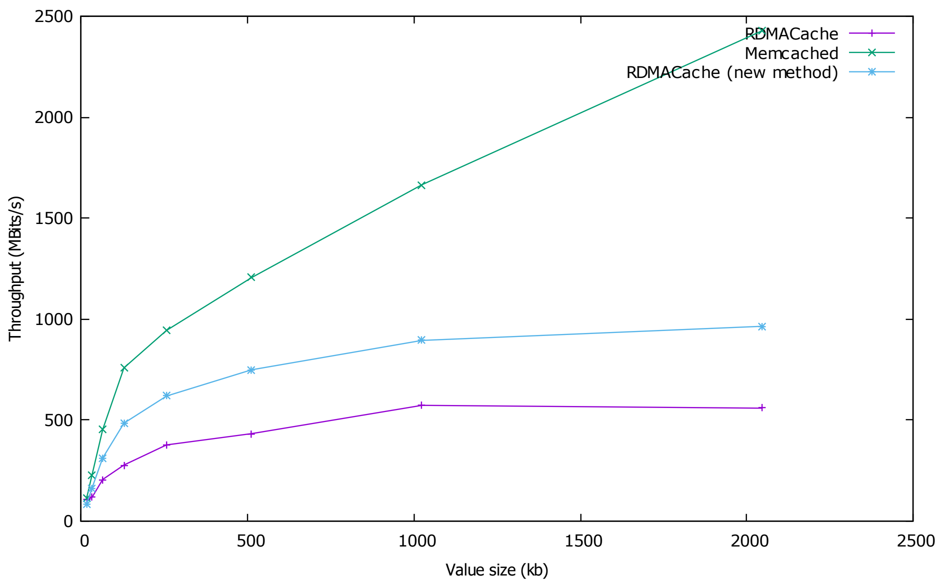

Figure 21: Throughput results for the one sided benchmark.

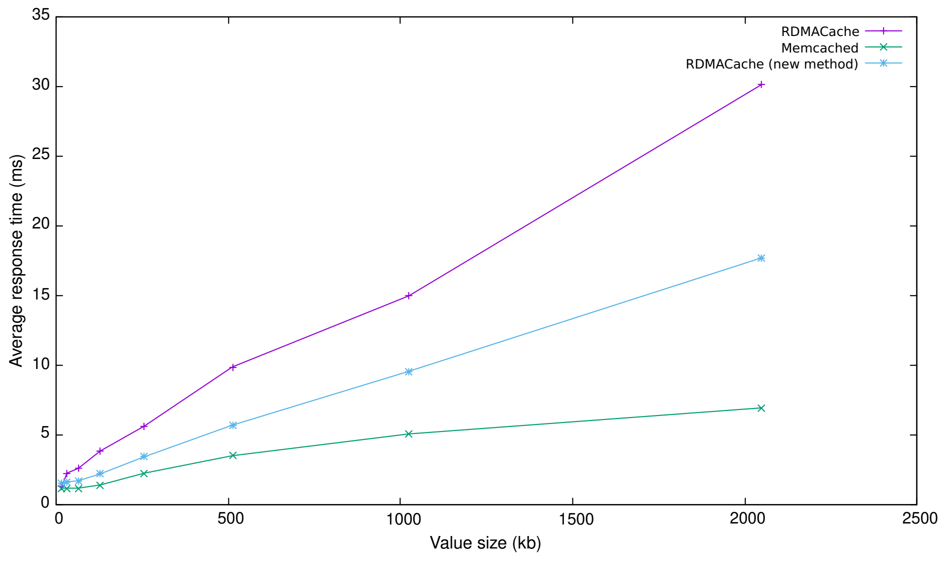

Figure 22: Response Time results for the one sided benchmark.

The two-sided experiments were performed in the same environment, but are more realistic when comparing to real-world applications. In a distributed computing environment, all the nodes will need to access the cache in each others' memory. In this experiment, the benchmark is running in both nodes. The first node is always configured to run the benchmark for 30 seconds, where the results will be recorded. The second node is configured with a longer running time to ensure the other node performs the benchmark while data is flowing in the other direction for the whole execution.

