# An In-Memory RDMA-Based Architecture for the Hadoop Distributed Filesystem

Master Thesis

Konstantinos Karampogias

August 21, 2012

**Supervisor**: **Prof. Dr. Bernhard Plattner**
**Advisors**: **Dr. Xenofontas Dimitropoulos**
**Dr. Patrick Stuedi**
**Dr. Patrick Droz**

Computer Engineering and Networks Laboratory, ETH Zurich
IBM Research Zurich

**Abstract**

In the past years, there has been a growing interest in realtime and low-latency data processing systems for the cloud. To accommodate those emerging demands, the underlying data storage systems should be fundamentally designed to make better use of the data center's powerful hardware. In this master thesis, we propose a redesign of the hadoop distributed file system (HDFS), tailored for in-memory storage and remote direct memory access (RDMA). Specifically, the original HDFS was modified to operate natively in memory, when data can fit into the cluster memory, and to adopt the RDMA communication model, which offers low latency and better CPU utilization. The proposed solution has been designed for large data volumes that cannot be held in physical memory. Therefore, the system operates on virtual memory, keeping only the hot data in physical memory at any point in time. As opposed to traditional RDMA-based systems, our modified HDFS runs in a cluster of commodity hardware using a software-based RDMA implementation. In this thesis, we evaluated the system in a real cluster versus the original HDFS version. The results show improved latencies for filesystem reads and writes as well as lower CPU usage across the datacenter nodes running HDFS.

**Keywords:** RDMA, HDFS, in-memory, big data, Hadoop, MapReduce

# Contents


**1 Introduction** **5**
- 1.1 Motivation . . . 5
- 1.2 Problem Statement . . . 6
- 1.3 Related work . . . 7
- 1.4 Structure . . . 7

**2 Background** **8**
- 2.1 Hadoop . . . 8
  - 2.1.1 MapReduce . . . 9
  - 2.1.2 HDFS . . . 10
- 2.2 RDMA . . . 14
  - 2.2.1 Benefits over Sockets . . . 14
  - 2.2.2 Remote Direct Memory Access . . . 14
  - 2.2.3 InfiniBand, iWARP, RoCE, SoftiWARP and RDMA Verbs . . . 15
  - 2.2.4 RDMA Data Transfer Operations . . . 15
  - 2.2.5 RDMA Queue-Based Programming Model . . . 16

**3 Design of an In-Memory RDMA-based HDFS** **18**
- 3.1 RDMA Semantics . . . 18
- 3.2 Hadoop Network I/O . . . 19
- 3.3 In-Memory RDMA-Enabled Architecture . . . 20
- 3.4 Hidden Costs of RDMA . . . 21
  - 3.4.1 Connection Establishment Cost . . . 22
  - 3.4.2 Buffer Registration Cost . . . 22
- 3.5 One-Sided Vs Two-Sided Operations . . . 23
- 3.6 Scaling Beyond Physical Memory . . . 24
- 3.7 Replication Mechanism . . . 24

**4 Implementation** **25**
- 4.1 Project Info . . . 25
- 4.2 Anatomy of an RDMA Node . . . 25
- 4.3 Removing Hard Disk from the Critical Path . . . 28
- 4.4 Refactoring HDFS Write Operation . . . 28
- 4.5 Anatomy of a Write . . . 28
- 4.6 Anatomy of a Read . . . 32

**5 Evaluation** **33**
- 5.1 Validation . . . 33
- 5.2 Performance Benchmarks . . . 34
  - 5.2.1 Setup . . . 34

5.2.2 Monitoring Methodology . . . . . 35
5.2.3 Uploading a Big File in the Cluster . . . . . 36
5.2.4 Micro-Benchmarks . . . . . 38
5.2.5 Scalability . . . . . 43
5.2.6 Read under Saturation . . . . . 44

**6 Conclusion** **45**
6.1 Summary . . . . . 45
6.2 Future work . . . . . 45

**Bibliography** **47**

# List of Figures


1.1 Target System . . . . . 6

2.1 HadoopEcosystem . . . . . 9
2.2 A MapReduce Computation . . . . . 10
2.3 HDFS Namenode . . . . . 11
2.4 HDFS Write Operation . . . . . 12
2.5 HDFS Read Operation . . . . . 13
2.6 Socket-Based Communication Model . . . . . 14
2.7 RDMA Communication Model . . . . . 14
2.8 RDMA Verb Providers . . . . . 15
2.9 RDMA Operations . . . . . 16
2.10 Queue-Based Model . . . . . 17

3.1 Hadoop Networking Phases . . . . . 19
3.2 RDMA-Enabled Namenode . . . . . 20
3.3 RDMA-Enabled Datanode . . . . . 20
3.4 RDMA-Enabled Architecture . . . . . 21

4.1 RDMA Node . . . . . 27
4.2 Normal HDFS Write . . . . . 29
4.3 RDMA-Enabled HDFS Write . . . . . 29
4.4 Sequence Diagram for RDMA-Enabled HDFS Write with replication . . . . . 31
4.5 Sequence Diagram for RDMA-Enabled HDFS Read . . . . . 32

5.1 Testbed Setup . . . . . 34
5.2 Example CPU Usage Graph during a WordCount Task . . . . . 36
5.3 CPU Usage, Write/Read 2048 MB File in a 3DN Cluster (=64blocks of 32MB) . . . . 37
5.4 CPU Usage, Write/Read 2048 MB File in a 1DN Cluster (=64blocks of 32MB) . . . . 38
5.5 Datanode CPU/Net Usage, Write Operation of 2048 MB File in a 1DN Cluster . . . . 38
5.6 Write Latency Histograms . . . . . 40
5.7 Write Delay Details (128MB File in 32MB block in 8MB packets) . . . . . 40
5.8 Read Latency Histograms . . . . . 41
5.9 Read Delay Details (128MB File in 32MB block) . . . . . 41
5.10 Scalability: Latency Vs Number of Clients . . . . . 43
5.11 Datanode Performance under Saturated 10Gbps Network . . . . . 44

# Chapter 1

# Introduction

## 1.1 Motivation

The big data paradigm is entering a new phase. In the past, data processing systems were designed to store and process huge data sizes for which the traditional relational database management systems (RDMS) would fail. These systems were used mainly as batch systems with a typical running time of hours or even days. Today, data analysis systems should be architected not only to process the increasingly large volume of data, but also to process it fast, possibly with a response time in the range of minutes or seconds. The response time has become a key factor as a new class of cloud applications, such as real time analytics or interactive applications, have emerged.

We are motivated that faster and more efficient distributed systems can be built by a profound software redesign that will take advantage of the data center's powerful hardware. In particular, systems should make better use of the cluster memory and the multi-gigabit network fabric, while at the same time, any slow storage device should be removed from the critical path. We believe that holding and analyzing datasets in memory will be a good option in terms of performance, and therefore, systems should be designed according to that principle. Using cluster memory instead of the slower hard disk as the prime storage has been already discussed in the RamCloud project [1]. RamCloud supports that memory will be inexpensive enough, and in the future the total cluster memory will offer adequate storage capabilities. Furthermore, there is a wide range of applications that will benefit from an in-memory system. First, a significant percentage of the jobs which are submitted to production analytics clusters have input data set sizes that can fit into cluster memory [2]. Second, there is an important class of applications that reuse the same data [3], or process them incrementally [4]; for example, interactive applications that require users to perform a sequence of queries over a specific dataset or applications that implement iterative algorithms. Third, jobs typically consist of a number of phases (chain jobs), which produce intermediate results that should be instantly consumed by the next phase. Storing these results in memory will significantly boost the overall performance. Finally, future solid-state disks might be used in the same way as memory is currently being used and therefore, they will further expand the total storage capability. For the above reasons, we believe that a fundamental redesign of a big-data scale system in order to also work natively in memory would have an impact and should be investigated.

Towards building an in-memory system, the remote direct memory access (RDMA) communication model can be of great benefit. RDMA technology originates from high-performance computing (HPC), and has been proven to offer low latency, better CPU utilization, as well as richer communication semantics. Additionally, RDMA will enable the system to effectively saturate the new multi-gigabit network fabrics [5] and is expected to increase the power efficiency of the data center [6]. RDMA in the past required specific hardware limiting its deployment only in application inside HPC world. With the advent of a fully software-based implementation of RDMA (SoftiWARP [7]), the technology can be extended to applications running in commodity hardware [8].

## 1.2 Problem Statement

There is a large number of available distributed systems for large-scale data analysis and storage. Among these systems, Hadoop ecosystem [9] is being widely considered to be the de-facto solution to handle big data. Hadoop is an Apache open source project which provides a highly scalable distributed file system (HDFS), a powerful programming model for parallel computation (MapReduce) and a plethora of other analysis tools. In this thesis, an in-memory, RDMA-enabled Hadoop was implemented. In more detail, the Hadoop distributed file system was redesigned to store the data in the memory and to adopt RDMA semantics replacing the existing socket-based communication patterns.

The goal of this master thesis was to build a robust in-memory Hadoop system, which through the RDMA communication model, will extend its efficiency, performance and capabilities. In **Figure 1.1**, the expected performance of the system is shown. In comparison to the current framework, we expect our system to operate as efficiently as a native in-memory system when the data can be stored in memory, to show the same performance as the RDMS systems when the data is growing, and to be able to scale with a better performance when the data volume reaches big data scale.

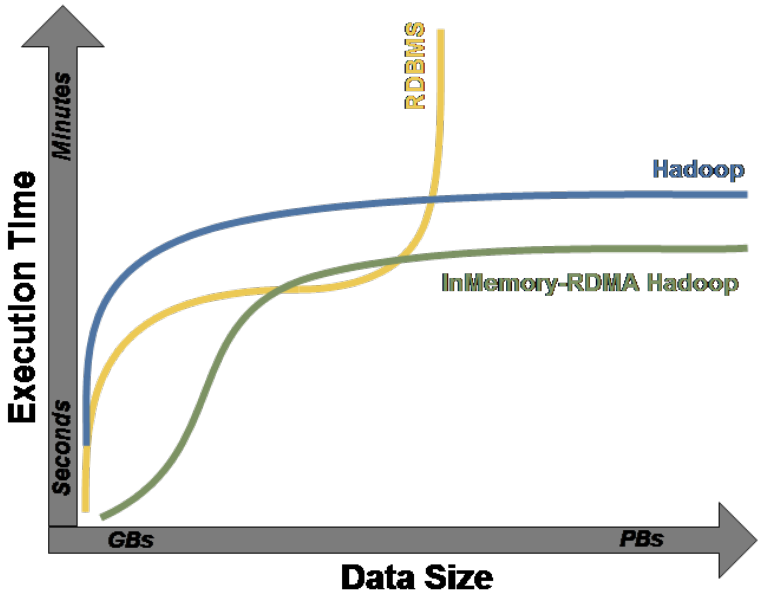



**Figure 1.1:** Target System

Additionally, as a result of our design, we expect to improve Hadoop in several aspects. For example, HDFS is not optimized for random reads and does not support file updates. In our implementation, random read performance should be better and file update feature might be easier to be added. Furthermore, Hadoop does not perform well in a virtual cluster, mainly because storage devices are not attached to the nodes and thus the system cannot be benefited by data locality. In our implementation, data locality is maintained since data is stored in memory, which is attached to the node. What is more, according to [10], Hadoop's interaction with local filesystems imposes a number of hidden bottlenecks which do not exist in our implementation. Finally, our system is ready to run natively on HPC environments in extreme performance.

