Increasing the Efficiency of Data Rebalancing in the Redis Distributed Key-Value Store

Efstratios Ntallaris

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Assoc. Prof. *Kostas Magoutis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH).

This work has been partially supported by the Greek Research Technology Development and Innovation Action "RESEARCH - CREATE - INNOVATE", Operational Programme on Competitiveness, Entrepreneurship and Innovation (ΕΠΑνΕΚ) 2014–2020 through the Proximiot project (T1EΔK-04810).

University of Crete
Computer Science Department

Increasing the Efficiency of Data Rebalancing in the Redis Distributed Key-Value Store

Thesis submitted by
Efstratios Ntallaris
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: Στράτος Νταλλάρης

Efstratios Ntallaris

Committee approvals:

KONSTANTINOS MAGOUTIS
11.04.2022 14:36

Kostas Magoutis
Associate Professor, Thesis Supervisor

ANGELOS BILAS
ANGELOS BILAS
Apr 12 2022 8:45 AM

Angelos Bilas
Professor, Committee Member

Evangelos Markatos
Professor, Committee Member

Departmental approval:

Polyvios Pratikakis
Digitally signed by Polyvios Pratikakis
Date: 2022.05.09 09:28:32 +03'00'

Polyvios Pratikakis
Associate Professor, Director of Graduate Studies

Heraklion, March 2022

Increasing the Efficiency of Data Rebalancing in the Redis Distributed Key-Value Store

Abstract

Data re-balancing is an important mechanism used in data-intensive distributed systems when a change in the underlying resources or imbalance in the workload (e.g., a skew in the access pattern) necessitates re-distributing data across available storage nodes. Data re-balancing requires transferring (migrating) data between nodes to ensure a more equitable allocation in terms of space and/or processing capacity. The efficiency of the data re-balancing process, in terms of speed as well as impact on application performance is important, especially when used in an online fashion; in this case, the re-balancing process is also known as *elasticity*. Fast, low-impact data re-balancing actions are desirable, however they require careful design and implementation to achieve in a distributed key-value store.

In this thesis, we study data re-balancing in a widely-deployed real-world system, Redis Cluster. Our study starts by analyzing the default data re-balancing mechanism (data path) in Redis Cluster, exposing sources of inefficiency in its design and implementation, and demonstrating their impact in measured performance in experiments under an I/O-intensive workload, the Yahoo Cloud Serving Benchmark (YCSB). We then design and implement a version of Redis Cluster that features an improved data-rebalancing path, leveraging remote direct memory access (RDMA) for low-overhead data transfers. Our results show that our improved version of Redis Cluster can transfer data to a new (joining) node in nearly wire speed (1GB/s) over a RDMA-capable network connecting a 5-node cluster, while allowing sufficient spare resources for application (YCSB) processing. Our improved version of Redis Cluster achieves significantly faster data re-balancing compared to standard (and tuned) Redis in YCSB read-only workloads while also maintaining lower CPU and memory footprint during data transfers.

Αποδοτική Επαναστάθμιση Δεδομένων στο Κατανεμημένο Σύστημα Αποθήκευσης Κλειδιού-Τιμής Redis

Περίληψη

Η επαναστάθμιση δεδομένων είναι ένας σημαντικός μηχανισμός που χρησιμοποιείται σε κατανεμημένα συστήματα αποθήκευσης μεγάλου όγκου δεδομένων μετά από αλλαγές στους πόρους του συστήματος ή ανισορροπία στον φόρτο και απαιτεί την ανακατανομή των δεδομένων μεταξύ των διαθέσιμων κόμβων του συστήματος. Η επαναστάθμιση δεδομένων απαιτεί μεταφορά δεδομένων μεταξύ κόμβων για να εξασφαλίσει ομοιόμορφη κατανομή από άποψη χώρου και επεξεργαστικής ισχύος. Η αποδοτικότητα της διαδικασίας επαναστάθμισης δεδομένων, σε όρους ταχύτητας και επίδρασης στην επίδοση της εφαρμογής (ειδικά κατά την διαδικασία της δυναμικής μεταβολής πόρων, γνωστή και ως *ελαστικότητα*), είναι σημαντική. Γρήγορες και με χαμηλό αντίκτυπο ενέργειες ανακατανομής δεδομένων σε κατανεμημένα συστήματα αποθήκευσης κλειδιού-τιμής είναι επιθυμητές ιδιότητες, απαιτείται ωστόσο προσεκτική σχεδίαση και υλοποίηση για να επιτευχθούν.

Στην παρούσα εργασία μελετούμε τον μηχανισμό επαναστάθμισης δεδομένων στο δημοφιλές σύστημα Redis Cluster. Αρχικά αναλύουμε τον προκαθορισμένο μηχανισμό επαναστάθμισης δεδομένων του Redis Cluster, μελετώντας τις πηγές αναποτελεσματικότητας στη σχεδίαση και την υλοποίησή του, και παρουσιάζουμε την επιρρόη τους με μετρήσεις κάτω από φόρτο εργασίας παραγόμενο από την εφαρμογή στάθμισης απόδοσης YCSB. Στη συνέχεια σχεδιάζουμε και υλοποιούμε μία έκδοση του Redis Cluster με βελτιωμένο μηχανισμό επαναστάθμισης δεδομένων, εκμεταλλευόμενοι τις δυνατότητες του μηχανισμού Remote Direct Memory Access (RDMA) για μεταφορές δεδομένων με υψηλή ταχύτητα και μειωμένο κόστος σε υπολογιστικούς πόρους. Τα αποτελέσματά μας δείχνουν πως η βελτιωμένη έκδοση του Redis Cluster μπορεί να μεταφέρει δεδομένα σε έναν νεοεισαγόμενο κόμβο κοντά στην ταχύτητα γραμμής (1GB/s) σε ένα δίκτυο συστοιχίας 5 κόμβων, αφήνοντας ταυτόχρονα ελεύθερους πόρους επεξεργασίας διαθέσιμους στην εφαρμογή. Η βελτιωμένη έκδοση του Redis Cluster πετυχαίνει μία σημαντικά πιο ταχεία επαναστάθμιση δεδομένων σε σύγκριση με τον μηχανισμό επαναστάθμισης δεδομένων της δημόσια διαθέσιμης έκδοσης του Redis Cluster υπό φόρτο του YCSB κατά την ανάγνωση κλειδιών-τιμών, με χαμηλή χρήση πόρων επεξεργασίας και μνήμης στην μεταφορά δεδομένων.

Acknowledgments

I would like to thank all those who contributed to the completion of this work. First and foremost, I am more than grateful to my supervisor Prof. Kostas Magoutis and Postdoctoral Researcher Antonis Papaioannou for their guidance and continuous support throughout my thesis. I would like to also thank the members of my committee, Prof. Angelos Bilas and Prof. Evangelos Markatos for their valuable questions and comments during my defense.

Contents

List of Figures

Chapter 1

Introduction

Distributed in-memory key-value stores (KVSs) are key building blocks for large-scale Internet applications today. With the continuous growth of data and the increasing popularity of web services, databases have been facing scalability issues. To address this challenge, in-memory distributed KVSs have been increasingly used as a caching layer between the web service and the database, thus reducing the access load to the database. In-memory KVSs are designed to provide high-performance (providing more operations per second) and low latency (small time for each request to be served). Besides better scalability, the significantly lower latency of serving directly from memory is particularly important for applications that provide interactive response to Internet users.

There has been a lot of attention from industry and academia to further improve performance, availability and scalability of distributed in-memory KVSs. Distributed in-memory key-value stores further need to adapt to change, such as rapidly add new capacity due to a workload surge. Such adaptation requires re-balancing data across the KVS cluster, a process that involves large amounts of data transfer and may incur significant I/O overhead due to protocol processing, memory copying, data serialization/deserialization, TCP/IP overhead.

Data rebalance may be needed for multiple reasons. The system may need better load balancing and keys that are accessed the most to be spread across all nodes in the cluster. A cluster node may need maintenance at some point of time and due to that access to this node may temporarily need to be restricted. We can migrate the data off the node temporarily until the maintenance is done. Workloads are often bursty, requiring dynamic adjustment in the processing capacity across all resources (processing and storage). Online adjustment of system capacity is often referred to as *elasticity*. Systems that support *elasticity actions* can adapt resources dynamically during its lifecycle and allow nodes in the system to join or leave without a significant impact on the performance or availability. Some distributed in-memory key-value stores are not elastic during re-balance.

Today's network technologies and protocols are not specifically geared to efficiently support the continuously growing demand for node-to-node movement of

data. Applications often require high bandwidths (which can be served by link speeds of 10 Gbps, custom today, to 40Gbps and more). Current network technologies and protocols (such as those based on the TCP/IP stack) cannot be used effectively at such speeds because of the high CPU overhead [16]. Remote Direct Access Memory (RDMA) is becoming widely adopted in data-centers because of its ability to deliver low latency, high bandwidth, and low CPU overhead to data-center applications. Even though many systems adopt RDMA in their designs, they have not specifically focused and studied the data-rebalancing path, which features heavy data transfers and requires significant processing. This path is key to rapid, efficient elasticity actions.

In this thesis we study the data re-balancing process in a widely used in-memory KVS, Redis, and its distributed version, Redis Cluster, and identify sources of inefficiency in its default data migration mechanism. In response, we design and implement a more efficient data migration mechanism for Redis Cluster [10] that uses remote direct memory access (RDMA) [1], to improve data-migration performance. Our results show that Redis Cluster can fully utilize the available network bandwidth for data migrations (transfer data at nearly wire speed (1GB/s)) while also minimize the CPU overhead of data transfers, thus, leaving spare cycles for application processing. An important observation of our work is that a challenge for single-threaded system designs such as Redis is how to handle concurrent tasks (such as application request processing, as well as tending to I/O completion) that need to simultaneously make progress during a data-migration phase. We have addressed this problem in our current implementation in a straightforward way by adding a second thread during this phase used when safety can be ensured (currently in read-only settings), as an interim (indicative, although not satisfactory) solution. Fully addressing this problem for general (read/write) workloads requires extensive re-design of Cluster Redis, which is out of scope of this thesis but part of our ongoing and future work.

1.1 Thesis contributions

In this thesis we make the following contributions

- Studied sources of inefficiency in the data-rebalancing path in the widely used in-memory KVS Redis
- Designed and implementated a new data migration mechanism on top of RDMA
- Highlighted the challenges of replacing and the requirements of incorporating RDMA protocols in real system that is designed to perform data rebalancing action with traditional TCP/IP mechanism
- Designed a new, more efficient data migration path for Cluster Redis

- Implemented of new path in Cluster Redis version 6.2.5
- Experimentally evaluated and compared to unmodified, tuned Cluster Redis

The rest of this thesis is structured as follows: Chapter 2 includes background information regarding the Redis KVS used to apply our proposed mechanism as well as related work in the field distributed KVSs that use RDMA protocols for their inter-node communication. In Chapter 3 we dive into more details in the data rebalancing mechanism in Redis Cluster KVS. In Chapter 4 we describe the design and implementation of our proposed data migration protocol during the rebalancing actions of the KVS while in Chapter 5 we evaluate the performance of our mechanism and highlight the performance improvements. Finally in Chapter 6 we conclude this thesis and discuss future work.

Chapter 2

Background and related research

In this chapter we provide background information on the standard way of network data transfer using sockets and TCP/IP, typically via the remote procedure call (RPC) abstraction, and the benefits of an alternative of transferring data via remote direct memory access (RDMA), which we leverage in this work. We then relate the work reported in this thesis to previous research in this space. Background about Redis is presented in Section 3.1 prior to our presentation of data rebalancing in the Redis Cluster key-value store.

2.1 Data transfer via the TCP Sockets API

In this section we describe the most important performance issues of TCP sockets and what are the drawbacks on multi-gigabit interconnects. On of the major issues is the multiple data placement. On the transmit path, when *write()* is issued on the transmit path, the data is first copied by CPU from buffers in user-space into intermediate socket buffers on kernel-space. Kernel then places the data into TCP packet along with other information such as Sequence Numbers, checksum, ports etc. TCP packet is then sent to the network interface controller via DMA. On the receive path, the procedure is similar but in reverse order. TCP packet is first copied from network interface(via DMA) to an intermediate socket buffer in kernel and then is copied by the CPU into userspace. An application can fetch the data on the receiver side by issuing *read*. The copying handled by CPU as well as the TCP/IP stack processing(compute checksum, ensure the packets are in order, etc etc) makes the TCP/IP stack inefficient for applications demanding high bandwidths (10 Gbps, 40Gbps or more). As being described in [16] most of the overhead comes from data copies performed by CPU in order to copy data from kernel to user-space application buffers.

An example of TCP data transfer is shown in Figure 2.1.