The results obtained for the RDMA cache's first method are represented in Table 8. The values show that even running in a two-sided setup, the performance is much better than what was accomplished in the proof of concept. This confirms that the buffer handling problems were solved.

Packet size	Throughput (op/s)	Avg Response Time (ms)	Throughput (MBits/s)
16 KB	838.66	1.19	109.92
32 KB	683.33	1.48	179.13
64 KB	463.33	2.17	242.92
128 KB	242.67	4.17	254.46
256 KB	145.33	6.95	304.78
512 KB	75.53	13.60	316.79
1 MB	42.11	24.12	353.24
2 MB	17.97	56.46	301.49

Table 8: Two sided results for the RdmaCache's first method.

Table 9 represents the results obtained by running the benchmark against Memcached. These results serve as a reference to compare with the two versions of RDMA Cache.

Packet size	Throughput (op/s)	Avg Response Time (ms)	Throughput (MBits/s)
16 KB	1942.16	0.51	254.56
32 KB	1854.70	0.54	486.19
64 KB	1641.67	0.61	860.71
128 KB	1054.08	0.95	1105.28
256 KB	596.18	1.68	1250.28
512 KB	329.63	3.04	1382.57
1 MB	160.01	6.28	1342.26
2 MB	86.44	11.62	1450.22

Table 9: Two sided results for Memcached.

The values presented in Table 10 correspond to the output of the benchmark when running against the RDMA cache's final version. It is possible to notice a big improvement when compared to the first version of buffer handling. These results are much closer to the Memcached ones, even though the benchmarks were performed in an emulated environment.

Packet size	Throughput (op/s)	Avg Response Time (ms)	Throughput (MBits/s)
16 KB	1144.67	0.87	150.03
32 KB	914.67	1.09	239.78
64 KB	689.33	1.45	361.41
128 KB	467.73	2.14	490.45
256 KB	218	4.54	457.18
512 KB	135.33	7.48	567.62
1 MB	72.67	14.03	609.60
2 MB	34.21	29.67	573.95

Table 10: Two sided results for the RdmaCache's new method.

To support the comparison of the values presented in the above tables, the results are presented in two 2-axis graphs. Figure 23 depicts the throughput graph and Figure 24 represents the response time graph.

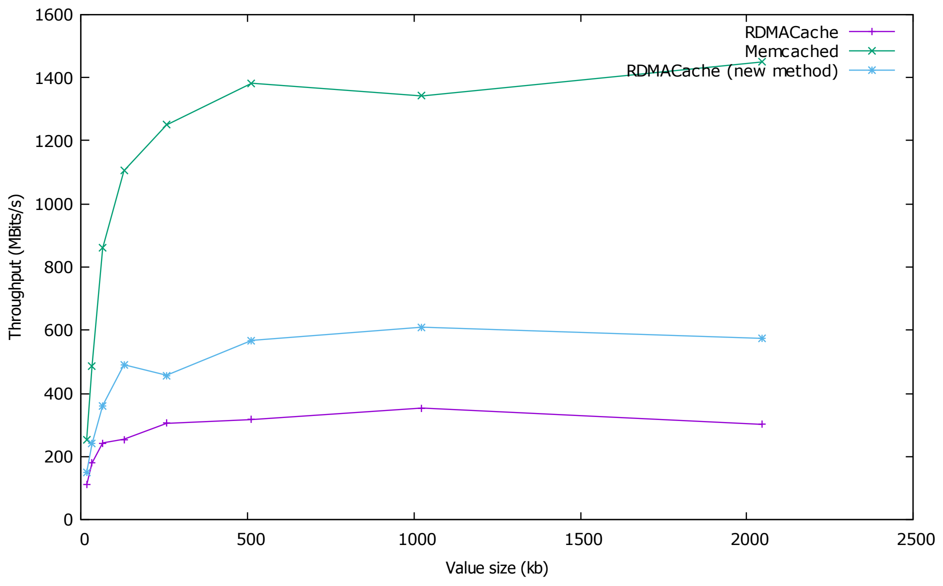

Figure 23: Throughput results for the two sided benchmark.