## 1.3 Related work

Improving a large-scale data analysis system is an interesting topic in both academia and industry, and therefore, there is significant work towards that direction.

**In-memory distributed storage and process:** Building an in-memory system is not a new idea and there are a number of implementations. For example, Spark system [3], which implements a distributed memory abstraction, called Resilient Distributed Datasets (RDDs) and enables in-memory computations on large clusters. Application specific systems such as Pregel, which supports iterative graph applications, or HaLoop and Twister, which optimize iterative algorithm programs. Other systems, such as Piccolo, DSM and RamCloud, which expose a storage system entirely built from DRAM and allow the user to perform in-memory computation. Our work mainly differs because we modified an existing big-data framework to maintain its programming broad model abstraction, to natively work in memory and to remain capable of scaling to large volume size.

**Alternative RDMA-based distributed storage:** Hadoop framework is an important distributed large-scale data analysis system. For that reason, developers of the most distributed file systems have a great incentive to make their implementation compatible with Hadoop. Therefore there are a number[1] of filesystems that can replace HDFS. For example, GPFS [11], Glusterfs [12], Lustre [13]. Among these filesystems, some are RDMA-enabled, such as Glusterfs. However, HDFS is being developed in parallel with the rest of the Hadoop ecosystem. Therefore, it is tailored to its requirements, which might not be the same as those of other filesystems.

Finally, no significant work has been carried out to integrate RDMA semantics in Hadoop. The reason is that Hadoop runs on commodity servers and RDMA in the past was available only through expensive hardware. Nevertheless, an optimization has been proposed in [14], where the MapReduce shuffle step was optimized using RDMA semantics.

## 1.4 Structure

The thesis is organized as follows: the first part of chapter 2 gives a background overview of Hadoop framework with focus on the underlying distributed file system (HDFS). The second part presents the RDMA communication model. Chapter 3 explains how the in-memory, RDMA-enabled HDFS was designed. In chapter 4, the implementation is described in more detail, and critical points are analysed. In chapter 5, the evaluation of our implementation is given along with some discussion of the results. The final chapter concludes and suggests potential future work.

[1] http://gigaom.com/cloud/because-hadoop-isnt-perfect-8-ways-to-replace-hdfs/

# Chapter 2

# Background

The first part of the background chapter presents an overview of Hadoop framework and its major components. A detailed description can be found in [15, 9].

## 2.1 Hadoop

Hadoop is an open-source framework for writing and running distributed applications that process very large data sets. There has been a great deal of interest in the framework, and it is very popular in industry as well as in academia. Hadoop cases include: web indexing, scientific simulation, social network analysis, fraud analysis, recommendation engine, ad targeting, threat analysis, risk modeling and other. Hadoop is core part of a cloud computing infrastructure and is being used by companies like Yahoo, Facebook, IBM, LinkedIn, and Twitter. The main benefits of Hadoop framework can be summarized as follows:

- Accessible: it runs on clusters of commodity servers
- Scalable: it scales linearly to handle larger data by adding nodes to the cluster
- Fault-tolerant: it is designed with the assumption of frequent hardware failures
- Simple: it allows user to quickly write efficient parallel code
- Global: it stores and analyzes data in its native format

Hadoop is designed for data-intensive processing tasks and for that reason it has adopted a "move-code-to-data" philosophy. According to that philosophy, the programs to run, which are small in size, are transferred to nodes that store the data. In that way, the framework achieves better performance and resource utilization. In addition, Hadoop solves the hard scaling problems caused by large amounts of complex data. As the amount of data in a cluster grows, new servers can be incrementally and inexpensively added to store and analyze it.

Hadoop has two major subsystems: the Hadoop Distributed File System (HDFS) and a distributed data processing framework called MapReduce. Apart from these two main components, Hadoop has grown into a complex ecosystem, including a range of software systems. Core related applications that are built on top of the HDFS are presented in **Figure 2.1** and a short description per project is given in **Table 2.1**.

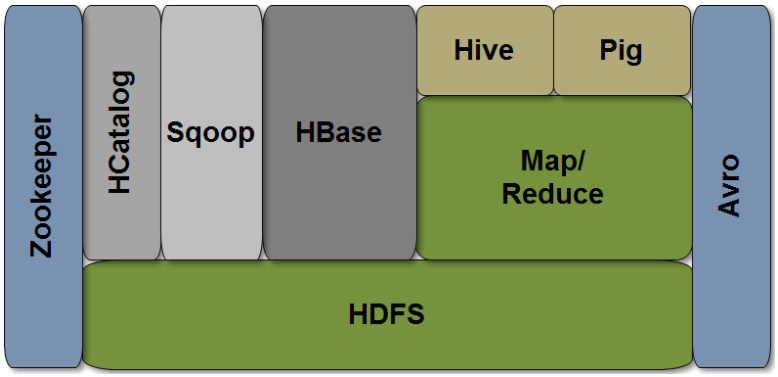



**Figure 2.1:** Hadoop Ecosystem

| Project | Info |
|---|---|
| HDFS | Distributed File System |
| MapReduce | Distributed computation framework |
| ZooKeeper | High-performance collaboration service |
| HBase | Column-oriented table service |
| Pig | Dataflow language and parallel execution |
| Hive | Data warehouse infrastructure |
| HCatalog | Table and storage management service |
| Sqoop | Bulk data transfer |
| Avron | Data serialization system |

**Table 2.1:** Project Descriptions

### 2.1.1 MapReduce

MapReduce [16] is a programming model developed for large-scale analysis. It takes advantage of the parallel processing capabilities of a cluster in order to quickly process very large datasets in a fault-tolerant and scalable manner. The core idea behind MapReduce is mapping the data into a collection of key/value pairs, and then reducing over all pairs with the same key. Using key/value pairs as its basic data unit, the framework is able to work with the less-structured data types and to address a wide range of problems. In Hadoop, data can originate in any form, but in order to be analyzed by MapReduce software, it needs to eventually be transformed into key/value pairs.

Hadoop implements MapReduce programming model using two components: a JobTracker (master node) and many TaskTrackers (slave nodes). The JobTracker is responsible[1] for accepting job requests, for splitting the data input, for defining the tasks required for the job, for assigning those tasks to be executed in parallel across the slaves, for monitoring the progress and finally for handling occurring failures. The TaskTracker executes tasks as ordered by the master node. The task can be either a map (takes a key/value and generates another key/value) or a reduce (takes a key and all associated values and generates a key/value pair). The map function can run independently on each key/value pair, enabling enormous amounts of parallelism. Likewise, each reducer can also run separately on each key enabling further parallelism.

When a job is submitted to the JobTracker, the JobTracker selects a number of TaskTrackers

[1] the new architecture MRv2 or YARN divides the function of JobTracker into two nodes ResourceManager and ApplicationManager, more detail in `http://hadoop.apache.org/common/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/YARN.html`

(not randomly but according to data locality) to execute a map task (Mappers) and a number of TaskTrackers to execute a reduce task (Reducers). The job input data is divided into splits and is organized as a stream of keys/values records. In each split there is a matching mapper which converts the original records into intermediate results which are again in the form of key/value. The intermediate results are divided into partitions (each partition has a range of keys), which after the end of the map phase are distributed to the reducers (shuffle phase). Finally reducers apply a reduce function on each key. A MapReduce paradigm is given in **Figure 2.2**. MapReduce is designed to continue to work in the face of system failures. When a job is running, MapReduce monitors progress of each of the servers participating in the job. If one of them is slow in returning an answer or fails before completing its work, MapReduce automatically starts another instance of that task on another server that has a copy of the data. The complexity of the error handling mechanism is completely hidden from the programmer.

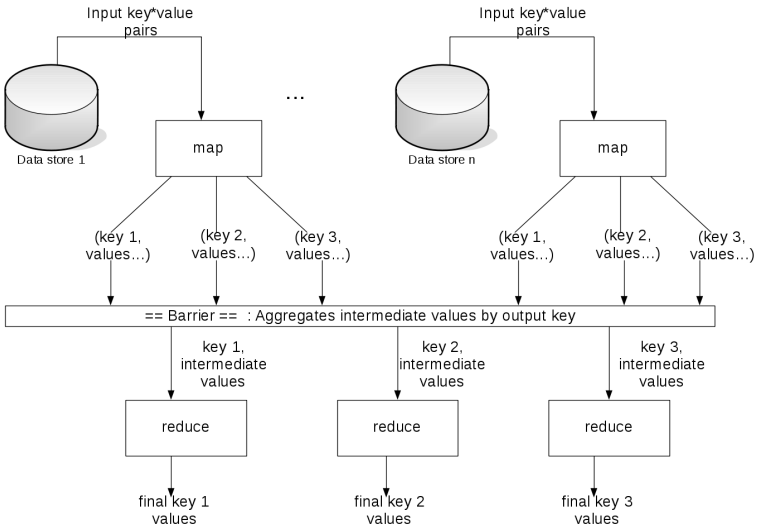



**Figure 2.2:** A MapReduce Computation, Image from [17]

### 2.1.2 HDFS

Hadoop comes with a distributed filesystem called HDFS [18], which stands for Hadoop Distributed File System. HDFS manages the storage of files across the cluster and enables global access to them. In comparison to other distributed filesystems, HDFS is on purpose implemented to be fault-tolerant, and to conveniently expose the location of data enabling Hadoop framework to take advantage of the "move-code-to-data" philosophy.

HDFS is optimized for storing large files, for streaming the files at high bandwidth, for running in commodity hardware. While HDFS is not performing well when the user application requires low-latency data access, when there are a lot of small files, and when the application requires arbitrary file modification. The filesystem is architected using the pattern write-once, read-many-times (simple coherency model). A dataset is generated once, and multiple analyses are performed on it during time. It is expected that the analysis will require the whole dataset, hence fully reading the dataset is more important than the latency of a random read.

In HDFS, files are broken into block-sized chunks, which are independently distributed in a number of nodes. Each block is saved as a separate file in the node's local filesystem. The size of the block is large and a typical value would be 128MB, but it is a value chosen per client and per file. The large size of the block was picked, firstly, in order to take advantage of sequential I/0 capabilities of disks,

secondly to minimize latencies because of random seeks and finally because it is logical input size to the analysis framework.

HDFS is also designed to be fault tolerant, which means that each block should remain accessible in the occur of system failures. The fault-tolerance feature is implemented through a replication mechanism. Every block is stored in more than one node making highly unlikely that it can be lost. By default, two copies of the block are saved on two different nodes in the same rank and a third copy in a node located in a different rank. The HDFS software continually monitors the data stored on the cluster and in case of a failure (node becomes unavailable, a disk drive fails, etc.) automatically restores the data from one of the known good replicas stored elsewhere on the cluster.

### Namenode

The namespace of the filesystem is maintained persistently by a master node called namenode. Along with the namespace tree, this master node holds and creates the mapping between file blocks and datanodes. In other words, namenode knows in which node the blocks of a file are saved (physical location). The mapping is not saved persistently because it is reconstructed from the datanodes during start phase, and because it dynamically changes over time. The internal structure of a namenode is given in **Figure 2.3**.