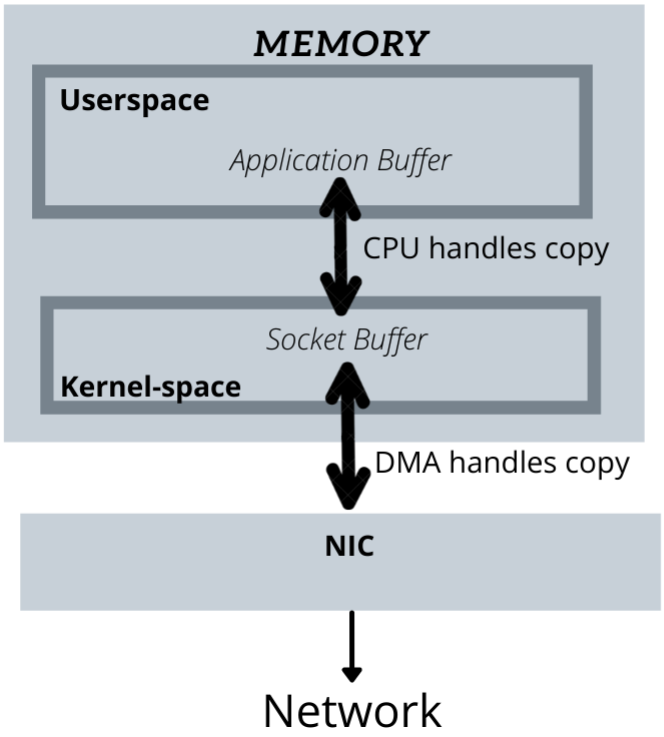

Figure 2.1: TCP Data Transfer example

2.2 Remote direct memory access

TCP/IP is the most used network stack in today's data center networks. However, with the increasing demands of network bandwidth and CPU resources in datacenters TCP/IP stack is heavy for today's data center networks [26]. With link speeds up to 10GBit per second, the processing for TCP/IP stack consumes significant amount of CPU Cycles(because of the reasons described in Section 2.1). *Remote Direct Memory Access (RDMA)* is the ability to access the remote host memory directly providing low-latency, high throughput with small CPU consumption. RDMA offers better performance compared to the traditional network technologies in three dimensions. First, RDMA provides one-sided RDMA semantics which bypass the receivers side CPU. Second the active side is issuing requests directly from userspace bypassing the kernel(thus avoiding unnecessary copies and CPU overhead). Third RDMA gives the ability to use a technique called zero-copy, meaning RDMA avoids memory copying while doing transfers. Three RDMA technologies are used nowadays:

- Infiniband defines its own protocol stack, specialized for its own adapters, switches and cables. Infiniband also supports IPoIB(IP over Infiniband) which allows socket based applications to run on Infiniband networks with

no modification.

- iWaRP defines its own protocol stack to deliver messages over TCP/IP.
- RDMA over Converged Ethernet (RoCE) is the most commonly deployed RDMA technology. RoCE is the only RDMA service that is used in Ethernet networks and operating over layer 2 and layer 3 Ethernet switches.

2.2.1 RDMA semantics

With RDMA, applications in userspace access RDMA Enabled Network Interface Cards (RNICs) directly using functions called *verbs* without involving the kernel. Verbs are distinguished in two categories. The one sided verbs in which only the host issuing the operations is actively involved in the data transfer.

One-sided verbs are:

- RDMA Read: source host specifies the local address to where the data should be written and also the remote address from where the data should be read.
- RDMA Write: source host specifies the local address from where the data should be written and also the remote address to where the data should be read.

And the two-sided verbs in which both host's CPU are actively involved in data transfer. Two-sided rdma data transfer is very similar to the traditional sockets data transfer. The receiving application proactively posts a work request containing the memory region data should be written. Unlike traditional sockets where data transfer requires intermediate buffers in the kernel, two sided verbs in rdma require pre-registered buffers in userspace in order to transfer the data into them and avoid kernel.

Each verb is posted inside RNIC in form of Work Request (WR). In order to post verbs a queue pair (QP) between the two hosts must exist. In RDMA terms a connection between two nodes is called QP. Each Queue Pair consists of a receive queue(RQ) and send queue(SQ). A queue pair also has a completion queue in order to place completions of work requests. Verbs are posted as work queue elements (WQE) in send queue or receive queue of a queue pair. Upon a completion of transaction of a work queue element a Completion Queue Element (CQE) is generated and placed on the completion queue. The application can poll a completion queue to know whether the data has been written on the remote side or read from remote side.

2.2.2 Memory regions

Applications using TCP/IP assumes protocol buffers are provided by the operating system. This approach however requires multiple intermediate buffer copies which leads to CPU and Memory overhead. RDMA on the other hand requires the

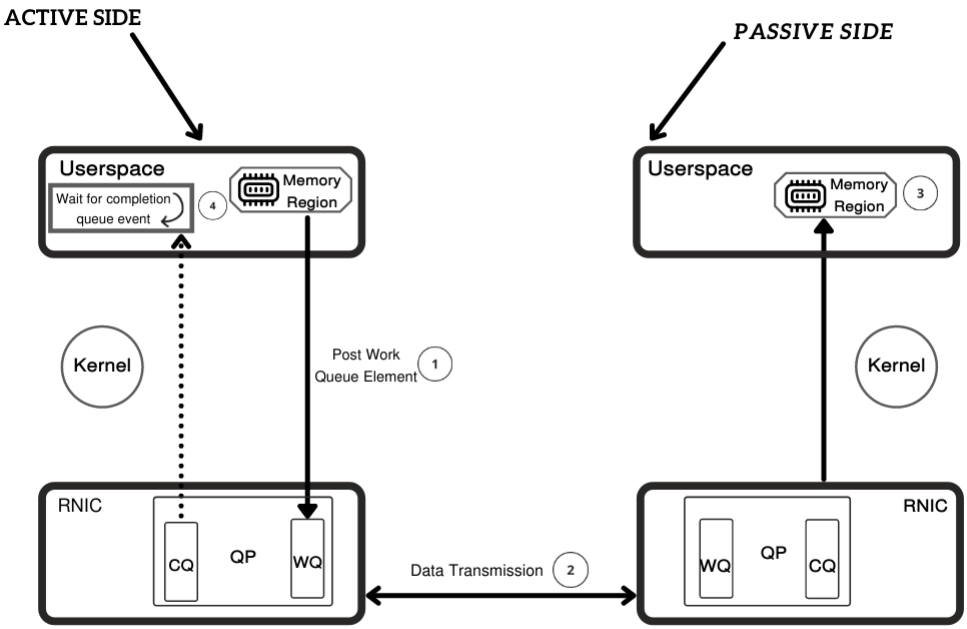

Figure 2.2: RDMA Request flow

application in user space to pre-allocate the communication buffers. Those buffers are called memory regions (MR). Memory regions are registered with RNIC to then be used with DMA. If one side communication has to be used, then the active side (the side issuing work requests) must know remotes side memory region address and the remote memory region private key in order to operate.

To perform one-sided RDMA operations, the application running in userspace has to allocate virtual memory and register a *memory region (MR)* with the RNIC. The RNIC assigns a pair of local and remote keys for each of the *MRs*. The active side in order to issue one-sided rdma verbs has to know a virtual address that falls into the remote *MR* as well as the remote key (*rkey*) of the that remote *MR*.

An example of one sided RDMA transfer can be seen in Figure 2.2. RDMA connection is successfully made and memory regions are registered in order to issue RDMA operations. First step the application on the active side is creating a work request and is posting it as a work queue element in RNICs Work Queue. Depending on the issuing operation on the active side, data is either being transmitted and copied to passives side memory region or being read from the passives side memory region and copied directly to the actives side memory region, without passives side CPU intervention.

2.2.3 RDMA enabled Network Card Interfaces

RDMA enabled Network Card Interfaces (RNICs) are the hardware that implements most RDMA functionalities. RNICs contain hardware logic to implement RDMA protocols and to store connections metadata such as memory region keys and virtual memory addresses. RNICs also store a page table entry cache with mappings MR virtual address pointers to DMA address. The memory on RNICs is limited, therefore only data that is used frequently is stored on the RNIC. When memory in RNIC is full, the data is stored in main memory(through PCIe bus) and fetched only when is requested.

2.2.4 RDMA Connection Types

RDMA supports two type of connections. The reliable connections which guarantees delivery as well as ordering of packets, and the unreliable connections which do not guarantee delivery and packet ordering. Reliable and unreliable connections can also support connected and unconnected queue pairs. In Connected queue pairs, each queue pair connects to exactly on queue pair of the remote node. In unconnected queue pairs one to many connections can be achieved. Infiniband also supports Reliable Datagrams (RD) and unreliable Datagrams. For unreliable datagrams only send and receive semantics can be used while for the reliable datagrams send and receive while also one sided rdma write/read can be used.

2.3 Related research

There has been much work in recent years to improve the performance of distributed key-value stores bypassing host CPU processing and kernel involvement via the use of RDMA [22, 13, 17, 20, 15, 25]. While RDMA-optimized storage systems have been an active field of research for more than 20 years [21, 27], the emergence of distributed key-value stores have thrown new light and stirred new research into low-overhead high-bandwidth RDMA-optimized in-memory data stores. In this section we review related work in the space of (mainly in-memory) RDMA-optimized distributed key-value stores, systems that are closest to our work.

Memcached/RDMA [25] proposed the use of one-sided RDMA reads for GET (read) operations as a way to lower the CPU overhead of the memcached server, while using regular RPC over sockets for SET (write) operations. Upon issuing a GET operation, a client will check their metadata *stag* (an identifier of a remote memory region) for a given key exists. If an stag metadata entry is not found, the GET operation is sent over standard RPC over sockets. The server then responds using one-sided RDMA write to store the requested key-value pair in the client's memory, along with the associated stag for the requested key-value pair. If an stag entry is found in the client's metadata for the requested key, a one-sided RDMA read is issued to read the requested key-value pair from server memory.

Pilaf [22] is a distributed key-value store that utilizes Cuckoo hashing [24] using three hash functions to compute three different hash buckets. Pilaf uses one-sided RDMA reads for GET operations. When trying to read a key-value from the server, the client first reads the first hash bucket. If the key is not there, it then reads the second hash bucket. If the key is still not found, the client issues one last RDMA read for the last bucket. PUT operations are done with lightweight RPC based on two-sided (send/receive) Verbs. Pilaf uses CRC64 checksums to validate that the key-value is not corrupted under data races during GETs.

FaRM [13] aims to provide a programming model for distributed shared memory. It uses one-sided RDMA reads for lookups (GET operations), and circular buffers with polling on receiver side to support bi-directional accesses for PUTs. RDMA writes are being used for write/update (PUT) operations. FaRM also provides a hash table, implemented on top of the shared address space. Hopscotch hashing [3] is used to keep key-value pairs that collide close to the original bucket. Each bucket keeps a single key-value pair. If a bucket is empty then the key-value pair is inserted. If the bucket is not empty then the algorithm tries to find a neighboring empty bucket to insert the key-value pair. If an empty bucket is not found, the algorithm displaces key-value pairs to move an empty bucket close to the neighborhood. Upon not finding an empty bucket or insertion of key-value pair fails the algorithm then resizes the hash table and repeats the procedure. The process of repeating reads by the the client to go over linked buckets in server memory increases bandwidth use and read latency in FaRM.

Comparing FaRM with Pilaf, FaRM fetches bigger memory blocks, larger than the actual key value size for a single key-value pair, while Pilar is requesting more RDMA reads for a single key-value pair, affecting the latency and the operations performed in the RNIC. While both Pilaf and FaRM aim to maximize use of RDMA in both reads and writes, Memcached/RDMA follows a more conservative design using RPC for writes, therefore it does not fully leverage RDMA.

HERD [17] uses one-sided RDMA writes to transfer (write) GET and PUT requests (commands) to appropriate server memory regions reserved for storing requests. A server polls the memory regions to find and process the incoming requests. HERD also uses two-sided (send/receive) verbs for messaging to reply to clients requests. This design needs only a single round-trip to write a request on server memory. One key aspect in which HERD differs from systems such as [22, 13, 20, 15, 25] is that HERD uses unreliable datagram (UD) Queue Pairs (QPs) to improve the performance and scalability of RDMA. FaRM, Pilaf and Memcached/RDMA use reliable connection (RC) QPs.

KV-Direct [20] is a recent system that leverages FPGA-based programmable NICs to extend RDMA semantics and to support key-value operations natively, further offloading the remote CPU. KV-Direct uses DRAM on the RNIC as a cache for key-value pairs to minimize PCIe access to main (host) memory.

FaSST [18] is working over UD QP connections with two-sided(send/receive) RDMA verbs. FaSST main aim is to scale as the number of nodes grows. It has a thread called coroutine to poll Completion Queues for incoming data and execute

RPCs.

L5 [15] proposes a low-latency messaging library for node-to-node communications. L5 uses RDMA writes with ring buffers (similar to FaRM [13]) while polling the memory to detect new messages. L5 uses a mailbox buffer where each client occupies 1 byte. When a client wants to perform SET operations, they first issue a one-sided RDMA write to write the payload on the server memory region reserved for the client. It then issues one more one-sided RDMA write on the single byte of the mailbox buffer that corresponds to the given client. Since one-sided RDMA writes are executed in order, the mailbox byte will be written when writing of the payload is complete. Since the client has to issue two one-sided RDMA reads for each GET operation, this affects the latency and the operations in RNIC.

Dragojević et al [14] described the implementation of transaction, replication and recovery protocols over FaRM [13] leveraging commodity networks using RDMA and large amounts of DRAM. For transaction executions, instead of the two-phase commit protocol (which requires round-trips and CPU processing) they use their own protocol named FaRM commit. In FaRM commit they use two-sided RDMA verbs (effectively RPCs) for the Lock phase of the protocol, and for Validate, Replicate and Update and unlock are done with one-sided rdma verbs. CPU is used only in the Lock phase. The rest of the protocol steps are low latency and do not require cpu intervention. In case of failure, theres a special role for a node in the cluster called configuration manager. The coordinaor manager sends heartbeats to all the other nodes and in case a failure is detexted, configuration manager coordinates all the nodes in the cluster to stop issuing one sided rdma operations to the node that has failed and then it sends the new configuration including the backup node and the new rdma regions in order to start serving requests.