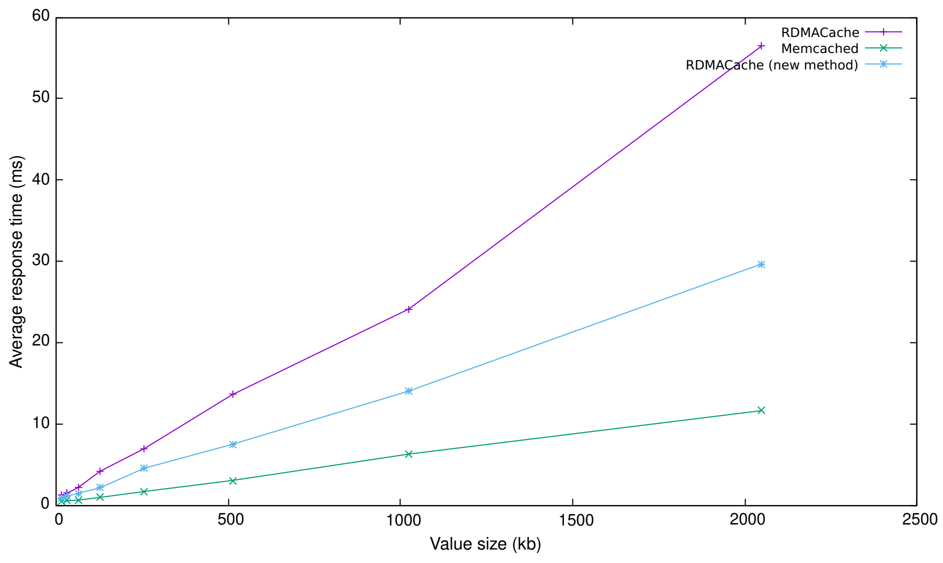

Figure 24: Response time results for the two sided benchmark.

The emulated environment provides valuable information to determine that the new strategy for buffer handling provides better performance. In the first approach, send/recv buffers were as big as the maximum value size required it. Even though only the filled part of the buffer is transmitted, the addressing tables in the RNIC are larger and, as such, the performance is worse. In the final version's approach, using short send/recv buffers and one data buffer whose size corresponds to the maximum value size solves this problem, as proven by the results presented by the benchmarks presented in this section.

5.3.2 *Infiniband environment*

For the Infiniband environment, all the experiments are two-sided as it is the closest to a real-world example and it does not make sense to consider one side for these tests. This environment provides important data as it holds the required RDMA hardware for the RDMA cache to achieve its potential. The first batch of benchmarks determined that the RDMA cache's final version is better in performance due to the new buffer strategy. Thus, in this phase, we compare the final version of the RDMA cache with Memcached.

Table 11 depicts the benchmark results when running against Memcached. In this setup, it is possible to observe that the results improved compared to the first setup even though there are two separate machines. The reason for that is better hardware and a fast network connection connecting the nodes.

Packet size	Throughput (op/s)	Avg Response Time (ms)	Throughput (MBits/s)
8 KB	5198.50	0.19	340.69
16 KB	3691.96	0.27	483.91
32 KB	2423.92	0.42	635.42
64 KB	1373.95	0.73	720.35
128 KB	754.74	1.33	791.40
256 KB	405.21	2.47	849.79
512 KB	214.37	4.66	899.13
1 MB	111.48	8.97	935.16
2 MB	55.81	17.91	936.34

Table 11: Infiniband - results for memcached.

Table 12 holds the results for the benchmark when running against the RDMA cache. The throughput and response times are significantly better than the Memcached ones. As predicted, accessing remote memory directly without involving the remote CPU improves the performance. In this case, we can notice roughly 2.7 times better results for small values and 3.9 times better for bigger values, which means that RDMA cache is better and also scales better.