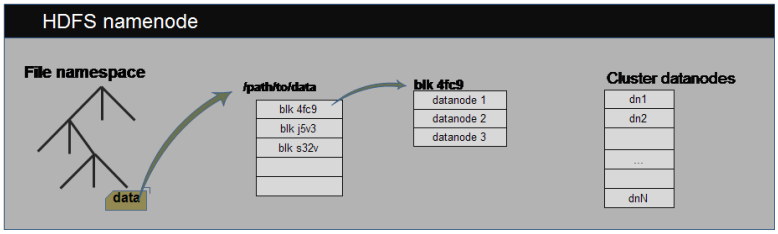



**Figure 2.3:** HDFS Namenode

Holding the critical information (namespace, file metadata, mapping) in a unique master node rather node than in a distributed way makes the system simpler. However, at the same time it makes this node a single point of failure (SPOF) and creates scalability issues. From performance point of view, it is required that the namenode holds the namespace entirely in RAM and that it will respond fast to a client request. For reliability, it is necessary for the namenode to never fail, because the cluster will not be functional. The information that is stored in the namenode should also never be lost, because it will be impossible to reconstruct the files from the blocks. For the above reasons, great effort has been made in order to maintain the master node, but at the same time to overcome the drawbacks. In past versions, it was a common tactic to backup the information either in a remote NFS mount or using a secondary namenode in which the namespace image was periodically merged and could replace the original in case of failure. Nevertheless, these solutions were not ideal and two new architectures (HDFS Federation and HDFS High-Availability) were introduced in the 2.x Hadoop alpha release series to fundamentally resolve the above issues.

### Datanode

As has been implied before, the blocks of a file are independently stored in nodes, which are called datanodes. Every datanode in the cluster, during startup, makes itself available to the namenode through a registration process. Apart from that, each datanode informs namenode which blocks has in its possession by sending a block report. A block reports are sent periodically or when a change takes place. Furthermore, every datanode sends heartbeat messages to the namenode to confirm that it remains operational and that the data is safe and available. If a datanode stops operating, there

are error mechanisms in order to overcome the failure and maintain the availability of the block. Heartbeat messages also hold extra information, which helps the namenode run the cluster efficiently (e.g. storage capacity which enables namenode to make load balancing). One important architectural note is that namenode never directly calls datanodes; it uses replies to heartbeats to order a command to be executed in the datanode (e.g. to transfer a block to another node).

## HDFS Client

User applications or Hadoop components (mappers or reducers) can access a file stored in the filesystem through the HDFS Client, a library which exports the interface of the underlying distributed filesystem. Like in other filesystems, HDFS supports all the basic operations such as reading files, writing files, creating directories, moving files, listing directories, deleting data[2]. HDFS does not support updating a file, only appending it. In all operations, the client contacts first the namenode and in case of a data transfer, the client directly contacts the datanode to perform the transfer.

For read/write operations, clients of the HDFS simply use a standard Java input/output stream, as if the file was stored locally. Under the hood, the streams are modified to retrieve the data from the cluster in a transparent way. A summary of the two main operations as they have been described in [15] follows.

### Write Operation

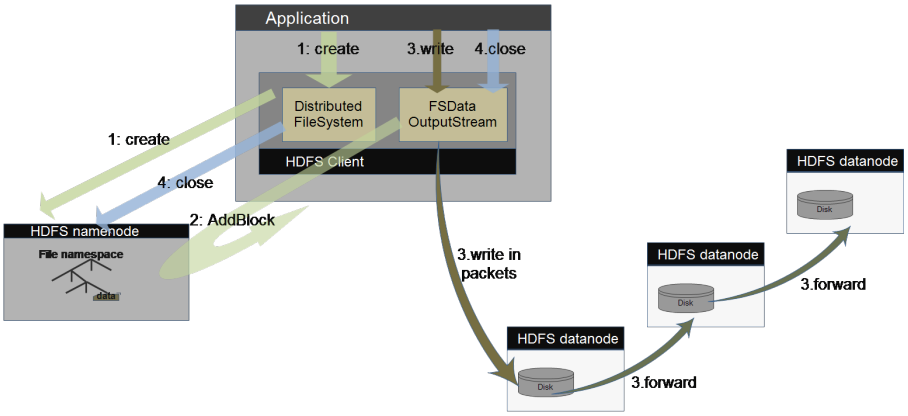



**Figure 2.4:** HDFS Write Operation

The overall procedure of the write operation is given in **Figure 2.4**. The client creates the file by calling create method on a DistributedFileSystem (DFS) object, then the DFS contacts the namenode and asks a new file to be created in the filesystem's namespace. If the call succeeds, the client gets a special output stream called FSDataOutputStream responsible for communicating with the namenode and transferring the data to the datanodes. As the client writes data to the FSDataOutputStream, it splits the input into packets, which are enqueued in an internal queue (DataQueue). The packets are consumed by another thread (DataStreamer), which is responsible for asking the namenode to allocate new blocks, for receiving the target nodes where the blocks will be stored, for setting up the pipeline and finally for forwarding the first packet into the first datanode. The first datanode forwards the packet to the second one, the second one to the next one until the pipeline is finished. In the same time, FSDataOutputStream maintains a second internal queue (Ack Queue). In that queue packets are entered when sent (moved from the DataQueue) and removed when successfully acknowledged by the datanodes. When the client finishes writing data, it calls close() on the stream and waits for all

[2] a full list is given by executing the command "hadoop fs -help"

packets to be acknowledged. Then it makes another call to the namenode to inform that the file is completed. The namenode waits for blocks to be minimally replicated[3] before returning successfully.

If there is a failure during the writing, HDFS provides mechanisms to recover and to successfully succeed the writing process. What is happening behind the scenes is transparent to the client application, which will be only informed in extreme unrecoverable cases (e.g., run out of resources).

### Read Operation

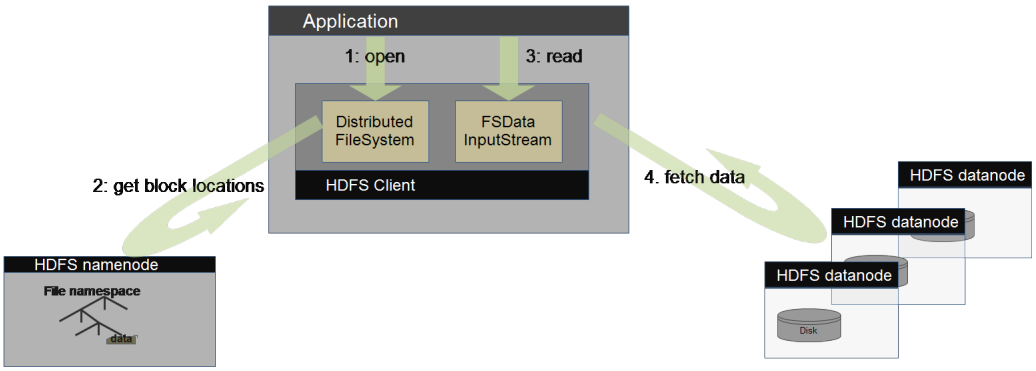



**Figure 2.5:** HDFS Read Operation

The client opens the file it wants to read by executing open method in a DistributedFileSystem(DFS) object. The DFS contacts namenode and asks for the locations of the blocks. For each block, the namenode returns the addresses of the datanode sorted according to their network proximity towards the client[4]. The DFS returns a FSDataInputStream to the client from which it can read the data.

When the client calls read method on the stream, the stream connects to the datanode holding the first block. Data is streamed from the datanode to the client, which calls the read method repeatedly. When the end of the block is reached, the stream will disconnect from the datanode, and will connect to the next datanode for the next block. This is happening internally in the stream and is completely transparent to the client which just reads from a stream.

[3] the blocks will be asynchronously replicated to its replication factor afterwards
[4] same node, same rank, same data center, different data center

## 2.2 RDMA

The second part of the background chapter presents a short overview of the Remote Direct Memory Access (RDMA) networking model. An extended and more complete overview of RDMA can be found in [19, 20].

### 2.2.1 Benefits over Sockets

In the normal TCP/IP communication socket based model, there are some limitations with respect to multi-gigabit interconnects. Since TCP/IP performance has been well studied, in this subsection we describe only one issue which is an important performance bottleneck and which constitutes a fundamental argument in favor of switching to a better communication model.

This issue is the indirect data placement imposed by the socket abstraction as shown in **Figure 2.6**. When the application transmits data, the data is copied by the CPU from an application buffer (user space) into a temporary socket buffer (kernel space). Then the TCP/IP stack uses the CPU to create an appropriate TCP packet by adding the header. Finally, the packet is brought to the network interface controller (NIC) by a DMA copy. On the receive path, the inverse procedure takes place. The copy from userspace to kernelspace causes an overhead to the CPU, and this overhead is function of the bandwidth the application communicates with. As it has been shown [21] the CPU overhead on the end-hosts in order to transmit/receive data in 10 Gbps is by far non-negligible and primarily is due to excessive data copy. As a result, the current communication model poses a performance as well as an efficiency bottleneck in applications which require significant amount of data transfers and which are interconnected by a multi-gigabit network fabric.

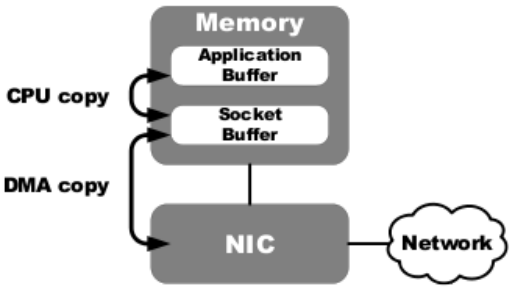



**Figure 2.6:** Socket-Based Communication Model, Image from [19]

### 2.2.2 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is a communication mechanism which allows a system to place the data directly into its final location in the memory of the remote system without any additional data copy (**Figure 2.7**).

This feature is called zero-copy or direct data placement (DDP) and enables efficient networking. An RDMA-enabled network interface controller (RNIC) provides to the operating system an RDMA-ed network stack which enables direct data transfers from the local memory of one host to the memory of the remote host through. The RDMA model differs significantly from the classical socket-based model. For example when a data transfer takes place, the following procedure is followed: applications in both end register to their RNICs which memory location (address and length) should be exchanged, then the RNIC fetches the data from that buffer without CPU involvement using DMA and transfers them across the network. The receiving RNIC places the inbound data directly to the final location at a destination buffer of the application.