Previous work has applied RDMA in the common (read/write) path in Redis [28], in which clients use one-sided RDMA (inlined) writes over UC transport. In this work, servers maintain registered memory regions specifically for RDMA communication, divided into N chunks (where N is the number of clients). Each of the N chunks is divided into Request Buffer and Respond Buffer. Clients write the request buffers. When a requested is processed by the server, a client's response buffer is updated with the result. The server then uses two-sided communication to send the response back to the client.

One important observation for systems like Pilaf [22], FaRM [13], HERD [17], and L5 [15] where RDMA-accessible buffers in server-side memory are scanned to detect and process incoming requests by the server, is that although these systems may benefit from RDMA transfers, the overhead of such polling may be high. In our review of related work we have not identified a thorough evaluation of the CPU overhead of this server-side mechanism.

Rocksteady [19] is a mechanism implemented on top of RAMCloud [23] that uses two-sided RDMA to migrate data. RAMCloud stores data in tablets, which are unordered tables that are parts of of key-value stores. Ordered secondary indexes can also be constructed on top of tablets. RAMCloud tablets to be moved in

a migration may be scattered in memory. Before the migration starts, Rocksteady transfers the ownership of tables to be migrated on the target node. Writes can be serviced immediately on the target, while reads can be serviced only if records have already been transferred to the target. If a read is requested on a record that does not exist in the target, the latter prioritizes the requested records to be migrated immediately and clients try again after a few milliseconds to fetch again. Evaluation is done on 40Gbps Ethernet network. Rocksteady can transfer data during migration at 758 MB/s (whereas the network supports up to 4GB/s) with 99.9th percentile access latency less than 250µs. During migration, Rocksteady copies data into intermediate buffers (logs) and then replays them on the receiver side. That procedure introduces processing overhead for recovery purposes and thus limits Rocksteady from transferring data at write speed (4GB/s), fully benefiting from RDMA.

The related work discussed in this section (with the exception of [19]) focuses mostly on the common path of receiving and processing key-value requests. Our focus in this thesis is on the data-rebalancing (migration) path where the data transfers are usually heavy and typically consume significant CPU and memory. For all of the systems described [22, 13, 17, 20, 15, 25] RDMA operations are performed between a client and a server, whereas in our work RDMA operations are performed server-to-server. Aspects of the related work relating to common-path processing could also be applicable to our work. Our focus in this thesis broadens the scope of RDMA as a key component for high-throughput low-overhead communication.

Chapter 3

Data rebalancing in Redis

In this chapter we discuss the Redis in-memory key-value store and the baseline data rebalancing mechanism in Cluster Redis, in which the data migration protocol plays a key role. We start by presenting the possible states of *slots*[1] in Section 3.4 followed by the redirection commands in Section 3.3. We then describe the data-migration protocol in Section 3.5. Finally, we discuss challenges and limitations for data-rebalancing stemming from the design and implementation of the baseline Cluster Redis data-migration protocol, which we address in the optimized design presented in the next chapter.

3.1 The Redis in-memory key-value store

Redis [9] is an in memory key-value store, and thus keeps all the key-value stores in memory while operating. Some of the data structures supported are Strings, Lists, Sets, Sorted Sets, Hashes, Bitmaps, Streams and much more. For some production systems, volatile in-memory storage is not acceptable, therefore data may also be persisted on disk periodically. Redis supports two different persistence options.

- AOF or append only file in which write commands are saved saved in file and in case of failure these commands are re-executed.
- Snapshot mode which periodically takes snapshots of the in-memory dataset and then persists it.
- If volatile storage is needed, Redis can operate under the Nosave mode, which disables the persistence of key-value stores completely

Redis can run as a single instance (with strong consistency), or in cluster mode (for high availability). In Redis Cluster mode key-values are partitioned in *slots*, which are akin to *shards* in other key-value stores. The keys are hashed using the CRC16 algorithm [2] and distributed to 16384 hash slots, and each slot is assigned

[1] Recall that a slot is akin to a shard, a common concept in distributed key-value stores

to one of the Redis nodes [10]. Slots are assigned to nodes when cluster is first started. If a cluster starts with three nodes (which is the minimum number of nodes of redis cluster) then an example of topology is the following:

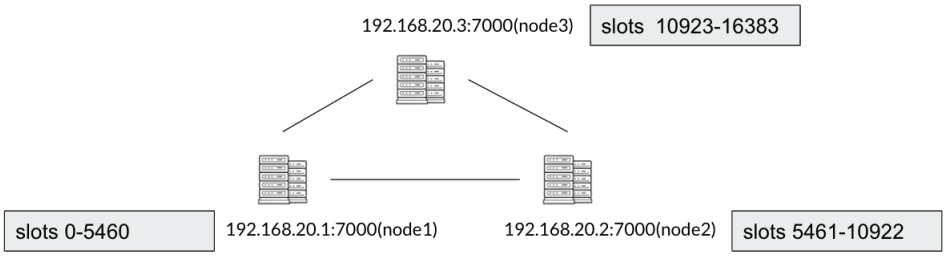

Figure 3.1: Redis Cluster Topology with three nodes

As seen in Fig 3.1 each of the three nodes is assigned about 5461 slots which is the 1/3 of the total slots. Slots are almost equally distributed across all three nodes. An important thing to note is that when a new node is being added to the cluster, resharding wont be done automatically. For example if *node4* joins the cluster after the initial setup, the cluster wont be rebalanced thus *node4* will remain with zero slots.

Redis Cluster has a replication protocol that follows the master-slave replication model. Each master[2] has a subset of those 16384 slots. If case a master node dies, then one of the replicas is elected as the new master node serving the slots of the failed node. Replication in Redis works in a asynchronous way, meaning that while the replication happens the master can serve requests.This is done in order to keep the performance guarantees of Redis, however it is possible to lose data if failure occurs while replication happens, after a master has acknowledged a write before replication has completed. Therefore consistency guarantees are violated.

Each of the Redis Cluster nodes keeps N-1 (where N is the number of cluster nodes) incoming and outgoing active connections with the rest of the nodes and uses gossip protocol for liveness and to distribute shared management state such as the slot-to-node assignments. The gossip protocol is also used to propagate information to discover new nodes, send ping packets to discover which nodes do not operate normally.

In Redis Cluster mode nodes use TCP sockets to communicate and transfer data which results in inefficiencies described in Section 2.1.

3.2 Internal data structures

In this section we describe briefly how data is stored in Redis. Data is kept in memory in the form of dictionary (as hash table). Each entry in the dictionary

[2]A master is a distinguished node (a leader) within a replica group

has a unique key and a value. Operations are done based on keys. Any key or value that is saved inside redis is saved as *redisObject*. This structure keeps the metadata of each key value pair in the dictionary. It contains a data pointer (*ptr*), reference counter (for the garbage collector) (*refcount*), a timer to know whether a key-value expires (*lru*), the encoding type in case data is encoded(*encoding*), the type of data(*type*) which defines the data pointer underlying data structure (whether it is a string, set, hash table etc). Based on redis source code the definition of *redisObject* (*robj*) is defined as:

```
typedef struct redisObject{

    unsigned type:4; // data type, is it set? string? hash?
    unsigned encoding:4; // if there is any encoding
    unsigned lru:LRU_BITS; //in case the object expires
    int refcount; // reference counter in order to do garbage collection
    void *ptr; // pointer of the actual data
} robj;
```

A dictionary contains hash table nodes (*DICTENTRY*) in which key values are stored. A *DICTENTRY* contains pointers for key metadata, value metadata and for the next *DICTENTRY* hash table node. The hash table uses chain address to resolve collisions. Multiple key value pairs can be assigned to the same index in the form of linked list. Each key is unique and all the operations are done based on the keys. Based on redis source code, *dictEntry* is defined as:

```
typedef struct dictEntry {
    void *key; // key
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
    } v; // value
    struct dictEntry *next; // next node (linked list) in the same
        dictionary index
} dictEntry;
```

In order to reach actual data we have two in-directions, one in-direction to reach the metadata(*robj*) from dictionary entry (*dictEntry*) and one to reach the actual data. This also can be seen in the next figure 3.2:

3.3 Redirection commands

When a redis cluster node receives a key related command, it first calculates the corresponding slot the key belongs and then finds the corresponding cluster node for this slot. If the cluster node containing the corresponding slot then the key command is processed. Otherwise a special reply is being sent back to the client

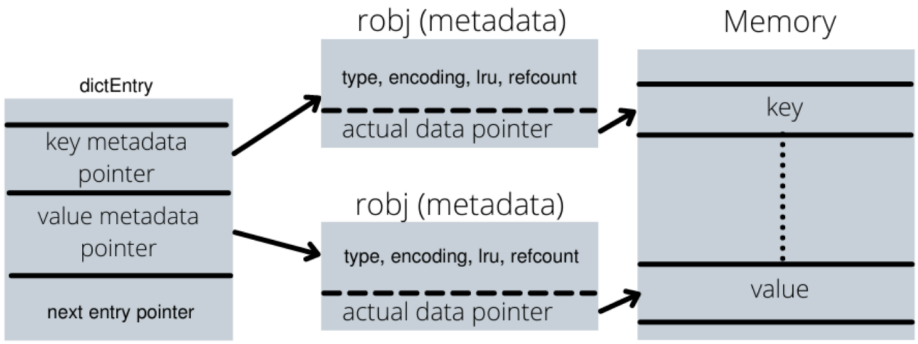

Figure 3.2: Redis Key-Value Memory Layout

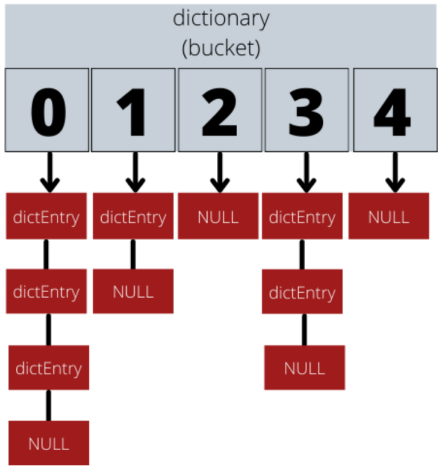

Figure 3.3: Redis Dictionary Memory Layout

(ASK, MOVED), and the client redirects the key command to the new cluster node.

There are two type of redirection commands in Cluster Redis:

- **ASK**: During rebalance if a command is received for a slot that is currently migrating and key does not exist in the node migrating the data, an ask reply is sent back to the client. Ask reply contains the new cluster node that the data is being migrated to. Client then redirects the command to the new cluster node with an ASK flag in order to be executed. Ask redirections can only be triggered by the node that it is migrating data. Client does not update its cache that the corresponding slot is on the redirected node.
- **MOVED**: The MOVED redirection can be issued by any cluster node that does not contain a specific slot to indicate that the slot having the key is on another cluster node. The client may update its cache with the new

corresponding node for the specific slot.

An example scenario where Slot 1 is being moved from Node 1 to Node 2, and at the time of re-balance a client requests an operation for key X, is depicted in Figures 3.5 and 3.4. If the key exists in Node 01 then the operation is executed as usual as shown in 3.4. Figure 3.5 depicts an alternative scenario where the key

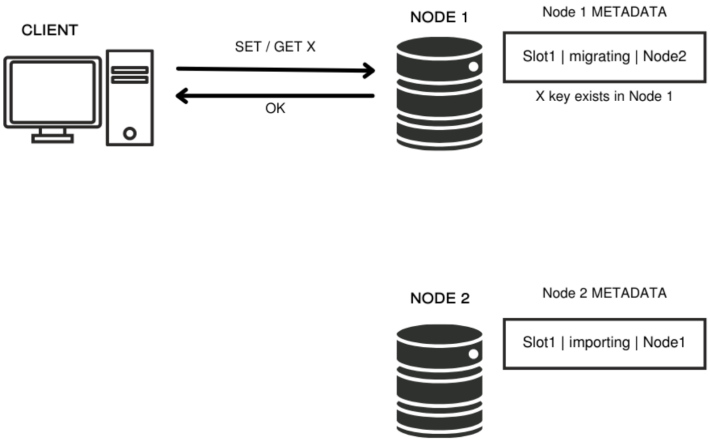

Figure 3.4: Client requests a key that exists in Node 1

does not exist in Node 1. In that case, an ASK reply is sent back to the client. Client then redirects the request to Node 2 with the ASK Flag.

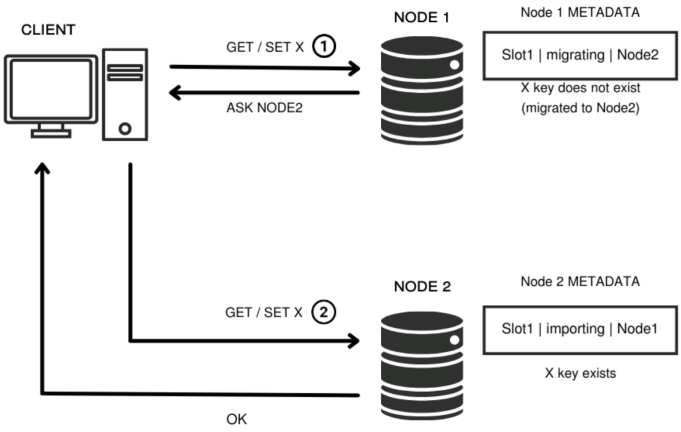

Figure 3.5: Client requests a key that does not exist in Node 1

When no rebalance action is happening, if a key related command is received on any redis cluster node, first the hashslot will be calculated. If the hashslot exists on the redis cluster node that received the request, the request will be served as usual. In case the slot does not exist on the redis cluster node, the redis cluster node will find the corresponding node to serve that key based on a slot-to-node

mapping on the metadata.Then "moved" command will be returned on client along with the redis cluster node that contains the slot for the specified key.