Packet size	Throughput (op/s)	Avg Response Time (ms)	Throughput (MBits/s)
8 KB	13846.11	0.07	907.42
16 KB	11495	0.09	1506.67
32 KB	7645.56	0.13	2004.24
64 KB	4991.11	0.19	2616.78
128 KB	2970.56	0.34	3114.86
256 KB	1612.22	0.62	3381.07
512 KB	819.99	1.22	3439.29
1 MB	430.56	2.33	3611.79
2 MB	218.89	4.61	3672.36

Table 12: Infiniband - results for RdmaCache.

The results are presented in 2-axis graphs, in Figures 25 and 26. Figure 25 has the results for the throughput comparison and Figure 26 for the response time comparison.

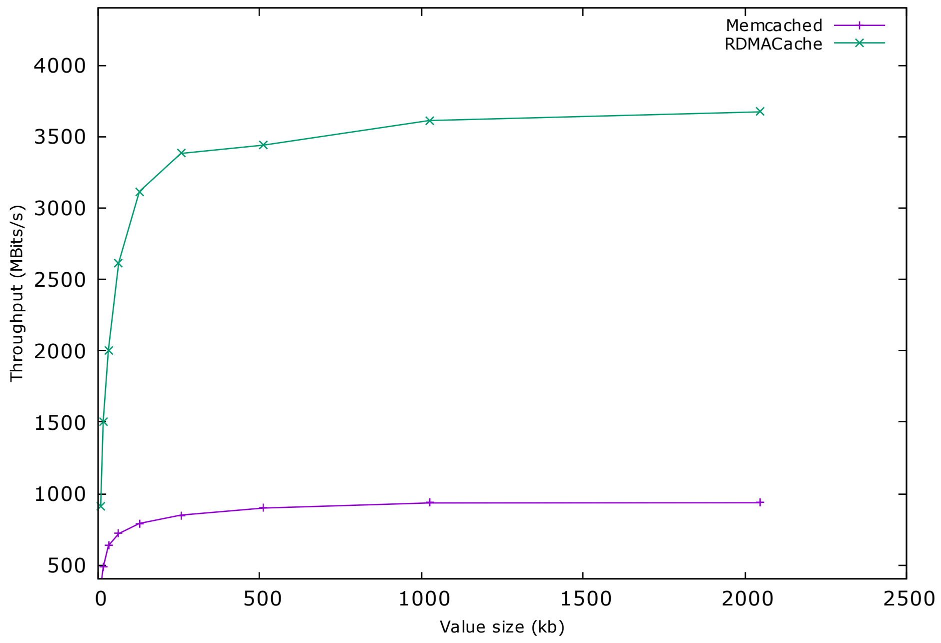

Figure 25: Infiniband environment - throughput.