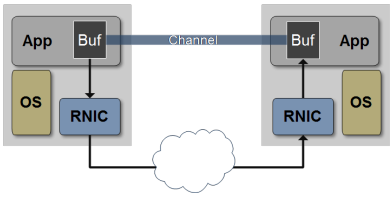



**Figure 2.7:** RDMA Communication Model

### 2.2.3 InfiniBand, iWARP, RoCE, SoftiWARP and RDMA Verbs

InfiniBand is a high speed network fabric used in high-performance computing and enterprise data centers. Nodes connected to that fabric require to have special host channel adapters (HCAs) which are the equivalent of a NIC in an ethernet network. Infiniband originally supported RDMA communication model through a queue-based programming model. This queue-based model is offered by the HCA to the applications through a verb interface. Internet Wide Area RDMA Protocol (iWARP) is a set of protocols on top of the IP layer which enable a network interface card to directly place data into destination buffers without CPU involvement. iWARP provides also a verb interface which is a subset of the InfiniBand verb interface. iWARP development was focus on TCP/IP transport protocol and hence it became synonymous to RDMA over TCP/IP. Nevertheless, there is also RDMA over Converged Ethernet (RoCE) which is an RDMA implementation that operates exactly over the ethernet layer. All the above implementations required specific hardware. SoftiWARP on the other hand is a software implementation over TCP/IP, which enables a commodity ethernet NIC to handle RDMA traffic in software. In essence, every RDMA-enabled NIC has to provide the RDMA Protocol Verbs interface and for that reason is called RDMA verb provider. In **Figure 2.8**, a visual summary of the existing possible RDMA providers is given. From an application perspective, application interacts through the verb interface, and should be designed to follow the queue-based RDMA networking model.

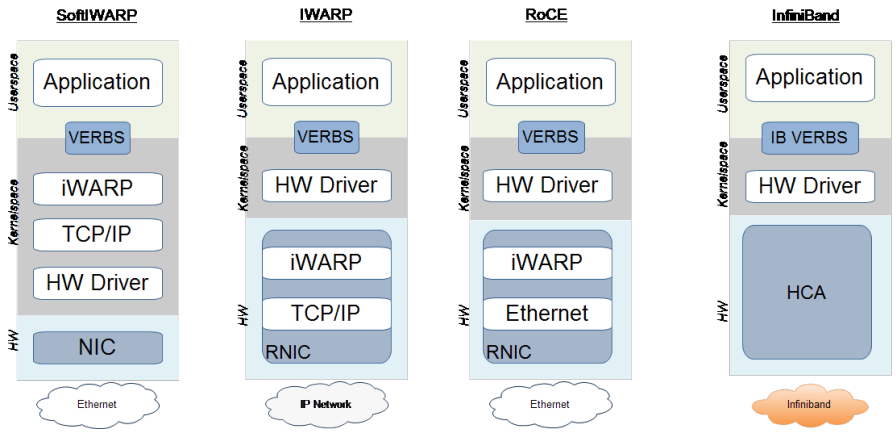



**Figure 2.8:** RDMA Verb Providers

### 2.2.4 RDMA Data Transfer Operations

An application using the RDMA networking concept can perform one of the following operations (**Figure 2.9**): send, receive, RDMA read and RDMA write. The first two (send and receive) are called two-sided operations because require both ends to actively be involved in the transfer. Specifically, every send operation should have a matching receive operation posted by the application at the remote side. Before a send operation takes place, the client is required to have registered to the RNIC the application buffer from which the data should be sent. In the receive operation, the client should have also registered the memory location where the incoming data should be placed.

RDMA applications can also perform a direct RDMA read or write operation. As the name implies, RDMA read operation copies the data from a buffer located in the remote memory into a local buffer, whereas RDMA write does the opposite. These operations are called one-sided operation because only the application that issues the operation is involved in the data transfer. At the remote side, the operation is served exclusively by the RDMA transport provider without application knowledge. In contrast to two-sided operation, application now should have a prior knowledge about the target

remote memory location. In more detail, the client should know the target memory address, the length of the data and an identifier of the buffer (called Steering Tag, or STag). Therefore, RDMA one-sided operations require an out-of-band buffer advertisement mechanism before the data transfer takes place.

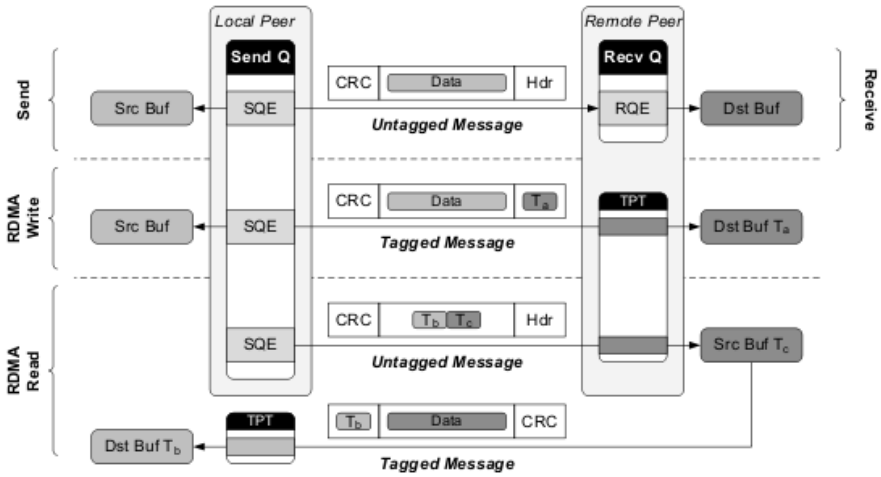



**Figure 2.9:** RDMA Operations, Image from [19]

### 2.2.5 RDMA Queue-Based Programming Model

RDMA communication model differs significantly from the classical socket-based model. It offers a queue-based asynchronous communication model between the application (verbs consumer) and the RNIC (verbs provider). Without going into detail, a brief summary of the model is the following: RDMA communication is based on three queues: send queue (SQ), receive queue (RQ), completion queue (CQ). The SQ and RQ are called work queues (WQs) and they are always created in a pair, referred to as queue pair (QP). One-sided operations along with send operations are posted on the SQ, while the receive operation is posted on the RQ. A completion queue (CQ), which is a container for work completion objects (WC), is attached to a QP and is used to notify application when an RDMA operation placed on that QP has been completed.

When sending data, application creates a send work request (SendWR) and posts this request on it's SQ. The SendWR informs RNIC which buffer should be sent. The RNIC asynchronously fetches the SendWR from SQ and executes the data transfer. On the receiving side, the application has posted a receive work request (ReceiveWR) on the RQ, which contains the memory location where the incoming data should be saved. The RNIC matches each incoming send message to exactly one ReceiveWR. Once an RDMA operation is completed a completion queue element (CQE) is created and placed on the CQ. Application by waiting on CQ channel for completion queue events knows when the data were transfered. The overall procedure is given in the **Figure 2.10** and is explained in detail [21].

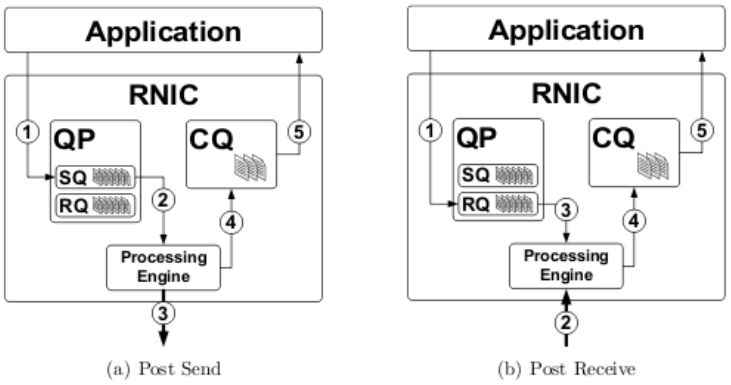



**Figure 2.10:** Queue-Based Model, Image from [19]

# Chapter 3

# Design of an In-Memory RDMA-based HDFS

This chapter describes the design of Hadoop Distributed File System in order to operate as an in-memory system and in order to adopt efficiently the RDMA networking model. The chapter is structured as follows: first, an overview of the RDMA semantics is given; second, Hadoop networking phases are analysed, and third, the in-memory architecture is described. Finally, in the last sections, important topics concerning RDMA, performance and scalability are discussed.

## 3.1 RDMA Semantics

The RDMA networking model provides rich communication semantics, among them the ones which characterize better the model and which were taken into consideration during the implementation are summarized in the **Table 3.1**.

| **Semantic** | **Info** |
|---|---|
| Buffer Management | Explicit communication buffer management: contrary to normal TCP/IP based application, an RDMA enabled application should allocate, register to the RNIC and manage its communication buffer (or Memory Regions). |
| One-Sided Operations | Application should primarily make use of one-sided operation, which require a buffer advertisement mechanism. The receiving side of one-sided operation should not be involved in application logic. For instance, when a client reads a block from a datanode, the datanode should do nothing in application logic. Main reason is that remote side of one-sided operation cannot implicitly be notified of a completion of an operation. |
| Asynchronous Semantics | The application should overlap communication with computation. |
| Control Vs Data Path | Application should carefully separate control path from data path. The data that the application uses should remain stored in a static location in memory. The control messages should only be exchanged between the application units. |
| Other | Semantic aggregation of operations, grouping of work posting, completion notification, RDMA hidden costs. |

**Table 3.1:** RDMA Semantics

## 3.2 Hadoop Network I/O

In this section, we looked into Hadoop network traffic which is a necessary step before integrating a new networking model. Hadoop is a framework running in clusters, and therefore creates significant network traffic. In **Figure 3.1** an overview of the network phases during a complete MapReduce task is given and in **Table 3.2** each phase is explained. As expected, the underlying distributed file system is mainly responsible for the network transfers.

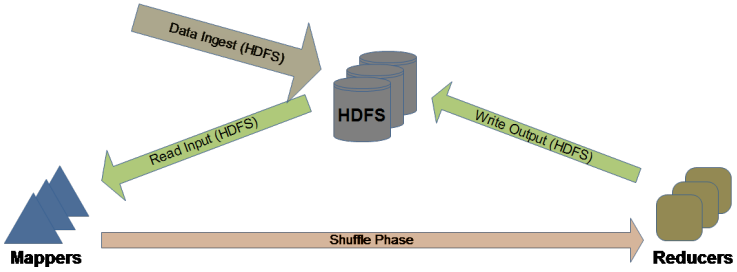



**Figure 3.1:** Hadoop Networking Phases

| Phase | Info |
|---|---|
| Data Ingest (+Replication) | During the initial data loading operation, huge size datasets are pushed into the cluster. Network links can be saturated, and significant portion of the resources should be allocated for the phase. The replication mechanism increases greatly the data size and therefore, the need for resources. In most cases, this phase is not time critical. In an RDMA-enabled HDFS, we expect to lower the CPU footprint, which will result in increasing the computation resources or will decrease the cluster power consumption. |
| Read Input | Due to data-locality philosophy, there is no significant networking traffic during this phase. In an in-memory Hadoop, data-locality implementation should be extended accordingly, so workers to read directly the data from the memory. |
| Shuffle Phase | The networking performance of this phase is crucial because it defines to a great extent the execution time of a MapReduce task. There has been proposed an RDMA optimization [14] which showed a 36% optimization in execution times by using RDMA technology. Nevertheless, this phase does not include HDFS, and hence, it is outside our work. |
| Write Output | Contrary to the data ingest, this step usually involves smaller datasets (size depends on the application) and it is time critical. In normal cases, it requires low latency because the job submitter waits for a reply or in a chain job, the output would be the input for the next step. In our implementation, we expect to improve the performance of this phase. |
| Administration | In a big cluster, there are many administrative tasks that require data transfer and demand resources. For example,"rebalancing" case in which the namenode takes care that data is uniformly stored in the cluster. This step is not time critical, and we expect to decrease again the CPU footprint. |

**Table 3.2:** Hadoop Networking

## 3.3 In-Memory RDMA-Enabled Architecture

The architecture of HDFS as it was described in the background chapter consists of one master node (namenode), a number of slave nodes (datanodes) and clients who use the filesystem. Altering HDFS to an in-memory system and applying the RDMA communication model requires modification in all parts of the architecture.