3.4 Slot states

Cluster Slots change states in order to be migrated from one cluster node to another. Depending on the slot state, operations maybe redirected (with ASK and MOVED commands) to the right nodes for the operation to be done.

- MIGRATING: A slot that is marked as migrating, any command received for the existing key is processed as usual. If a key does not exist in the cluster node and in the slot is in migrating state, then an ASK reply is being transmitted back to the client containing the node that the command has to be redirected to(migrating target node).
- IMPORTING: For a slot that is marked as importing, commands will be executed only if the command has been redirected with ASK.
- STABLE: A slot is marked as stable when a rebalance between two cluster nodes is complete. It clears the IMPORTING and MIGRATING states of slots.

3.5 Standard (unmodified) migration protocol

In this section we describe the Redis Migration Protocol. For the rest of the protocol description Sender is the cluster node moving the data(key-value pairs) away to the new node. Receiver is the cluster node receiving data(key-value pairs) from sender. When a new node joins the cluster or user decides that re-balance action must be done, the following actions takes place:

As seen in Figure 3.6, the Cluster Node represents a cluster node that we have connected with the Redis Command Line Interface (Redis-cli) to trigger rebalance action between cluster nodes. This can be any node in the cluster. The cluster node is responsible to orchestrate the whole data rebalancing action i.e. decide the slots to be moved and schedule the data migration tasks. User selects how many slots have to be moved, then the underlying algorithm divides the number of slots each node has to send and which slots have to be moved. Slots are selected sequentially from each node until the number of slots that have to be moved is fullfilled. This is the default way to select and move slots, however the number of slots as well as which cluster nodes will participate in the migration can be changed from user. Sender is the cluster node that the data has to be moved from. Receiver is the cluster node that data is moved to. At first, the cluster node sends commands to change slot states on Sender and Receiver (Step 1 and 2 in Figure 3.6). This is done with the commands CLUSTER SETSLOT <slot> IMPORTING to change the slot state to importing on the receiver, and CLUSTER SETSLOT

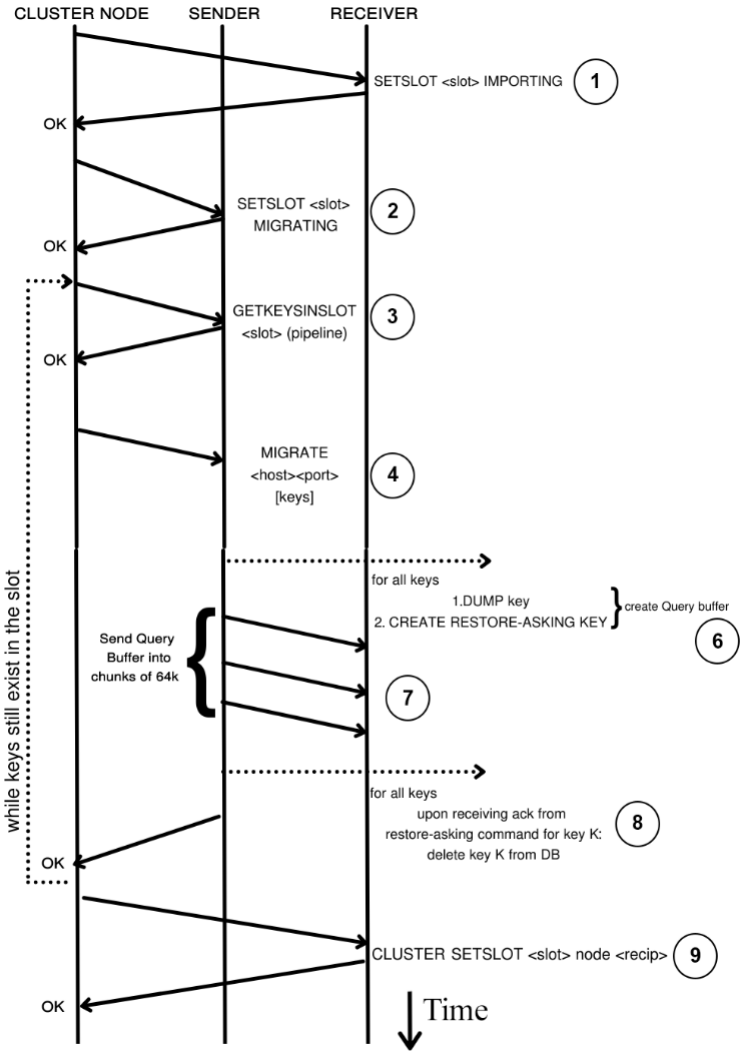

Figure 3.6: Redis Migration Protocol

<slot> MIGRATING to change the slot state to migrating on Sender. From now on operations for keys existing in the slot will be treated as described in the Section 3.3. Then Cluster Node sends GETKEYSINSLOT <slot> to the sender to get a set of keys from the slot that has to be migrated(Step 3 in Figure 3.6). Pipeline presents the number of keys that command GETKEYSINSLOT receives each time. The set of keys returned from GETKEYSINSLOT is then passed as a parameter to the MIGRATE command and sent to the Sender node(Step 4 in Figure 3.6). Then on the Sender node all the keys will be serialized (DUMP key) and formed to a new query in order to be executed to the Receiver node. Those new queries that have to be executed on the Receiver node will be appended on a query buffer(Step

6 in Figure 3.6). This Query Buffer is passed from sender to receiver as chunks of 64kB and executed(Step 7 in Figure 3.6). Each key is deserialized and appended to the database and for each key an ack is sent back to the sender(Step 8 in Figure 3.6). For each of the keys, the Sender node waits for an acknowledgement that a key has been deserialized and successfully added to the database and deletes it from the database of the Sender.

The last step is to send CLUSTER SETSLOT <slot> NODE <receiver> to notify all the nodes in the cluster that slot has changed ownership(Step 9 in Figure 3.6). Changing the ownership requires at least one node (Receiver) to be notified that the ownership changed. Then through the gossip protocol all the nodes will be notified that the ownership of the slot has changed. However CLUSTER SETSLOT <slot> NODE <receiver> may be sent to all the cluster nodes in order to be notified for the ownership change as soon as possible.

3.6 Challenges and limitations

The baseline Redis implementation exhibits certain challenges in relation to performance during the data-migration protocol. For one thing, Redis always serializes data in the sender side and deserializes them at the receiver, costing CPU cycles and hurting throughput. Our implementation (described in the next chapter) allows us to not serialize nor deserialize data (when it is safe to do so[3]) which benefits us by not spending CPU cycles for those procedures.

Another challenge is that data must be copied to intermediate buffers on the sender side and also on the receiver side consuming more space than it is required (in kernel and in user-space for the protocol) and wasting CPU cycles. With our implementation we avoid copying data to intermediate buffers by transferring them directly to remote memory regions and then directly point to the actual data in the dictionary.

A resulting challenge is that while Redis Cluster is re-balancing, resources of the nodes participating in the action are not used efficiently, as we demonstrate in the evaluation section. In summary, we show that network bandwidth is bound to 10MB/s (while the available network bandwidth is 1GB/s) and no more than 3% of the total CPU resources are being used, leading to long time rebalancing actions and affecting the overall performance of the cluster.

Another limitation of Redis stems from the fact that it is single threaded. Although multiple Redis instances can run on same machine and therefore still benefit from multiple cores, a Redis instance cannot serve requests during data-migration RPC processing, and this affects migration performance. RPCs that involve complex processing (such as MIGRATE) can affect the cluster performance dramatically as is demonstrated in the Evaluation Section. No matter how many Redis instances running on a node the problem of serving still remains since slots are bound to a Redis instance.

[3]When byte ordering, memory layout/references, etc. are not expected to be an issue

We can get over some limitations discussed in this Section and improve the overall Redis performance. Moving more than one slot at a time we can increase the bandwidth use. By adding a new thread to handle the rebalance procedure will improve the overall performance since requests will not be queued and will be served independently of the migration procedure. However because of the protocol (serialization, de-serialization, copying to intermediate buffers etc) and TCP/IP stack, there will be CPU and memory overhead. With our implementation described in Chapter 4 we avoid intermediate data copies due to protocol or TCP/IP stack, avoid serialization when it is not necessary, and we can transfer to nearly wire speed without CPU overhead.

Chapter 4

Improving data migration with RDMA

In this section we describe the design and implementation of an improved data rebalancing mechanism that aims to solve the limitations and challenges described in Section 3.6. Our aim is to exploit the benefits of RDMA and multi-core CPUs to improve the current Redis data migration protocol. We do this by extending the protocol with new commands described in Section 4.2. In Section 4.1 we touch upon optimizations in memory allocation that were implemented outside the scope of this thesis but used in the implementation and evaluation of Redis-Opt. In Section 4.3 we provide an overall view of the Redis-Opt data migration protocol and explain its workings in detail.

4.1 Data migration protocol improvements

In the standard default Redis migration granularity is in term of keys. The standard (default) Redis memory allocator does not enforce any sort of affinity (coalescing) between logically related entities(e.g keys belonging in the same slot). RPCs used to migrate data between nodes (*MIGRATE* RPC [8]) take keys as parameters. This bring inefficiencies in the migration protocol since 1.On donor side slots have to be scanned in order to find the keys belonging in the same slot and 2. RPCs have limits in terms of arguments and size. As the number of keys grows per slot the the overhead of the overall migration grows.

In our implementation migration granularity is in terms of slots, instead of keys used in standard Redis. We group keys to slot by implementing an allocator specialized for this kind of procedure. We try to store keys of the same slot in contiguous memory regions(called blocks). By using our allocator we reduce the overall overhead of the protocol since the number of slots are fixed (16384) while the number of keys can be infinite. We also avoid scanning the slots in order to find the keys existing in that slot. Rather than the initial protocol optimizations, for RDMA operations memory coalescing is an important factor for optimization,

since less memory registrations are done in RNIC and multiple work requests can be posted as one (since data is contiguous and therefore can be treated as one big chunk).

A brief example of the Redis (unmodified) memory layout can be seen in Figure 4.1:

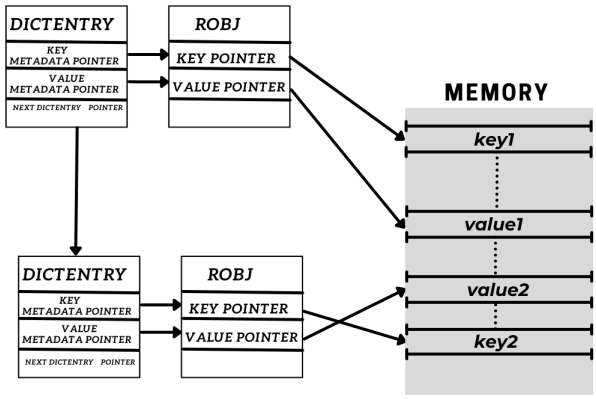

Figure 4.1: Redis (unmodified) memory layout

As can be seen in Figure 4.1 key value pairs are saved randomly in memory. For each key value pair (as well as their metadatas) we have to register a memory region in RNIC (two memory regions for the metadata structures, one memory region for key and one memory region for value) and then post four work requests to transfer a single key value pair. Although the protocol will successfully transfer key-value pairs to the recipient side, it brings inefficiency and RDMA cannot be used effectively. With our implementation of the allocator we only register as memory regions the blocks of the slots that have to be transferred and post a work request for each block.

4.2 Commands

We use and modify the existing protocol in order to prepare rdma structures and connections on both sides (sender of the slots and the receiver). The new commands implemented as an extension to the Redis slot migration (Section 3.5) to support optimized data rebalancing protocol, are the following:

- **rdmaMigrateSlots** [HOST] [PORT] [SLOTS]
  The receiver of this command starts a new thread to handle migration. The referenced slots will be moved from the receiver of this command to the Redis instance serving at [HOST]:[PORT] . A similar command is the *MIGRATE* command of redis (unmodifed). *MIGRATE* command takes as parameters

keys. Our command (rdmaMigrate Slots) takes as parameters slotIDs that have to be moved. This command is doing the proper rdma setup and then issues one sided rdma operations in order to transfer the data.

- **initRDMAServer** [RDMA PORT]
  The receiver of this command initiates server listening to [RDMA PORT]. This command is being sent from the node that is issuing the migration(the node that sents the data) to the recipient of the data.

- **CLUSTER SETSLOTS**
  An extention of CLUSTER SETSLOT COMMMAND [[ADD REFERENCE]]. This command changes the state or the ownership of multiple slots. CLUSTER SETSLOTS is the batching version of SETSLOT. Instead of sending multiple SETSLOT commands, we send only one containing multiple slots.