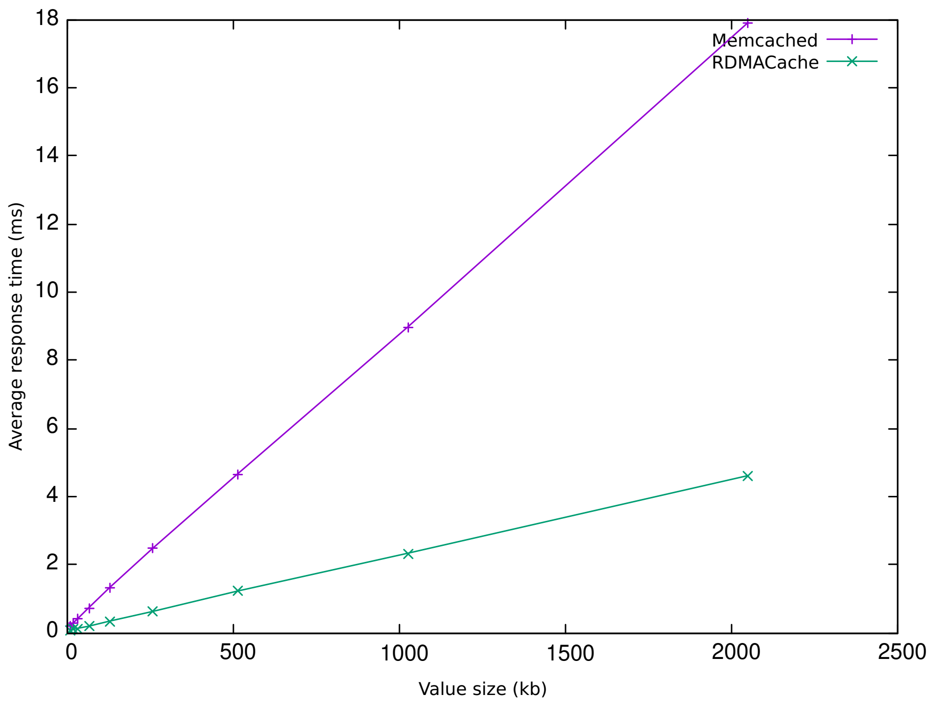

Figure 26: Infiniband environment - average response time.

The RDMA cache provides better performance, as predicted, due to the zero-copy protocol used. As the RDMA Cache uses RDMA operations to perform the heaviest data transmission, which happens when transferring the values, the latency and throughput improve significantly.

Finally, to evaluate if a variable-sized value collection impacts the performance, an experiment with random value sizes was conducted. The value sizes used for this experiment were: 16KB, 32KB, 64KB, and 128KB with a uniform distribution. Also, in this case, the benchmarks run for 30 seconds and 10 minutes to understand if longer tasks affect the performance.

Table 13 represents the results obtained when running the benchmark on Memcached. Comparing these results to the results obtained when executing the benchmark with values of invariable size, the results reside closer to the higher value sizes. Therefore, having different sized values does impact Memcached performance. This conclusion is based on the assumption that the values for the response times should average 0.42 ms, which is the 32KB's result in the last experiment. The results are compared to the 32KB's results because it is the average size of the values in this experiment. Furthermore, the 30-second run and 10-minute run resulted in very similar results in terms of response time and throughput. The slight difference is caused by a variation in the sizes of the values generation and, as such, the running time does not impact performance, assuming the same conditions.

Execution time (s)	Throughput (op/s)	Average Response Time (ms)
30	1395.81	0.72
600	1371.69	0.73

Table 13: Infiniband - results for Memcached with variable sizes and execution times.

Table 14 outlines the results obtained when running the benchmark against the RDMA Cache. Comparing these results to the results obtained when executing the benchmark with invariable value sizes, the results are close to the average value, considering the range of value sizes used in this experiment. Thus, using different sized values in the RDMA cache does not impact the performance. As a value set of variable-sized values impacts the Memcached performance and not the RDMA Cache's, the difference between the two caches in terms of response time and throughput increases when compared to the previous scenarios. The response time achieved with the RDMA Cache is roughly 5 times lower and the throughput is about 5 times higher. Running the benchmark with a longer execution time did not affect the results. The slight difference depicted in the table can be explained, again, by the variation of sizes in the value generation.

Execution time (s)	Throughput (op/s)	Average Response Time (ms)
30	6985.33	0.14
600	7609.33	0.13

Table 14: Infiniband - results for RdmaCache with variable sizes and execution times.

5.4 DISCUSSION

The first step in the evaluation process determines if the buffer handling decisions addressed in chapter 4 were impactful. Considering the tests in the emulated environment, it is clear that the new approach is better when compared to the first version of the RDMA Cache, and the results are very conclusive. In the same environment, the evaluation assesses if the new buffers, registered as memory regions for data transfer and message exchanging, result in a more efficient protocol.

As outlined in chapter 4, the size of the buffers changed when compared to the proof of concept in chapter 3. The new sizes of the send and receive buffers result in better performance, by relying on a data buffer to transfer the values from clients to servers. Most of the improvements are due to the relief of the RNIC memory tables by relying on smaller buffers for message exchanging. This is true because, even when not using the whole buffer, the performance is still affected. In the final version, the value is not sent inline with the put request and, instead, is read by the server using one-sided RDMA. The new approach may seem counter-intuitive, because of the extra round-trip, but the results prove that the extra round-trip is worth and improves performance. Performance is only guaranteed when all the servers are in the same network, as is usual in analytical environments.