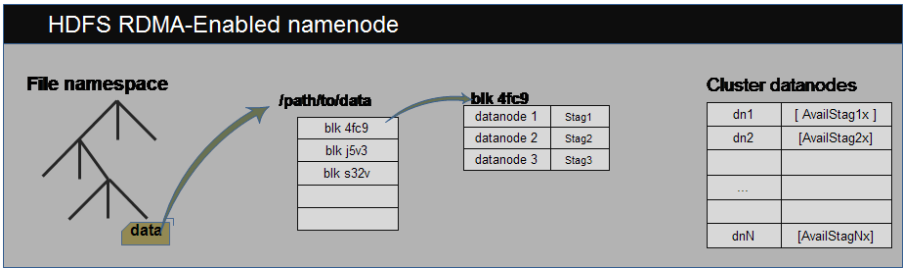



**Figure 3.2:** RDMA-Enabled Namenode

To start with, the namenode should hold the extra information needed by the RDMA communication model. In comparison to the normal architecture, namenode maintains a list of available stags[1] for every operational datanode in the cluster. These stags will be used by a client in order to write a new block. Furthermore, for every stored block in the cluster, namenode holds the information in which datanodes the block is physically stored, along with the exact memory location. This information will be used by a client during a read operation. The summary of the namenode is given in **Figure 3.2**.

On the datanode side, extensive modification is also required. First, a new component, which will manage the memory and will serve the RDMA operations, should be added. The performance of this unit is crucial in terms of performance and scalability. Second, data is essentially moved from disk to memory. Specifically, data should be found in memory during a read or a write operation. Outside HDFS operations, data is stored also on a disk which is necessary for scalability (big data) and administrative (reboot) issues. In essence, slow storage devices are removed from the critical path and maintained outside of it. Furthermore, the node is designed to allocate memory space for future write operations (available stags) and to store the mapping between memory locations (block stags) and the existing stored blocks. In **Figure 3.3** the structure of an RDMA-enabled datanode is shown.

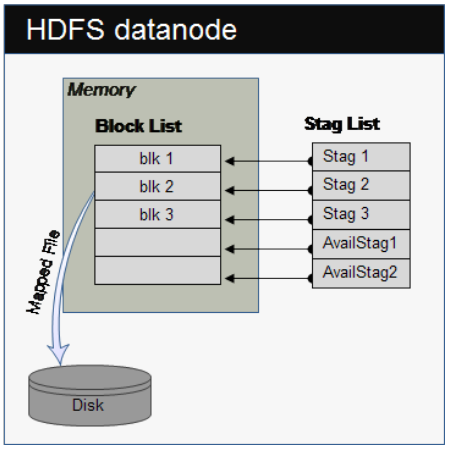



**Figure 3.3:** RDMA-Enabled Datanode

[1]Steering Tag (STag): is an identifier of a nodes memory buffer registered for RDMA operations. An Stag is used to reference a registered buffer for local and remote read and write operations

Finally, the communication model among namenode, datanodes and clients requires also to be extended. In the RDMA-enabled architecture (**Figure 3.4**), datanodes inform the namenode first about their available stags, and second in which memory locations are located the blocks that they already store. The communication between namenode and clients should be extended in order to include the necessary stag information during read/write operation. This would be the out-of-band buffer advertisement mechanism that RDMA one-sided operations require, and the HDFS RPC framework for message exchange between nodes is a good fit. Eventually, the data transfers protocols need to be modified to be RDMA based; this requires modification in the transport level in both HDFS client library and in the datanodes.

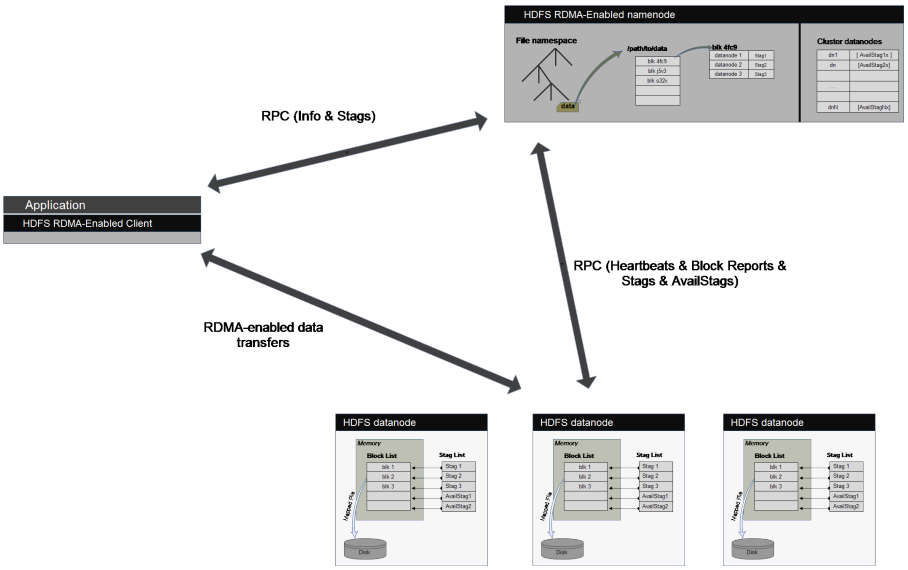



**Figure 3.4:** RDMA-Enabled Architecture

### Write - Read Operations

The read, write operation works in a straightforward way under RDMA. For reasons that will be explained later in this chapter, these two operations are based on one-sided operations. When writing a block into HDFS, the client will ask the namenode to which datanodes the new block should be saved. The namenode will respond with the target nodes, but it will include also a memory location (stag) for each node in the response. This memory location is available for writing, and it has been advertised by the datanode to the namenode in a non-critical moment. The client can directly perform an RDMA write operation on the memory location indicated by the namenode. Data is replicated forward in the pipeline by further RDMA write operations. In a similar way, when a client wants to read, it will receive from the namenode the datanode, along with the memory location, in which the block is saved. A direct RDMA read operation will fetch the data locally. A detailed technical description is given in the next chapter.

## 3.4 Hidden Costs of RDMA

According to [21] the performance of the RDMA networking model is greatly defined by how efficiently two inner tasks are used. These two tasks are the connection establishment and the memory registration.

### 3.4.1 Connection Establishment Cost

Before an RDMA operation can be performed, a connection setup between the two RNICs should take place. This connection setup, contrary to the TCP/IP handshaking, is a time-consuming process, an estimation of the overhead delay is around 200 ms as stated in [21]. Therefore, in our implementation, special focus was given to maintain the connections opened. In particular, every connection, when established successfully, is cached, and can be reused. Of course, maintaining a such cache of established connections requires that both ends remain active, which is not always the case in HDFS application logic. For example, connections between datanodes[2] can be cached since datanodes remain alive for ever. Whereas caching connections from/to HDFS clients has a temporary significance since client instances are terminated when finish. In our implementation, connections are established only when it is necessary, and they are maintained as long as possible.

A challenging issue is how the following problem can be solved: a client instance A creates a connection to a datanode. When A finishes its work, it dies. Another client instance is created in the same physical machine and needs to connect to the same datanode, can the new client make use of the connection that the first client established? A potential solution would be an RDMA-connection cache to be maintained by the RNIC, and the application should be able to use it.

### 3.4.2 Buffer Registration Cost

Registering memory to the RNIC is a time-consuming, as well as, CPU demanding task. For that reasons, there should be given extra attention not to have memory registration during the critical path. Furthermore, for better CPU utilization, application should reuse the already registered buffers. In our design, there are mainly two cases where memory registration takes place.

First, the datanode registers the buffers in which clients will write blocks in the future. The registration is not in the critical path and usually takes place during the initialization of the node. In more detail, there are two strategies, one: the node to register all the buffers at the beginning or two: node to register buffers incrementally when a write takes place. Both cases are from an RDMA perspective the same, since the registration is outside the critical path and the reuse of these buffers is not possible[3]. Which of the two strategies[4] should be followed is a memory management topic (how much memory is available for HDFS, how much memory other processes in the node require, etc.)

Second, memory registration takes place in a client when it wants to write a block. The problem here is that input file is split into slices of variable size (known at runtime), each slice is saved in a buffer, and these buffers are given to HDFS client. Furthermore, after the application has written some slices, the destination node becomes known to the client. In other words, HDFS client does not know neither the destination, neither which buffer will have the data to pre register it. Thus, the memory registration of the communication buffers is, in application logic, located into the critical path, i.e., client creates a buffer, it puts data to the buffer, it gets the destination, then it can register the buffers and sends them. There can be many solutions to this problem. A good solution to that problem is: the client will ask buffers from a pool, it will register the buffer, and it will return them back to the pool after successfully place the content to the remote memory. Hence, the client, after a number of slices, will start reusing the buffers and there will be no need for further registration.

[2] maintaining the connection between datanodes in the same rank can be very beneficial because replication mechanism puts the first two replicas in the same rank

[3] the implementation does not support the deletion operation, in that operation the buffer will be reused

[4] in the future SoftiWARP + memory lazy pining will make this decision irrelevant since the allocated HDFS buffers will not remain the memory but they will be swapped to the disk releasing memory

## 3.5 One-Sided Vs Two-Sided Operations

Two-sided[5] operations do not express the RDMA semantics well enough. Apart from this common belief, we decided to base our implementation on one-sided operations for the following reason: the communication pattern and the behavior of the client is completely unknown. Hence, the datanode cannot post in advance the appropriate post-receive calls, which will serve the clients request. In more detail, for every connected RDMA client, datanode is not aware whether the client is a reader or a writer of a block (it might be both), it is not aware when the client will perform an operation (maybe never), and finally it has no information about the length of the data. Due to that, two-sided operation will create a number of issues:

- Memory scalability issue: Datanode cannot reserve resources for long-lived clients. Lets say we have 10 clients, connected to the datanode and block size equals to 64MB. In that case, the datanode will have constantly reserved 640MB of memory in order to serve a potential send post from the clients. Register a HDFS-block-sized buffer per active client greatly reduce how many clients can simultaneously be connected.

- Memory efficiency utilization: Memory might not be efficiently used because resources will be reserved waiting for clients to make use of them. For example, the datanode has 256MB memory and 4 HDFS-blocksized buffers (b1,b2,b3,b4) of 64MB are registered. The datanode has 3 active clients (c1, c2,c3) which means that b1 buffer is assigned to post-receive for c1 client, b2 for c2, b3 for c3. Client c1 writes once buffer b1, server assigns buffer b4, c1 writes again buffer b4, c1 wants to write one more time. In that situation, no memory exists for a further write by c1. Clients c2, c3 might never write again.