- **rdmaRegister** [$Slot_1$, $Slot_1\_NUMBLOCKS$, $Slot_2$, $Slot_2\_NUMBLOCKS$, ..., $Slot_N$, $Slot_N\_NUMBLOCKS$]
  This commands takes as arguments slotIDs and number_of_blocks to be registered for each of the slotID. The receiver of this command will allocate on the receiver side the needed blocks and register the memory regions with the RNIC for the data to be transferred with rdma. This command returns the pointers as well as the keys of the memory regions registered in order to initiate one sided RDMA communication.

- **rdmaDone** [Slot_1, Slot_2, ..., Slot_N]
  This new command is sent to the recipient after RDMA transfer is done.It takes as parameters the slotIDs that have been transferred with RDMA. It starts a new thread, which iterates all block slots finds the key-value pairs, and is doing the proper pointer fixes. Recipient will receive the key-value pairs as well as their metadata. As seen on 3.2 metadata has one indirection to reach the actual data. When data is received on the recipient side, it contains pointers which does not point to the correct data. Therefore the thread is managing to do the proper pointer fixing for the metadata to point to the actual data based on the allocator data layout. Then the thread is proceeding with adding the key-value pairs into the receivers dictionary.

4.3 Protocol

Commands that have long processing time (rdmaMigrateSlots, rdmaDone) are handled by a additional separate thread introduced in our implementation of the data rebalancing mechanism; thus, the main Redis thread can continuously process client requests as data migration actions are hadled by a separate thread. In standard Redis while migration command is served, no other request can be served. Therefore we want time consuming RPCs to be handled by threads. One major difference between our protocol and standard Redis is that the migration in our

protocol migration is done for more than one slots. In standard Redis, for each slot the state is changed then migrated and then the ownership is changed for that slot. In our implementation we first change the state for all the slots that will eventually be migrated, then move all the migrating slots, and then change the ownership for all the migrated slots. We want to move data at wire speed (which in our case is the bottleneck) and benefit from RDMA capabilities.

Figure 4.2 describes our protocol. A Redis-Cli is connected to any cluster node and starts the procedure of migration by sending rdmaMigrateSlots command taking as parameters the slots ($slot_1, slot_2, ..., slot_N$) that has to be moved from donor to recipient. The cluster node receiving rdmaMigrateSlots command with create thread (rdmaThread) which will handle the rest of the migration procedure (Step 1 in Figure 4.2). Thread rdmaThread starts by setting the state for each of the slots ($slot_1, slot_2, ..., slot_N$) to migrating on the local node (donor) (Step 2 of Figure 4.2), and to importing on the receiver(recipient)(Step 3 of Figure 4.2). After successfully changing the state of the slots, rdmaThread sends initRDMAServer command to recipient to open an rdma server listening to port P and wait for a connection(Step 4 in Figure 4.2). After successfully initiating rdma server on the recipient side and receiving acknowledgment, donor will connect to recipient on port P and will create queue pair (Step 5 in Figure 4.2). Donor can now register all the blocks of the allocator that contain data for the slots ($slot_1, slot_2, ..., slot_N$).

Since donor now knows how many slots have to be transferred from donor to recipient, command rdmaRegister $Slot_1$ $Slot_1\_numblocks$ $Slot_2$ $Slot_2\_numblocks$ ... $Slot_N$ $Slot_N\_numblocks$ (where $Slot_i$ is the Slot id and $Slot_i\_numblocks$ is the number of blocks that have to be allocated and registered on the recipient side) is sent from donor to recipient. Recipient for each of the slots will allocate the number of specified blocks and will register the memory regions to the RNIC and return the pointers as well the keys of the memory regions to the donor(Step 6 in Figure 4.2). Donor now knows the data that has to be transferred and also the remote memory regions pointers and keys and can create the work requests. Donor will create a work request for each of the slot blocks ($WR_1, WR_2, WR_3, ..., WR_N$). Work requests are then formed as linked list, for each work request is write opcode is added ($IBV\_WR\_RDMA\_WRITE$) (in order to write recipients memory regions) and in the last work request WR_N send_flag $IBV\_SEND\_SIGNALED$ is activated (Step 7 in Figure 4.2).

That way one post will be made (Step 8 in Figure 4.2) to the RNIC and one completion event will be received (since write work requests are posted in order)(Step 9 in Figure 4.2). When a completion event is received on donor side, the transfer is complete and rdmaDone $slot_1 slot_2 ... slot_N$ can be sent to the recipient. Upon receiving rdmaDone on the recipient side, a thread (*PatchAndAddThread*) is created and an ack is returned on the donor side(Step 10 in Figure 4.2). Thread *PatchAndAddThread* is iterating all the blocks received from rdma, finds all the key-value pairs doing the proper backfixes on key and value metadata and appends them in the dictionary. When *PatchAndAddThread* completes iterating all the blocks and successfully appends the key-value in the dictionary, it

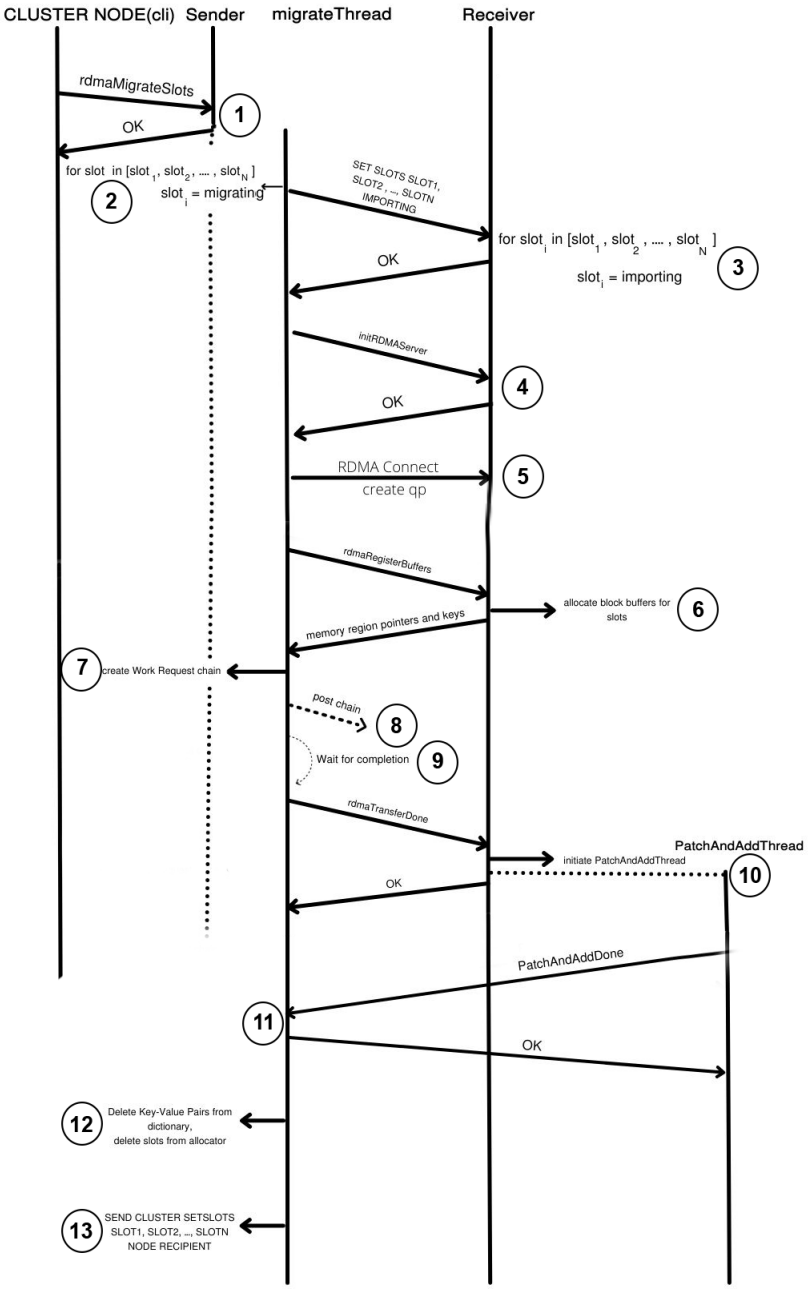

Figure 4.2: Redis-Opt Protocol

sends an *PatchAndAddDone* command to donor (Step 11 in Figure 4.2). Donor then knows that all the key-value pairs are appended on recipients dictionary and can delete the keys from the local dictionary (Step 12 in Figure 4.2) and update the ownership (Step 12 in Figure 4.2).

In the default Redis data is being copied to intermediate buffers on sender and receiver in order to be serialized and de-serialized which results to CPU and memory overhead. In our protocol, because of the implementation of the Redis Internal Data Structures described in Section 3.2 we can point directly to key-values after transfers are done and also avoid serialization/de-serialization CPU and memory overhead. Because of RDMA capabilities, our protocol does not use kernel buffers as well as buffers used by Redis default protocol. Data is copied directly to the final memory region and then pointed from the Redis hash table.

Chapter 5

Evaluation

In this section we evaluate the benefits of the optimized data migration path incorporated in the Redis-Opt implementation described in Chapter 4 during elasticity actions, and compare it to the baseline (but tuned) Redis memory key-value store. Our experimental results demonstrate that Redis-Opt leads to faster elasticity actions while reducing the overall performance impact observed by the clients during the elasticity action.

Our evaluation testbed hosting Redis clusters consists of 4 servers each featuring a Intel Xeon Bronze 3206R processor clocked at 1.90GHz with 32GB DDR4 memory, a Dell Micron 480GB SSD and a Toshiba 2TB 7200RPM hard disk. Network communication is performed over RDMA-capable RoCE QLogic/Marvell 41162 10Gb/s NICs via a Dell S4128T-ON OS10 switch. Each server runs Ubuntu 18.04.5 LTS. A fifth server with nearly identical specifications is used as a dedicated workload driver for the Redis cluster.

Throughout our evaluation, our initial Redis cluster consists of 3 servers (we refer to them as redis0, redis1, redis2), the minimum number of servers required to set up a Redis cluster[1]. Each Redis server is deployed on a dedicated node. The replication factor is set to one, i.e, the cluster consists of primary replicas only. This choice allows us to focus our study on the performance of the data-migration path between master nodes only; extending our study to include the effects of replication is part of our future work. We have also disabled persistence to disks [11]. While we believe that asynchronous persistence modes such as supported by Redis should not pose any performance impact on the data-migration path, this should also be investigated in future work.

In our evaluation scenarios we use the YCSB workload generator [12] configured to produce a 100% read workload on a dataset of 50 million unique records. Each record size is 1 KB resulting in 50 GB of total data size. The number of YCSB client threads (number of parallel connections between database client and servers) is set to 16, as we have observed that this is sufficient to maximize the achievable YCSB throughput in all tested configurations. The requested keys follow a zipfian

[1] https://redis.io/topics/cluster-spec

distribution [4].

We run a modified YCSB workload generator on the dedicated fifth node. Our modified YCSB workload generator uses lettuce redis client [6] instead of Jedis [5] as we have observed that in some occasions jedis does not cache mapping for slots to cluster nodes and therefore redirections happen and performance is negatively affected In the initial Redis Cluster of 3 master nodes, splitting the key space across them results in 5461 slots on each node (Section 3.1). This means that each of the three Redis servers initially hosts on average 16.7 million key-value records (17 GB in size). During the elasticity action when a fourth node is added to the cluster, a data migration action is necessary to rebalance the key distribution across the cluster nodes. In our experiments redis0, redis1, redis2 act as *donors* (slots are moving away from them) and redis3 as *recipients* (slots are moving towards it). We trigger a rebalance by moving 1365 slots (4.5 GB average total size) from each of the initial nodes (redis0, redis1, redis2) to the fourth node (redis3). After rebalancing is complete, the fourth node has received a total of 4095 slots from all the donors (resulting in 13.5 GB of data size of all the transferred key-value pairs).

Next we demonstrate and compare the performance impact and the resource utilization of an elasticity action under both the vanilla Redis KV store and our modified version of the store integrating the RDMA protocol for data migrations. During elasticity action, a number of Redis slots (Section 3.1) have to be moved from the existing nodes to the new node. The vanilla Redis store is configured to perform data transfers in bulk (all keys belonging to a single slot are moved in a single batch "move" command) using the –cluster-pipeline configuration option as we have empirically determined that under this settings Redis reduces the cost of data transfers. We will be referring to this configuration of Redis as Redis-Batch, evaluated in Section 5.1.

In the next subsections we describe the performance impact the system exhibits under vanilla Redis and Redis-Opt. We use two different performance metrics: the throughput (operations per second) and the response time reported every 2 seconds by the YCSB workload generator. We also report the CPU utilization on all active Redis nodes using the mpstat [7] tool.

5.1 Redis-Batch

In this section we analyze the performance of the standard Redis key-value store configured to migrate all keys belonging to a single slot within one move command. This optimization (referred to as pipelining in Redis documentation) is essentially a batching option, we thus refer to this version of Redis as Redis-Batch.