As one of the main goals, the final version of the cache is compared to a real-world distributed cache, to understand if using RDMA mechanisms improves the performance when RDMA-capable hardware is available. It is important to note that Memcached and the presented cache both manipulate raw data, which is the main reason that it was the choice for this comparison. The results in the Infiniband environment show that the RDMA-based cache is more performant. These improvements are mainly due to RDMA being a zero-copy protocol, as discussed previously, bypassing all the usual network stack in the operating system.

Finally, the solution provides a simple API for the programmer while achieving substantial performance improvements. We can conclude that the cache provides better performance when compared to Memcached, achieving 3-4 times better results in terms of throughput and response time.

6 CONCLUSION

6.1 CONCLUSION

This dissertation is based on the need to process columnar data in an analytical environment, maximizing performance. Developing a distributed cache relying on RDMA mechanisms was set as the main goal due to its zero-copy and high throughput characteristics.

As RDMA mechanisms require understanding new and complex concepts, one of the main goals of the presented thesis is to provide an easy-to-use interface for programmers, and abstract the intricacies of RDMA. This abstraction eases the development and decreases the resistance to use RDMA for improved performance. Thus, this dissertation developed a middleware solution, providing a simple interface, as long as RDMA-capable hardware is available.

The main components of the cache are the Endpoint Manager and the instances of Client and Server endpoints that rely on DiSNI to establish and manage connections. The Endpoint Manager is a crucial component as it manages all the endpoints available and exposes functionality through the Client API.

The main features provided by the cache are a simple PUT/GET API, available through the middleware, and a performant solution for data transfer between nodes, relying on RDMA mechanisms. Anyone with access to the hardware can use it transparently as if it was any other cache with no extra knowledge required.

The use case analysis shows the impact that network overhead has when sending data using TCP sockets. The devised experiment shows that even when connecting to a local endpoint, the overhead caused by the TCP connection is very significant. When compared to direct memory access on the local machine, it performs 1000 times worse. This result is essential because many distributed caches rely on TCP connections between the nodes, including Memcached. The overhead impacts remote and local accesses, because, for example, Memcached uses a TCP connection to fetch a value in the local node. This overhead is mainly due to memory copy between userspace and kernel space, and with RDMA we can benefit from a zero-copy protocol.

The results obtained with the prototype in chapter 3 are very important to understand the impact and the intricacies of buffer handling. The distributed cache has a vital role as it takes care of the buffer handling, providing an easy to use API. In the experiment with the prototype, it is possible to understand the impact of inadequate buffer handling.

The benchmark and analysis of the proposed solution against one of the most used distributed caches, Memcached, corroborated that the RDMA-based distributed cache is more performant. Compiling all the experiments, it is possible to say that resorting to the presented cache improves the performance three to four times in terms of throughput and response time. In particular, if we consider the experiment with variable sized values, the Memcached's performance decreased, and the distributed cache maintained the same average values. These results show that the distributed cache is five times more performant in this particular case, which is closer to a real-world scenario.

6.2 FUTURE WORK

The distributed cache is a middleware, and the next step is to integrate it into a system that requires distributed caching. Regarding the scope of this dissertation, it is most useful to integrate the cache in a distributed query processing engine, for example, Apache Spark or Apache Flink.

The cache is not production-ready, and it needs two enhancements to ensure its stability for a production environment. The first action is to benchmark the cache in a distributed computing framework, to understand if the performance improvements sustain in production environments. Also, fault-tolerance could be considered, resorting to redundancy, ensuring that all values stay in cache until evicted, even in the event of node failure.

Apache Flink's methods are more exposed to the development level, which allows easy integration of a new cache. As such, it is a strong candidate as a framework to evaluate the cache's performance with an industry-standard benchmark.

Currently, values are stored in one node and, if that node dies, the cache continues operational, but the values in that node are no longer available. This is a problem because the values are deleted from a fault and not from eviction. The introduction of redundancy to the cache will provide fault-tolerance in these cases and, as a result, values only disappear from the cache when evicted.