- RDMA failures: The length of the data and how many times the client will post a send request is defined on the fly. For example, a client wants to write a block of 32MB. In order to replicate the block through a pipeline the data should be transferred in chunks of 8MB. In that situation, the client should post 4 send requests and the server should post on time 4 receive requests assigned to the same buffer[6]. Nevertheless, in Hadoop, both block size and chunk size is defined by the client on the runtime. Thus, another client might write a 64MB block with 4MB chunk size or with 64MB size if no replication is needed; the server cannot know how many receives should post.

The above remarks are indicative situations in which two-sided operations might not be a good fit, and in large distributed systems, more issues might appear. In all cases, one-side operations seem to be more flexible and better suited for this communication pattern when transferring data. Of course, it is a matter of decision and implementing Hadoop write operation using two-sided operation is possible as long as all the problems would properly be identified and handled. To our belief, such implementation would have added a great amount of complexity, and it would have not taken advantage of RDMA semantics as much as one-sided operations.

On the other side, two-sided operations are a good fit for control messages. Control messages can be stored in a buffer with known and small size. Therefore, posting a number of receives per client will not prevent the system from scaling, neither will consume important resources for clients who might be inactive for long time. Furthermore, transferring control messages using two-sided operation does not require a careful synchronization between post sends and post receives. The server should just manage to always have a receive, connected to a control buffer, posted.

[5] post send on the client side, post receive on the datanode server side

[6] the post-send/receives cannot be posted synchronously in one call because of the pipeline

## 3.6 Scaling Beyond Physical Memory

Up to this point, the proposed design indicated a distributed data processing system which operates completely in memory. Maintaining all data in the physical memory is clearly not a scalable solution, in particular, when the system targets at big-data scale. Our system is designed to scale up to large data volumes. This target can be accomplished by adapting a lazy memory pinning technique as proposed in [22].

In a hardware-based RDMA implementation, registering memory buffers to the RNIC would require that this memory location gets pinned by the OS and stays resident to the physical memory. In a such RDMA system, memory resources will shortly be run out, because the memory, which is registered for RDMA operations, cannot be swapped out to disk. In the software-based RDMA implementation, we can overcome this limitation. Any memory registered to a SoftiWARP RNIC will not be pinned until it is accessed for an RDMA operation. Outside of RDMA operations, memory can be paged out, and physical memory resources can be used by other processes or for storing further data. In more detail, when sufficient memory is available, all data would be in memory, and the system would operate natively as an in-memory. When nodes are short of physical memory, data would be swapped out on disk. If an RDMA operation needs to be performed on that memory, it would be swapped in again. The overhead, as well as the performance, of a such system should be compared to a system, which is working with files and makes use of the operating system cache.

Lazy memory pinning would be provided by the RNIC, and application implementation does not need to be modified. The task of transferring data between disk and memory is assigned completely to the operating system. Our benefit would be that the total storage capacity of the system would be equal to the total virtual memory of the cluster, which exceeds by far the physical memory.

## 3.7 Replication Mechanism

HDFS is a high fault-tolerant distributed filesystem, a feature which is provided through a replication mechanism. In an in-memory system this mechanism should be fundamentally questioned and redesigned for a number of reasons. First, replicating blocks and maintaining them in a node's memory greatly limits the storage capabilities of the cluster. Therefore, new approaches can be followed: for example, one replica of the block should be always in a node's memory, while the remaining copies should be stored on the disk; namenode on demand will define which block will be in memory and should forward this information to the client application[7]. Second, error recovery efficiency depends on the fact that the network topology is known and that replicas are strategically saved on a combination of nodes which minimize the possibility of a data loss. Nevertheless, this information might not be always known, e.g., in a virtual environment, and thus the fault-tolerance performance is significantly decreased. Third, this replication mechanism, which is based in a pipeline, was suggested because it minimizes the network traffic. This approach fits in a data center with the classical hierarchical network topology. However, "flat fabric" (e.g., Portland[23]), as well as multi-gigabit network architectures are being planned at a growing number and for that reasons, a different approach, like parallel writing of the block, could be a better idea. Nevertheless, this master thesis did not further investigate this topic, and the original mechanism was maintained.

[7] e.g., if possible a map task should be executed in the node which holds the data in memory and not in a node which have the data on the disk

# Chapter 4

# Implementation

## 4.1 Project Info

Hadoop is an Apache project and all the components are available through the Apache open source license. It is written in Java, mainly for cross-platform portability. In this master thesis, we picked to modify Hadoop version 0.23 as it was released in February 2012. The decision in favor of an unstable alpha version versus the latest stable release, was taken because a future RDMA integration might be possible. However, working with such a new version made things more difficult because of its complexity (complete new architectures were introduced), the lack of documentation (how to set up a cluster) and its instability (execution times varied significantly during benchmarking). The Java version under which the system was compiled and tested was 1.7. Integrating RDMA communication model in a Java project was feasible thanks to jVerbs [24].

## 4.2 Anatomy of an RDMA Node

Every RDMA-enabled node in a distributed system should be robust and flexible. It is required to act efficiently either as server or as client and always in a completely asynchronous manner. In our design, there are two main classes, which implement an RDMA node: RdmaServer and RdmaClient. An overview of the design is given in **Figure 4.1**.

RdmaServer is created first and handles all the critical actions, while the RdmaClient(s) is created afterwards and originates from a parent RdmaServer (in order to make use of its registered memory). RdmaClient is, in essence, an active connection from the local RdmaServer to a remote RdmaServer. An important note here is that the RdmaClient can be created either when the local RdmaServer passively receives an incoming connection, either when the RdmaServer actively creates a connection to a remote node due to the application logic. For example, a datanode should create an inbound client when a HDFS application is writing a block to it, and an outbound client when the node itself connects to an another datanode when forming the pipeline. In more detail, RdmaServer is responsible for the following tasks:

1. It handles all the memory registration. This task should be performed under a unique protection domain in order all RdmaClients to be able to perform operations on the same memory locations (e.g., a datanode should forward a local memory buffer, which was written by a client, to another datanode).

2. It listens in a loop for new incoming connections and creates on demand new outgoing connections. For every connection, a predefined window of post-receive calls is posted[1]. In a distributed

[1]reminder: two-sided operations carry only control messages, and therefore, the server knows what buffer size to post

system, the implementation of this part is critical, and it should enable a very large number of clients to establish a connection concurrently[2].

3. It maintains a unique cache of all active clients, inbound or outbound. Every client should be identified by the pair IP address and the port number, because this information is only given to the transport layer by the application[3].

4. All RDMA operations (post-Send/Receive, RDMA-write/receive) are executed always through this server.

Managing the memory efficiently is crucial, for that reason the server maintains a subclass, called PoolBuffer. PoolBuffer is responsible for a dynamic distribution of the direct buffers in order to reuse buffers as much as possible and to totally control the memory usage[4]. The following example explains how the pool buffer is working:

1. Client C1 requires to transfer data of size 512 Bytes to a remote node.
2. C1 asks from the PoolBuffer a buffer of 512 Bytes.
3. PoolBuffer creates (allocateDirect) a buffer of size 512 Bytes and returns it to C1.
4. C1 performs an RDMA operation, during which the buffer is registered to the RNIC[5].
5. When the buffer serves its purpose (e.g., when consumed by a post receive call), it is returned to the pool.
6. Later on, another client C2 requires a buffer of size 512 Bytes, PoolBuffer will return an already register buffer[6].
7. C2 will perform an RDMA operation without memory registration[7].

### Hiding Complexity of Control Messages

As we have discussed, data transfers are made through one-sided RDMA operations, which simplified significantly the implementation of the data path. Nevertheless, in real world applications, like HDFS, a large number of control messages should be exchanged. Moving the responsibility of handling all the communication buffers for the control messages to the upper layers of the application adds great complexity. For that reason, a good idea would have been to maintain the existing socket-based communication just for the control path. Nevertheless, it was decided to implement all the functionality of HDFS using the RDMA communication model. For that purpose, a robust mechanism, which hides the explicit buffer management from the upper layer, was created.

The RdmaClient creates two queues, named postQueue and commandQueue. The postQueue holds in the right order the buffers that are expected to be consumed by the post-receive calls. The commadQueue holds the content of the communication buffers after they have been consumed. At the same time, RdmaServer creates a unique daemon, called Poller. The Poller loops over a global completion queue (CQR), which is only connected to the completed post receive calls. Upon a completion event, Poller identifies in which client (through the QP number) the post-receive was consumed, and transparently transfers the content of the communication buffer to other part of the program, which will make use of it. A description of the mechanism is the following:

[2]this part of the server was implemented in Java according to an example in C as found in `http://thegeekinthecorner.wordpress.com/2010/09/28/rdma-read-and-write-with-ib-verbs/`

[3]acquiring the IP address of an inbound client is a future work in jVerbs

[4]in Java direct buffers are outside the heap and GC is not working efficiently

[5]i.e., client asks the RdmaServer to register the buffer and a hash map entry between the buffer and an IbvMr is added to a hash table, called rBufMrMap

[6]Neither PoolBuffer or the client is aware that the buffer is registered, only RdmaServer knows through the hash table rBufMrMap

[7]The client will query the hash table rBufMap maintained by its parent RdmaServer in order get this knowledge

1. An external node makes an RDMA connection to the server.
2. A client instance is created, three buffers from the pool are inserted on the postQueue, and three post-receive calls are placed.
3. Asynchronously, the post-receive call is consumed and the Poller gets a completion event. Poller identifies the right client, extracts the content of the buffer and places it to the commandQueue. Finally, Poller returns the buffer to the pool and orders a new post-receive for that client.
4. Parts of that program that are interested in the content, should loop over the commandQueue and will be informed instantly.

To our belief, the above mechanism provides a transparent use of two-sided operations given that only RdmaServer is managing communication buffers. At the same time, the threads which make use of the RDMA connection, receive the control messages through a fixed interface (commandQueue). Furthermore, the mechanism offers efficiency because all buffers are returned to the pool and can be reused. Additionally, it is robust because many connections (RdmaClients) can be supported. Every time a completion event takes place, another post-receive call should be placed. In that way, the other side can continue post send operations to the server. If reposting a receive is not possible, an error mechanism should be implemented.