Figure 5.1 depicts YCSB throughput (operations per second) when scaling a Redis-Batch cluster from three to four nodes. The x-axis show the timeline of the experiment (as time of day). The vertical dashed lines show the duration of the elasticity action in which the three donors start simultaneously their data-migration protocols. The inner graph (YCSB 100% READS B) focuses on

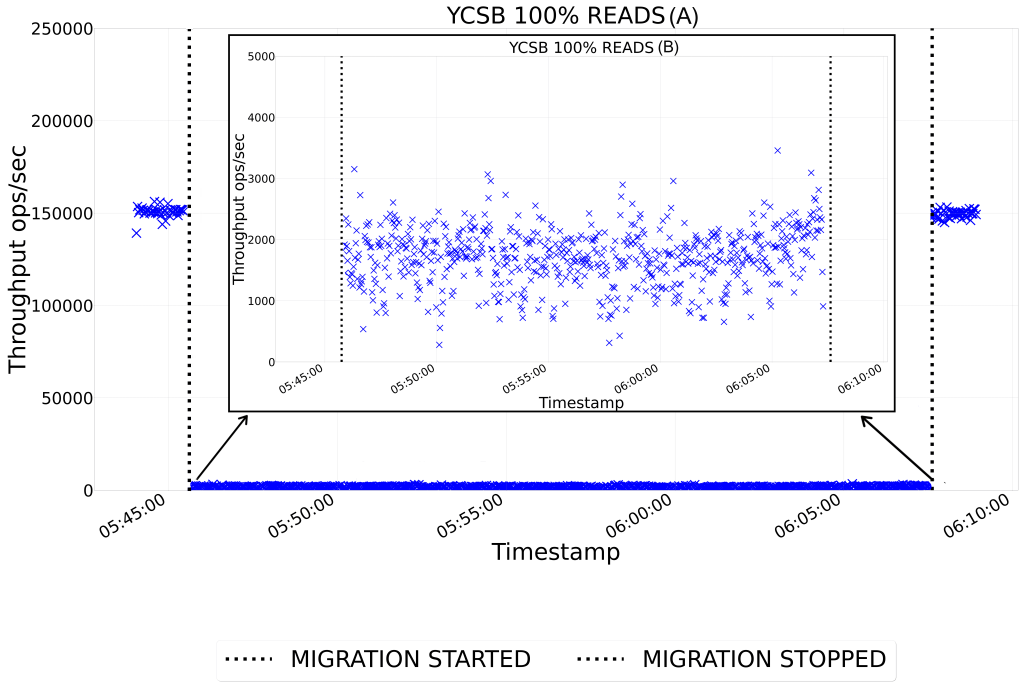

Figure 5.1: YCSB throughput 100% reads (Redis-Batch)

throughput specifically during the elasticity action, to provide more detail. Initially the average throughput is 150,928 ops/s. At 05:45:45 the system begins the process of data rebalancing as we trigger the elasticity action. We observe a throughput drop of 98.6% that persists throughout the elasticity action. That kind of drop is because of Redis' single-threaded design as data-migration RPC processing keeps the main thread mostly busy. Protocol overhead (data serialization, data transfers,intermediate copies) in this phase indeed requires significant involvement of the main thread. Finally when the data rebalancing is over and the elasticity actions is completed the throughput increases again to an average of 151,264 ops/sec.

Figure 5.2 depicts the latency (µs) of YCSB reads before, during and after rebalancing involving three donors and a recipient. The x-axis show the timeline of the experiment (time of day). The vertical dashed lines show the duration of the elasticity action. Our results are consistent with the throughput view presented earlier. Before rebalancing, the average delay is 208µs, drastically increasing to 18,628µs during rebalancing, restored to 210µs after rebalancing completes.

Figure 5.3 depicts the CPU usage of donors during the experiment for each of the nodes redis0, redis1, redis2. The vertical lines present the start and stop of the rebalance procedure of each node. Before migration each node uses almost 1 core (average 11% of cpu usage). Note that being a single-thread process, Redis can fully utilize only up to one CPU core. As we deploy a single Redis instance on each 8-core server, the maximum CPU utilization of each Redis instance (process)

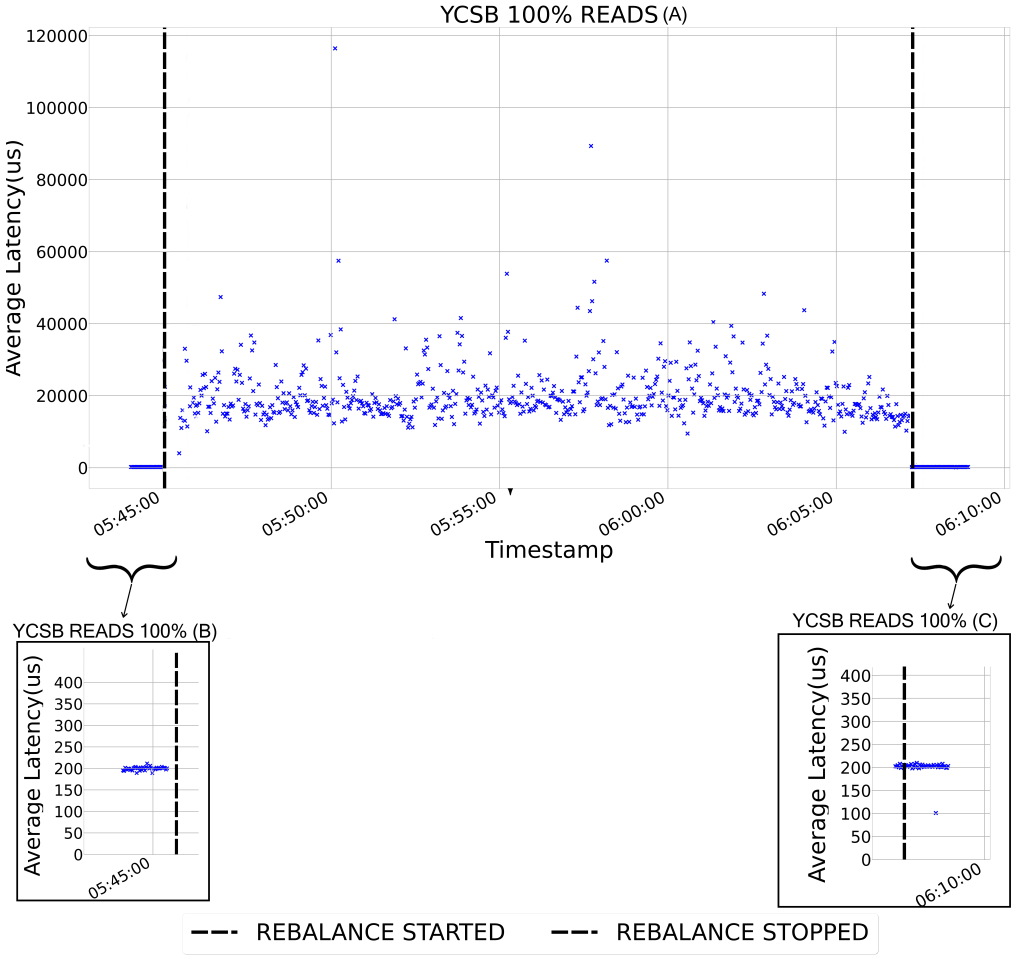

Figure 5.2: YCSB latency 100% reads (Redis-Batch)

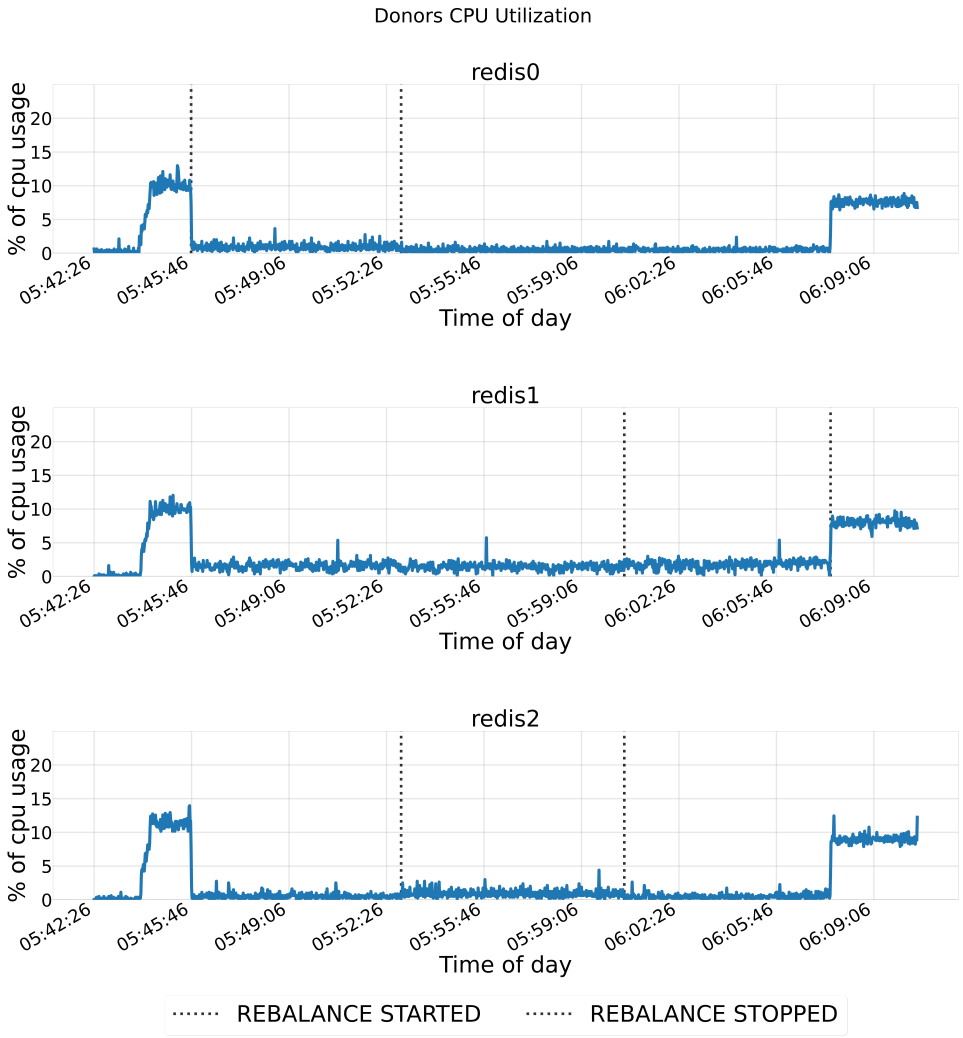

Figure 5.3: Donors' CPU utilization (Redis-Batch)

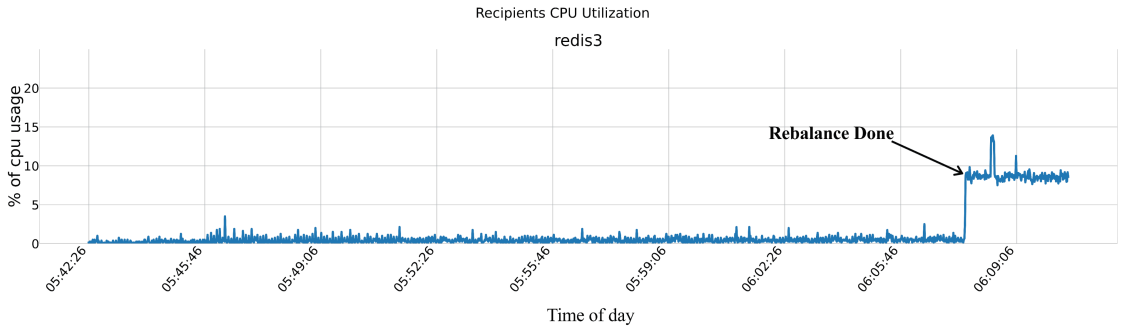

Figure 5.4: Recipient CPU utilization (Redis-Batch)

is 12.5%.

Consistent with our throughput and latency results, average CPU usage during rebalancing drops to 1.8% in redis0, 1.9% in redis1 and 1.9% in redis2. We observe that the data transfer phases for each donor appear to be serialized (redis0, redis2, redis1) as this is the default schedule enforced by the CLI (Section 3.5). When the first node (redis0) starts rebalancing, all nodes' CPU usage drops from 12% to 1.9% on average. CPU usage drops on all nodes as YCSB threads trying to do operations on key-value pairs that are on node that it is currently migrating. When the last rebalance finishes (redis1) we see the CPU usage increasing again to an average of 8.2%. Redis0 rebalance for 1365 slots took 7 minutes and 11 seconds, redis1 took 7 minutes and 3 seconds and redis2 took 8 minutes and 22 seconds. The overall data rebalancing process took 22 minutes and 36 seconds to transfer 4095 slots. We observe that despite our tuning of standard Redis, data transfer reaches only a small fraction of the feasible performance (limited by the 10Gbit/sec bandwidth supported by the network), set back by the inability of the data-migration protocol to batch transfers beyond a single slot.

Finally, Figure 5.4 depicts CPU utilization on the recipient (redis3, the newly added node to the cluster) during the experiment. CPU usage is on average 0.8% during rebalance (indicating that the recipient is not too busy during data transfer). After rebalance completes, CPU usage increases to about 8% as redis3 starts serving YCSB requests.

5.2 Redis-OPT

In this section we analyze the performance of our optimized version of Redis, which we refer to as Redis-Opt.

Figure 5.5 depicts the throughput (ops/sec) of YCSB reads before, during and after rebalancing of each donor. X-Axis is the time of day. During the migration action, existing redis nodes redis0, redis1, redis2 take turns to migrate data to the new redis instance redis3. To better visualize each data-migration action between a (donor, recipient) pair, we introduce a delay of 90 seconds between each such action. Since rebalance is handled by a dedicated thread (in addition to the main Redis thread), concurrent data transfers would introduce interference between multiple migration threads on the recipient.