These two aspects improve the cache in different fashions. The benchmark serves the purpose of determining if the cache can benefit a real-world application with easy integration. The fault-tolerance contributes to the delivery of a better product, that is production-ready and stable.

BIBLIOGRAPHY

[1] Supplement to infiniband^{TM} architecture specification, annex a16: Rdma over converged ethernet (roce), volume 1, release 1.2.1. Technical report, Infiniband^{TM} Trade Association, September 2014.

[2] Supplement to infiniband^{TM} architecture specification, annex a17: Rocev2, volume 1, release 1.2.1. Technical report, Infiniband^{TM} Trade Association, September 2014.

[3] Infiniband^{TM} architecture specification, volume 1, release 1.3. Technical report, Infiniband^{TM} Trade Association, March 2015.

[4] Alluxio, 2018. URL http://alluxio.org/.

[5] Apache Arrow - A cross-language development platform for in-memory data, 2019. URL https://arrow.apache.org/.

[6] Apache Flink - Stateful Computations over Data Streams, 2018. URL https://flink.apache.org/.

[7] Apache Hadoop, 2019. URL https://hadoop.apache.org/.

[8] Apache Parquet, 2019. URL https://parquet.apache.org/.

[9] Apache Spark - Unified Analytics Engine for Big Data, 2018. URL https://spark.apache.org/.

[10] M. Chen, S. Mao, and Y. Liu. Big data: A survey. *Mobile Networks and Applications*, 19, 04 2014.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[12] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, 2014. USENIX Association. ISBN 978-1-931971-09-6.

[13] Dremio - Data-as-a-Service Platform, 2019. URL https://www.dremio.com/.

[14] U. Elzur, J. Carrier, R. Recio, P. Culley, and S. Bailey. Marker pdu aligned framing for tcp specification. RFC 5044, IBM Corporation and Hewlett-Packard Company and Sandburst Corporation and Broadcom Corporation, October 2007. URL https://tools.ietf.org/html/rfc5044.

[15] M. Girault. Hash-functions using modulo-n operations. In D. Chaum and W. L. Price, editors, *Advances in Cryptology — EUROCRYPT' 87*, pages 217–226, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg. ISBN 978-3-540-39118-0.

[16] J. Hilland, P. Culley, J. Pinkerton, and R. Recio. Rdma protocol verbs specification. Technical report, IBM Corporation and Hewlett-Packard Company, april 2003. URL https://tools.ietf.org/html/draft-hilland-rddp-verbs-00.

[17] iWARP RDMA Here and Now. URL https://www.marvell.com/documents/54a11326t7lnomlwfmof/.

[18] B. Jenkins. A hash function for hash table lookup, 1996. URL http://www.burtleburtle.net/bob/hash/doobs.html.

[19] A. Kalia, M. Kaminsk, and D. G. Andersen. Using rdma efficiently for key-value services. *ACM SIGCOMM Computer Communication Review - SIGCOMM'14*, 44(4):295–306, 2014.

[20] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. ACM STOC*, 02 2001. doi: 10.1145/258533.258660.

[21] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. *USENIX NSDI*, 2014.

[22] Memcached, 2018. URL http://memcached.org/.

[23] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. *USENIX ATC*, 2013.

[24] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A remote direct memory access protocol specification. RFC 5040, IBM Corporation and Hewlett-Packard Company, October 2007. URL https://tools.ietf.org/html/rfc5040.

[25] Redis, 2018. URL https://redis.io/.

[26] R. Rosen. Infiniband. In *Linux Kernel Networking: Implementation and Theory*, chapter 13. Apress, New York City, 2014.

[27] R. Sedgewick and K. Wayne. Algorithms. 2011.

[28] H. Shah, J. Pinkerton, R. Recio, and P. Culley. Direct data placement over reliable transports. RFC 5041, IBM Corporation and Hewlett-Packard Company and Microsoft Corporation and Broadcom Corporation, October 2007. URL `https://tools.ietf.org/html/rfc5041`.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX. ISBN 978-931971-92-8. URL `https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia`.