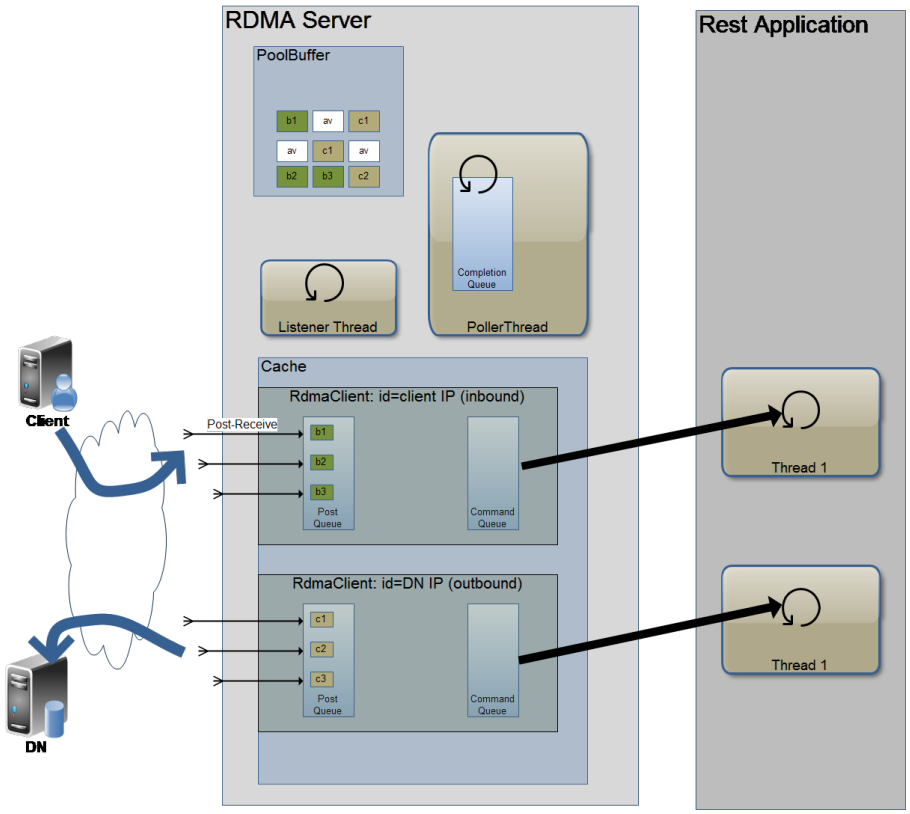



**Figure 4.1:** RDMA Node

## 4.3 Removing Hard Disk from the Critical Path

According to the design, slow storage devices should fundamentally be removed from the critical path, but maintained out of it. Specifically, during a read or a write operation, the data should be read from or written in a memory location without touching disk. If data is not in memory, the OS will fetch it from the disk using the virtual memory mechanism. The above design requirement was implemented by storing the data using memory mapped files. Memory mapped files can be modified as a memory buffer, but the content is written back to disk in a synchronous or asynchronous way. In more detail, we succeeded the following: during a write operation, a block is written quickly to a buffer in memory, while asynchronously the content of the buffer is written to the hard disk. Contrary to our design, normal Hadoop is waiting data to be written on the disk. An important note here is that finally, OS caching defines whether disk is used or not in critical path.

## 4.4 Refactoring HDFS Write Operation

In this subsection, without going into the details of the source code, we try to give an insight of how things were modified and what the expectations are. **Figure 4.2** describes how a client executes a write operation, e.g., when a reducer writes an output file.

The data are received by the HDFS client layer as a sequence of chunks (every chunk is a byte[] buffer of 512 Bytes). The total number of chunks depends on the file size. For every chunk, a header is added and they are copied into a packet object, which is a container of chunks. The size of the packet is usually 64 KB. When a packet is full, it is pushed to a queue. Another thread (DataStreamer) waits on that queue, and sends the packets into the socket when it is ready. Ready means that the thread has asked the namenode, it has received the target datanodes, and it has established the pipeline. The data is copied to a temporary direct buffer owned by JDK code, and then is copied into a kernel buffer before finally send over the network. On the receiving side there is a slight lighter procedure, but it remains quite the same. It is obvious that in overall, a number of unnecessary copies takes place, which makes the communication at multi-gigabit networks extremely not efficient in terms of performance and CPU utilization.

In our implementation, things changed according to the RDMA semantics. HDFS client receives the data and stores them directly to a userspace buffer. No other data transfer takes place internally. This is the data path (green area in **Figure 4.3**). The rest of the implementation was maintained, but now it just carries control messages. For example the data queue contains a packet which basically holds a memory reference. This constitutes the control path (blue area in **Figure 4.3**). When the DataStreamer thread fetches the packet, it just RDMA writes from the local address, which was indicated by the packet, to a remote address, which was indicated by the namenode. Due to the above modification, we expect that the write operation would be faster and would require less CPU cycles.

## 4.5 Anatomy of a Write

In this section, we provide a sequential description of the write operation. The sequence diagram is given in **Figure 4.4**. Firstly, the client executes the create method which asks the namenode to create an empty file. The namenode, after checking a number of parameters (permissions, free space, etc.), creates successfully the file and informs the client to proceed.

The input file is split in small slices (8 MB) and every slice is saved into a temporary buffer. The address of the buffer is added into a queue, called data queue. When there is a buffer to be sent in the queue, a thread, called DataStreamer, contacts the namenode using the method addBlock and asks to write a new block. The namenode will respond with a pipeline of datanodes along with the block id. The namenode will respond only after checking that the client is the lease holder for the file and any prior operation has been completed (e.g., previous write).

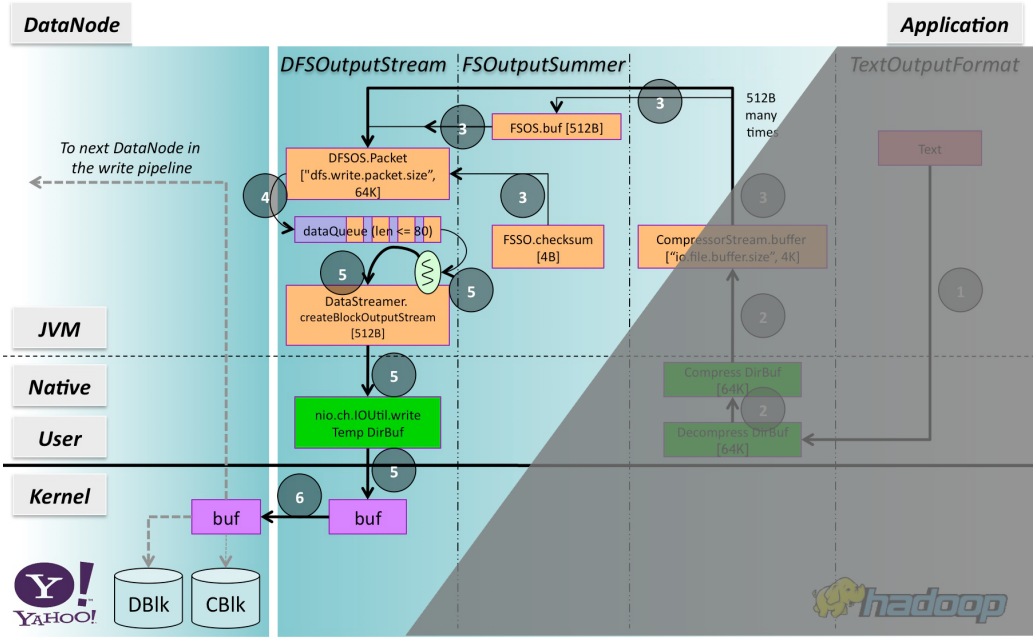



**Figure 4.2:** Normal HDFS Write, Image from developer.yahoo.com/blogs/hadoop/posts/2009

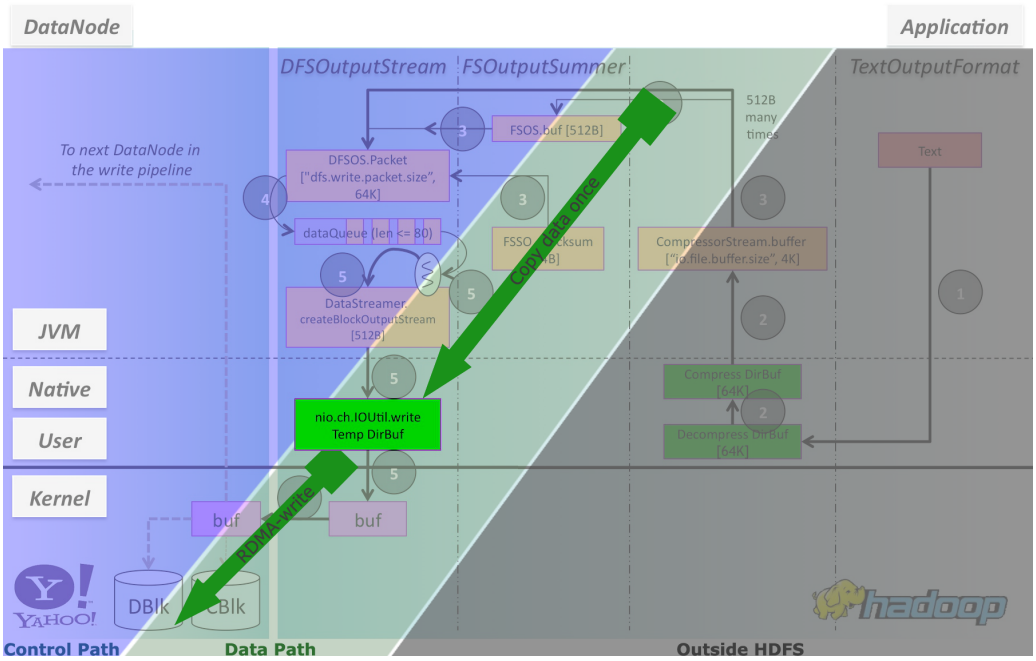



**Figure 4.3:** RDMA-Enabled HDFS Write

When the client will have the target nodes, it will remove the buffer from the queue and it will send it through an RDMA write operation to its destination. In more details, the client will post a non-signaled RDMA write from the local address (address of the buffer) to a remote address as indicated in the pipeline. In the same RDMA call, it will post a zero signaled RDMA read operation. When this call (write and read together) is completed successfully (the client will wait on the CQ), it will post a send, including a control message which will contain the directions to the first datanode

how to forward the data over the pipeline. In that way, we have a true zero transfer (non-signal write), we are sure that the data transfer has finished (signaled read) and then we can safely ask from the datanode to forward further the data. Grouping everything in one call and eliminating the zero read for performance issues is a future optimization. The buffer is forwarded through the pipeline. In our implementation, datanodes depend completely on the information of the control messages in order to forward the buffer. The last node in the pipeline will acknowledge directly to the client that the buffer has been forwarded to its memory. Thus, there can be a flow control in the system as exist in the normal Hadoop version. At some point, the buffer, which was extracted from the data queue, would be marked as the last buffer in the block. Then the client will also post a control message indicating the end of block. All datanodes will receive this message, they will store the block (this stag will be connected to that block id) and they will inform the namenode about the reception. If there are more buffers in the queue, the same procedure will be repeated.

When the client is done writing, it will call the complete method to the namenode. This function returns a boolean to indicate whether the file has been closed successfully or not. A call to complete method will not return true until all blocks have been replicated the minimum number of times. Client might call complete method several times before succeeding.