The average throughput before rebalancing is 151,110 operations per second. Vertical lines represent the phases of Redis-Opt during rebalancing, namely:

1. **RDMA buffer preparation (at donor and recipient)**: from vertical line STARTED_RDMA_PREPARATION to START_POST_WRITES

2. **Data transfer:** from vertical line START_POST_WRITES to START_DELETE

3. **Deleting keys at donor:** from vertical line START_DELETE to STOP_DELETE

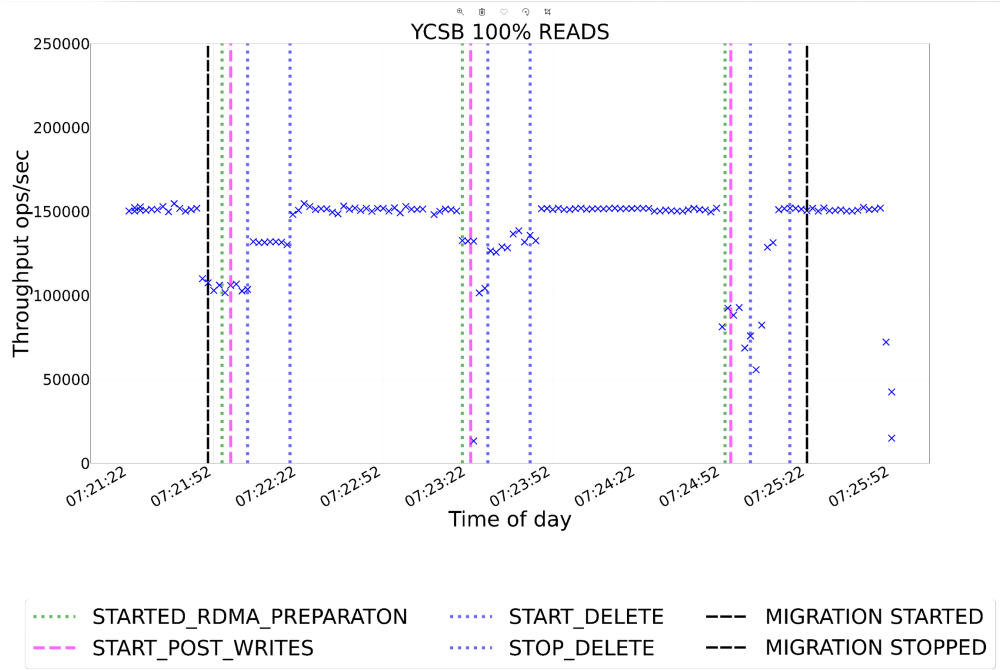

Figure 5.5: YCSB throughput 100% reads (Redis-Opt)

After the end of each rebalance action the throughput remains at nearly 151,260 operations per second. During Phase (2) for redis0, redis1, redis2 throughput seems to drop during data transfer as data migration network transfers compete with YCSB requests for network bandwidth (the donor's network link seems to be saturated during migrations). During Phase (3) there is also a throughput drop due to lock contention as both the migration thread and the core thread[2] need to lock the dictionary, the former to delete key-value entries and the latter to fetch key-value pairs.

Figure 5.6 depicts the average latency (µs) of YCSB reads before, during, and after rebalancing of each donor. The x-axis shows time of day. During the migration actions, existing nodes redis0, redis1, redis2 take turns to migrate data to the new instance redis3. The average delay before migrations is 211µs. Vertical lines represent the phases of Redis-Opt during rebalancing: (1) RDMA buffer preparation (at donor and recipient) (Steps 5,6,7 of figure 4.2); (2) data transfer(Steps 8,9 of figure 4.2); (3) deleting keys at donor(Step 12 of figure 4.2). During rebalancing the average delay increases to 360 µs for redis0, redis1 and 410 µs for redis3. Increasing in latency happens because: (1) YCSB requests compete with rebalancing data transfers for network bandwidth and (2) the migration thread locks the dictionary to delete key-value entries while also the core thread locks

[2]Recall that in Redis-Opt we use a separate thread to perform data migrations during elasticity actions (Chapter 4) so requests can be served by the core Redis thread during data migration

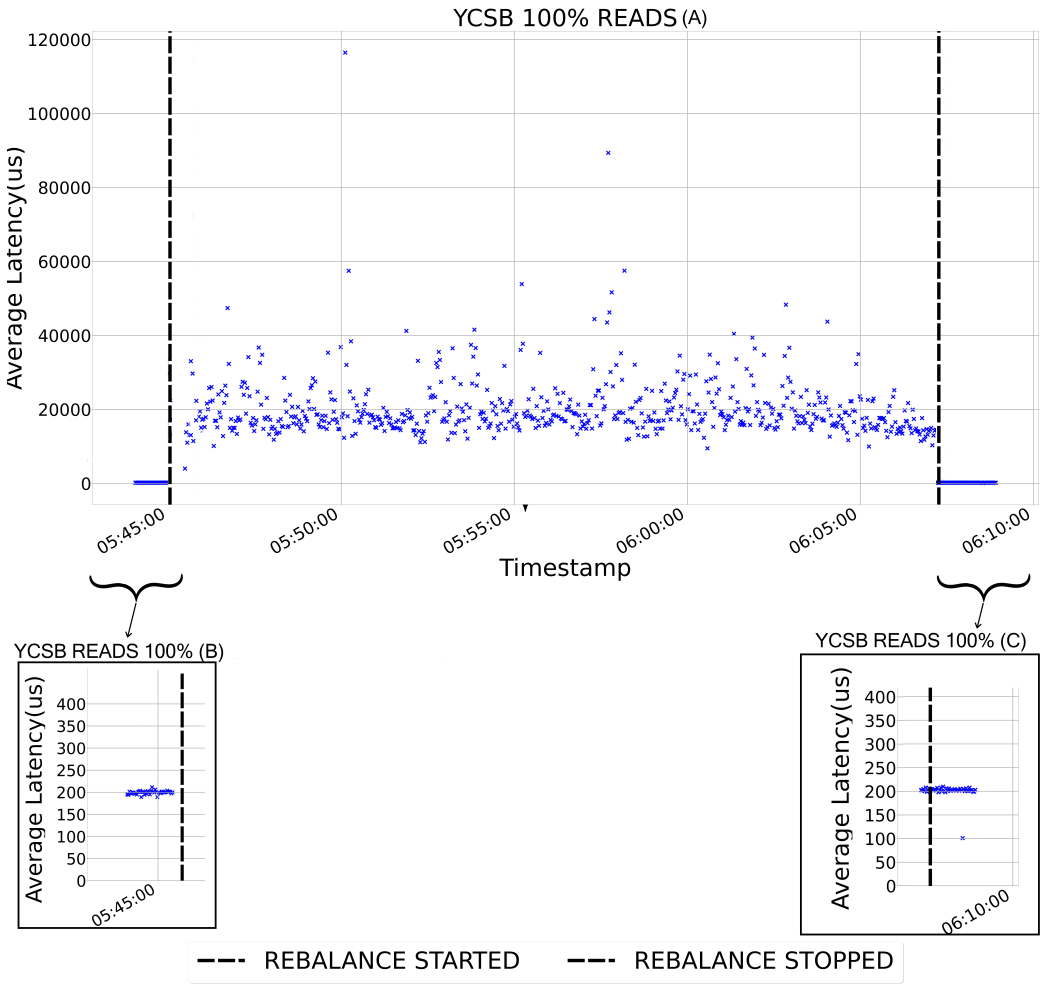

Figure 5.6: YCSB average latency 100% reads (Redis-Opt)

the dictionary to fetch key-value pairs. After the end of each rebalance action the average delay returns to an average of 221 µs.

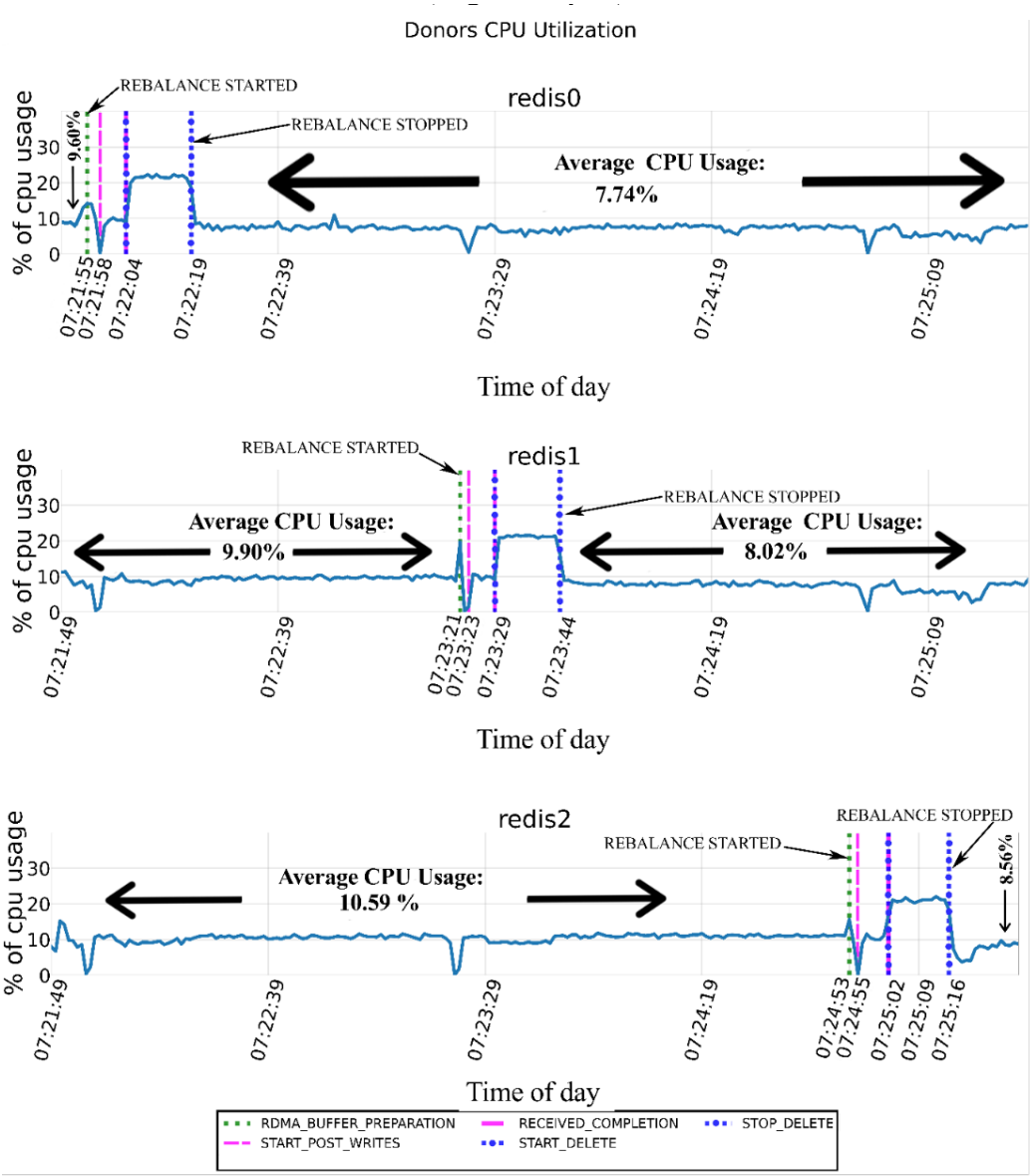

Figure 5.7: Donors' CPU utilization (Redis-Opt)

Figure 5.7 depicts the CPU utilization of donors during rebalancing. Note that with two active threads (one handling data migrations and another processing requests) we expect up to 25% total CPU utilization during elasticity actions. Vertical lines are the Redis-Opt rebalancing phases. The x-axis depicts time-of-day during the experiment. For redis0 the entire migration procedure took about

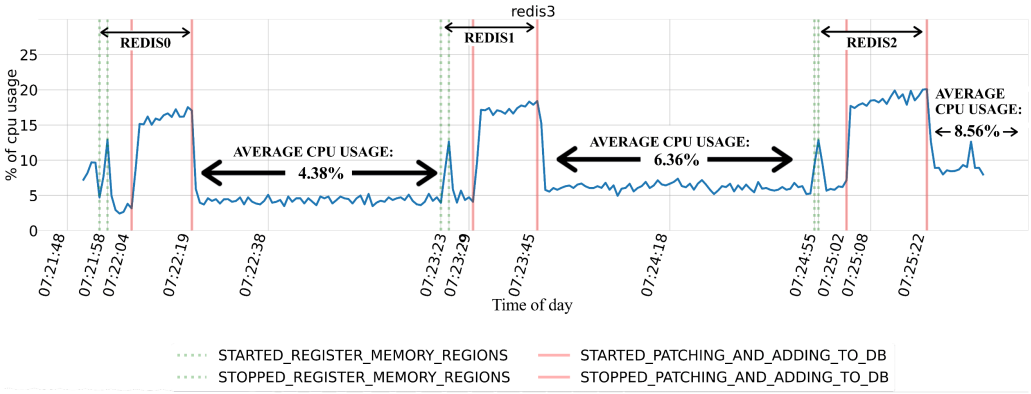

Figure 5.8: Recipient CPU utilization (Redis-Opt)

24 seconds, for redis1 lasted 23 seconds, and for redis2 23 seconds. Without the delays of 90 seconds the whole rebalance procedure for redis0, redis1, redis2 would last about 67 seconds.

Figure 5.8 depicts the overall CPU utilization on the recipient side. Vertical lines delineate the Redis-Opt phases on the recipient. We observe that after each rebalance action the average CPU usage increases (4.38% to 6.36% to 8.56%) as the recipient starts serving the new slots it just received each time.

Figure 5.9 shows CPU utilization for each donor while rebalancing in more detail. During RDMA Buffer Preparation (phase (a) in Figure 5.9) we see more than one CPU core being used. We then see a significant drop right before posting writes. This is because while the donor is preparing buffers an exclusive lock is used and therefore operations in the dictionary are temporarily not served. RDMA Buffer Preparation takes about 2 seconds. During RDMA transfers (phase (b) in Figure 5.9) no more than a single CPU core is used and the transfer takes about 5.5 seconds to transfer 4.7 GB of state which results in 8.5Gbit/sec, which is close to the wire speed (10Gbit/s). During deletes in phase (c) more than one core is being used. The time for deletes is about 16 seconds for 4.7 million keys on average.

Figure 5.10 presents CPU utilization of the recipient for a single rebalance action. RDMA Memory Registration(phase d, Step 6 of figure 4.2) takes about 2 seconds and the cpu usage is bound to one core since until buffer registration no thread handles rebalance. During phase (e) (Steps 8,9 of figure 4.2) rdma transfer is happening. There is no CPU overhead for migration data transfers during that time on the recipient side. CPU Core is used to serve requests. During phase(f) patching of pointers for each of the key-value pairs and adding to the dictionary is done and takes about 15 seconds. Phase(f) consumes more than one CPU core.

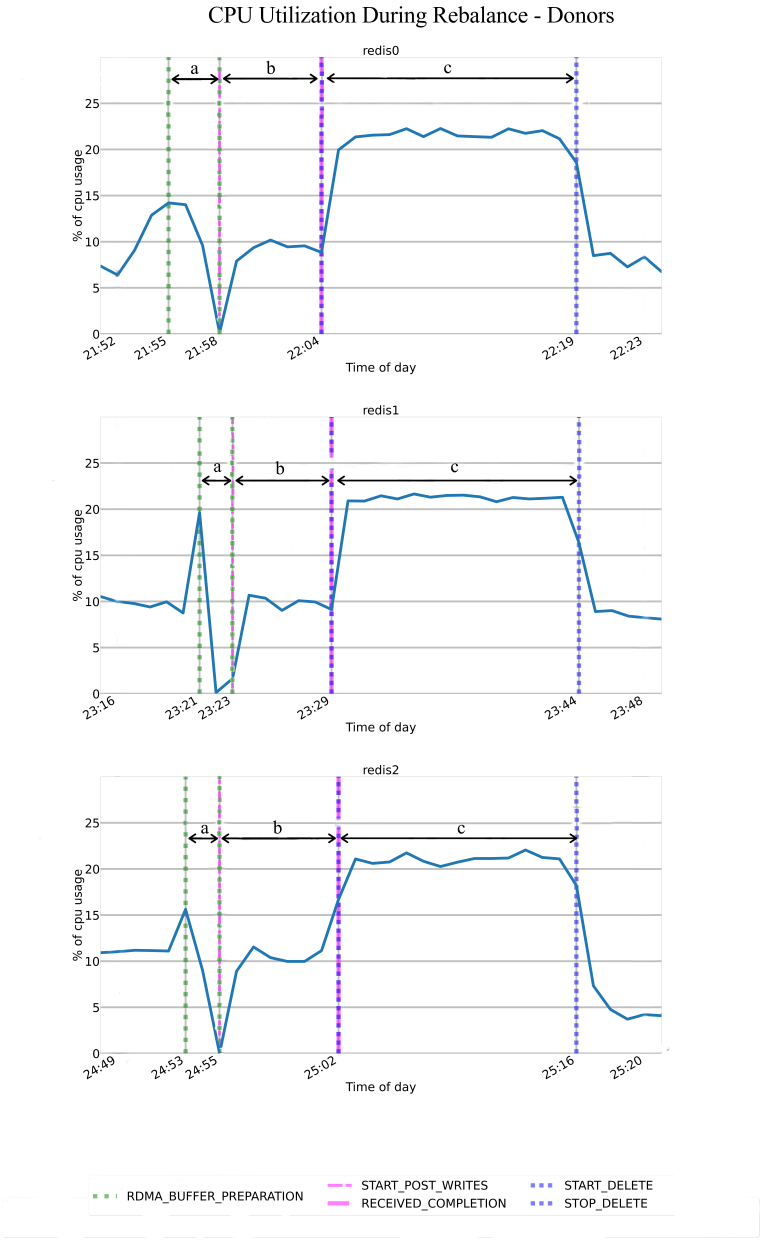

Figure 5.9: Donors CPU utilization during rebalance (Redis-Opt)

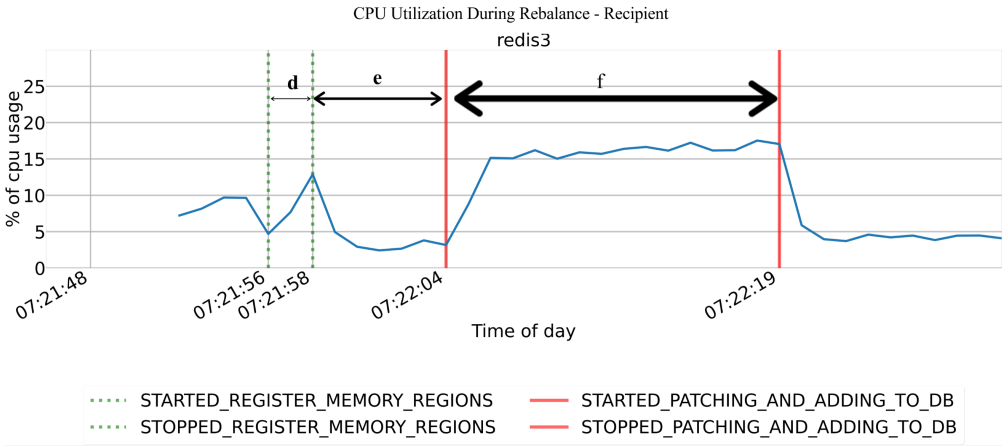

Figure 5.10: Recipient CPU utilization during rebalance (Redis-Opt)

5.3 Discussion

Redis is generally designed as a high performance database, providing high throughput with low latency. In both occasions (Redis-Batch and Redis-Opt) we see that the throughput decreases during rebalancing. In the Redis-Batch version throughput decreases drastically, by about 98.6%, while rebalancing, whereas latency increases on average 8.5 times. For Redis-Opt, throughput decreases by 34% and latency increases 1.67 times on average. Redis-Opt exhibits 2.1 times faster rebalancing than Redis-Batch. As can be seen in Figures 5.3 and 5.4 Redis-Batch during rebalance is not limited by the CPU. The effective bandwidth used during rebalancing is 10MB/sec, which means that the network (capable of 1GB/sec) is also not a bottleneck.

We should expect an increase of throughput when a new node is being added to the cluster and start serving requests for both Redis-Opt and Redis-Batch.

CPU measurements of static cluster of three nodes(while YCSB is producing read-only workload) can be seen in Table 5.1

Table 5.1: Redis Cluster with 3 nodes CPU Usage

Node	***Average CPU Usage***	***STD DEV***
redis0	9.92%	1.20%
redis1	9.90%	1.26%
redis2	10.59%	1.42%

On average 10.1% of each cluster node CPU is being used.

CPU measurements of static cluster(while YCSB is producing read-only workload) of four nodes can be seen in Table 5.2.

On average 8.2% of each redis cluster node CPU is being used. Therefore, Redis Cluster CPU resources are not the bottleneck of our experiment. However

Table 5.2: Redis Cluster with 4 nodes CPU Usage

Node	***Average CPU Usage***	***STD DEV***
redis0	7.74%	0.44%
redis1	8.02%	0.41%
redis2	8.71%	0.60%
redis3	8.56%	0.42%

we have observed that YCSB reaches 100% CPU usage while doing operations. Therefore the reason the throughput is not increasing is that the YCSB workload generator reaches a CPU bottleneck in the client.

Chapter 6

Conclusions and Future work

In this thesis we study the Redis Cluster data rebalancing mechanism and improve its performance via design and implementation of an improved such mechanism that leverages RDMA. The new mechanism improves the bandwidth and latency while also reducing the CPU and memory overhead during transfers. Results show that we have managed to reduce the duration of the overall rebalance action 18 times compared to the off-the-shelf public version of Redis Cluster. Through our mechanism, the nodes participating in rebalance actions can still serve requests, while data transfers are performed at nearly wire speed (1GB/s).

In terms of future work, we aim to further improve our mechanism during rebalancing actions. The current implementation has focused on fully supporting reads from the database during migration; however, it does not yet fully support write operations as they involve a level of complexity that is beyond the scope of this thesis. In our follow-on work, we aim to enable write/update operations on key-value pairs during data migration, fully maintaining consistency.

One possible way to support writes during data migration, is to temporarily buffer all write operations to donors (engaged in data migration to a recipient) in M ephemeral buffer blocks containing key-values of specific (full) slots. When transfer and patching to the recipient dictionary is complete and the ownership of the slots is transferred to the recipient, write operations (now coming into the recipient) can be buffered on the recipient in N temporary blocks of slots. A donor will transfer all its temporary blocks M to the recipient and apply them (either update or add new key-value pairs to the dictionary). When this is complete, the key-value pairs from the N temporary blocks will be added to the dictionary, as we know that writes buffered in M precede writes buffered in N. From now on, new writes will be applied directly to the dictionary. Data is written to M and N temporary blocks to ensure that we respect order. We are working on extending our design and implementation in this direction in our ongoing and future work.

Bibliography

[1] What is RDMA . (Accessed Mar 2022). URL: https://community.mellanox.com/s/article/What-is-RDMA.

[2] Cyclic Redundancy check. (Accessed Feb 2022). URL: https://en.wikipedia.org/wiki/Cyclic_redundancy_check.

[3] Hopscotch Hashing . (Accessed Mar 2022). URL: https://en.wikipedia.org/wiki/Hopscotch_hashing.

[4] Zipfian Distribution https://en.wikipedia.org/wiki/Zipf_law. (Accessed Mar 2022). URL: https://en.wikipedia.org/wiki/Zipf%27s_law.

[5] Jedis Client. (Accessed Mar 2022). URL: https://github.com/redis/jedis.

[6] Lettuce Client. (Accessed Mar 2022). URL: https://lettuce.io/.

[7] MPStat command . (Accessed Mar 2022). URL: https://man7.org/linux/man-pages/man1/mpstat.1.html.

[8] Redis MIGRATE RPC . (Accessed Mar 2022). URL: https://redis.io/commands/migrate.

[9] Redis Website https://redis.io. (Accessed Feb 2022). URL: https://redis.io/.

[10] Redis Cluster Specification. (Accessed Feb 2022). URL: https://redis.io/topics/cluster-spec.

[11] Redis Persistence . (Accessed Mar 2022). URL: https://redis.io/topics/persistence.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1807128.1807152.

[13] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, 2014. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevic.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 54–70, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2815400.2815425.

[15] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. Low-latency communication for fast dbms using rdma and shared memory. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1477–1488, 2020. doi:10.1109/ICDE48307.2020.00131.

[16] Philip Werner Frey and Gustavo Alonso. Minimizing the hidden cost of rdma. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 553–560, 2009. doi:10.1109/ICDCS.2009.32.

[17] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. *SIGCOMM Comput. Commun. Rev.*, 44(4):295–306, aug 2014. doi:10.1145/2740070.2626299.

[18] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with Two-Sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, November 2016. USENIX Association. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia.

[19] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. SOSP '17, page 390–405, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132784.

[20] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 137–152, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132756.

[21] Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, Margo Seltzer, Jeff Chase, Drew Gallatin, Richard Kisley, Rajiv Wickremesinghe, and Eran Gabber. Structure and Performance of the Direct Access File System (DAFS). In *2002 USENIX Annual Technical Conference (USENIX ATC'02)*, Monterey, CA, June 2002. USENIX Association.

[22] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA reads to build a fast, CPU-Efficient Key-Value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, San Jose, CA, June 2013. USENIX Association. URL: `https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell`.

[23] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3), aug 2015. `doi:10.1145/2806887`.

[24] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In Friedhelm Meyer auf der Heide, editor, *Algorithms — ESA 2001*, pages 121–133, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[25] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. Wimpy nodes with 10GbE: Leveraging One-Sided operations in Soft-RDMA to boost memcached. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 347–353, Boston, MA, June 2012. USENIX Association. URL: `https://www.usenix.org/conference/atc12/technical-sessions/presentation/stuedi`.

[26] Hari Subramoni, Ping Lai, Miao Luo, and Dhabaleswar K. Panda. Rdma over ethernet — a preliminary study. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–9, 2009. `doi:10.1109/CLUSTR.2009.5289144`.

[27] T. Talpey and C. Juszczak. Network File System (NFS) Remote Direct Memory Access (RDMA) Problem Statement. RFC 5532, RFC Editor, May 2009.

[28] Wenhui Tang, Yutong Lu, Nong Xiao, Fang Liu, and Zhiguang Chen. Accelerating Redis with RDMA Over InfiniBand. In Ying Tan, Hideyuki Takagi, and Yuhui Shi, editors, *Data Mining and Big Data*, pages 472–483, Cham, 2017. Springer International Publishing.