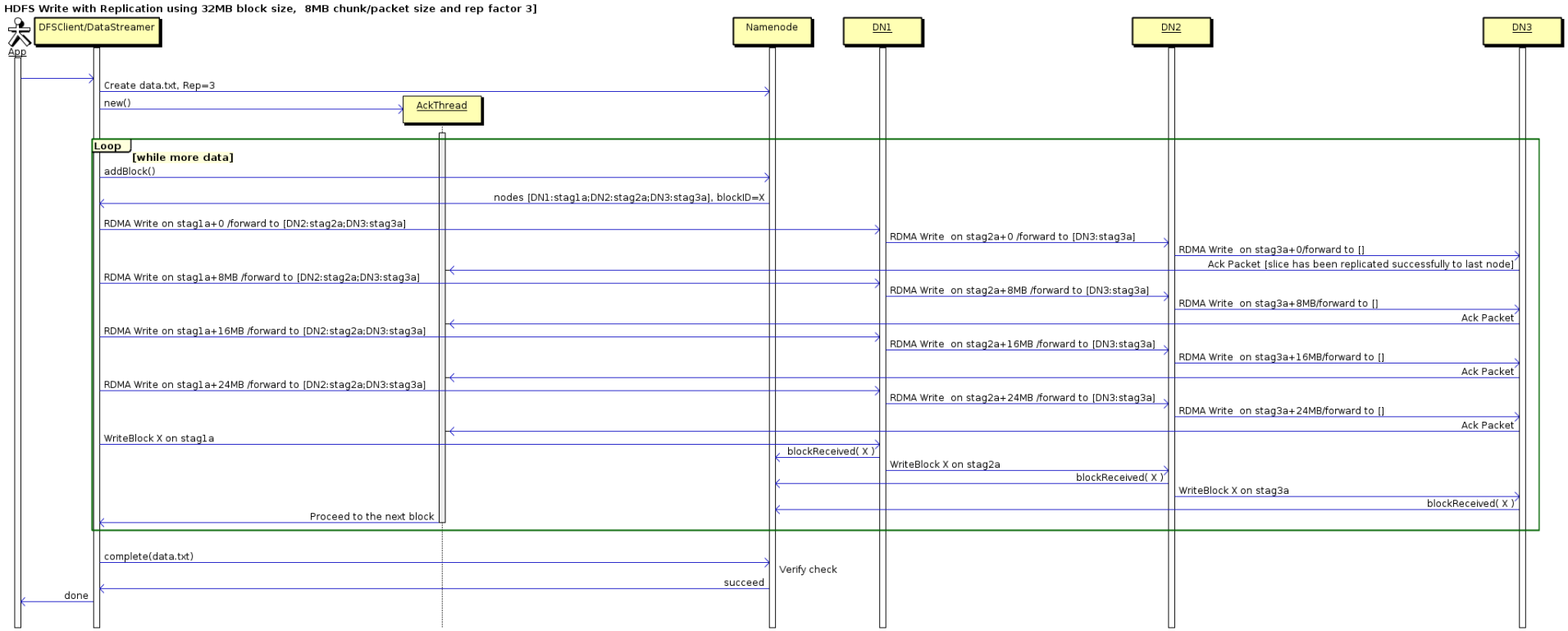



**Figure 4.4:** Sequence Diagram for RDMA-Enabled HDFS Write with replication

## 4.6 Anatomy of a Read

The RDMA-enabled read operation has not significantly changed from the original version because the normal API had to be maintained. In **Figure 4.5** the sequence diagram of a read operation is shown. The overall procedure remains the same as described in the background chapter. The difference here is that when the InputStream will ask for the location of the blocks, the namenode will return the extra information where the blocks are saved in memory (stags). Therefore, when the stream will contact directly the datanode to fetch the data, it will do it through an RDMA read operation. The buffer which finally stores the data, will be wrapped as stream and will be given to the client for reading. In our implementation when the stream requires data from a datanode it fetches always the complete block[8]. This might be a disadvantage if the client does not want to read fully the file, nevertheless it provides the best performance in the normal case. In the future, the HDFS API of the reader should be modified in order to take advantage of the RDMA communication model. The API should be formed around the topic "ByteBuffer-based read API for DFSInputStream" (jira HDFS-2834). This improvement, which was introduced in a later version than the one we modified, adds great flexibility and fits greatly to the RDMA communication model. In a nutshell, the client will supply its own buffer and the data will be copied directly into that buffer. In this way, client defines the size and the offset of the data, which is not the case when using a stream.

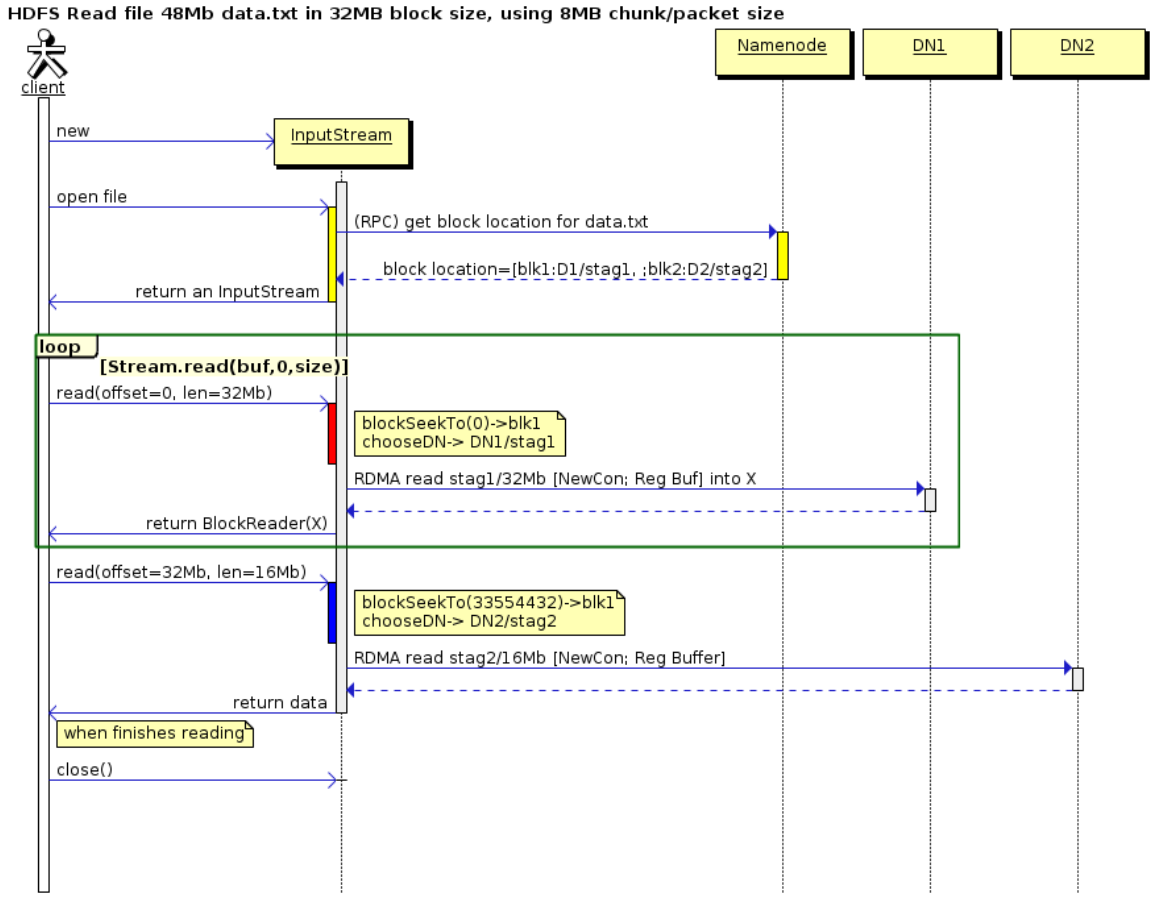



**Figure 4.5:** Sequence Diagram for RDMA-Enabled HDFS Read

[8] it fetches always from an offset to the end of the block

# Chapter 5

# Evaluation

This chapter evaluates the implementation of our in-memory RDMA-enabled HDFS. In the first part of chapter, we verify the correctness of our work through a number of validation checks. In the second part, we proceed to the assessment of the performance, and compare our modified version to the original version.

## 5.1 Validation

The validation of the implementation was checked in the following ways:

- A bunch of read/write operations: as described in **Listing 5.1**, we were testing constantly the system by reading and writing files of variable size. The validation check was obvious, to receive the exact same data without error.
- Replication validation test: as described in **Listing 5.2**, the replication mechanism was tested by writing files into HDFS with replication factor equal to the number of datanodes in the cluster. Then, the validation check was to verify that the data folder in each datanode holds the exact same data.
- Execution of MapReduce tasks: making a real MapReduce task to work would indicate whether the HDFS API was supported correctly or not. The implementation worked for the MapReduce tasks listed in **Listing 5.3**. The output of our implementation was checked for its correctness towards the output of the normal Hadoop.

```
FILES="file32MB file64MB ... file1024MB"

#Uploading Files
for f in FILES
    hadoop fs -put f f

#Downloading Files
for f in FILES
    hadoop fs -get f f_cp

#Validation Check
for f in FILES
    md5sum f f_cp
```

**Listing 5.1:** Validation 1: Write/Read

```
FILES="file32MB file64MB ... file1024MB"

#Uploading Files
for f in FILES
  hadoop fs -Ddfs.replication=4 -put f f

#Get the data from DN
for node in DNs
  x[i++]=`ssh node md5sum ~/hddata/blk*`;


#Validation Check
  compare([x])
```

**Listing 5.2:** Validation 2: Replication

```
#WordCount
hadoop jar ~/hadoop/hdhome/share/hadoop/mapreduce/hadoop-mapreduce-examples-0.23.1.jar
    wordcount file output

#TestDFSIO (before ack mechanism)
hadoop jar ~/hadoop/hdhome/share/hadoop/mapreduce/hadoop-mapreduce-client-jobclient-0.23.1.jar
     TestDFSIO -write -nrFiles 2 -fileSize 100MB
hadoop jar ~/hadoop/hdhome/share/hadoop/mapreduce/hadoop-mapreduce-client-jobclient-0.23.1.jar
     TestDFSIO -read -nrFiles 2 -fileSize 100MB

#A Streaming job
hadoop jar ~/hadoop/hdhome/share/hadoop/tools/lib/hadoop-streaming-0.23.1.jar -input file -
    output f -mapper 'cat' -reducer 'wc -l'

#PI Example
hadoop jar ~/hadoop/hdhome/share/hadoop/mapreduce/hadoop-mapreduce-examples-0.23.1.jar pi 2 10

#Random Writer
hadoop jar ~/hadoop/hdhome/share/hadoop/mapreduce/hadoop-mapreduce-examples-0.23.1.jar
    randomwriter -Dmapreduce.randomwriter.mapsperhost=1 -Dmapreduce.randomwriter.bytespermap
    =10000 output

#Sort
hadoop jar ~/hadoop/hdhome/share/hadoop/mapreduce/hadoop-mapreduce-examples-0.23.1.jar sort -
    r 1 output output2
```

**Listing 5.3:** Validation 3: MapReduce Tasks

## 5.2 Performance Benchmarks

In the second part of the evaluation chapter, we executed a number of benchmark tests in order to assess the performance of our implementation towards the original version.

### 5.2.1 Setup

The testbed was a small non-uniform cluster of five IBM servers. Each server was running Linux (Debian stable 64 bit, kernel version 2.6.36.2), was equiped 4x Intel(R) Xeon(R) CPU E55XX @ 2.XXHz, the memory was between 6 GB to 8 GB. All servers were connected to a 10 GbE switch. The Hadoop version was 0.23.1(RDMA), the java version was 1.7.0_03 , and the HDFS block size was 32 MB. We used exclusively SoftIWARP implementation, which means that the RDMA functionality was provided in software level. With the exception of a machine, which was always dedicated as a namenode, the role of the rest machines were changing according to the benchmark needs. The cluster structure is given in **Figure 5.1**.

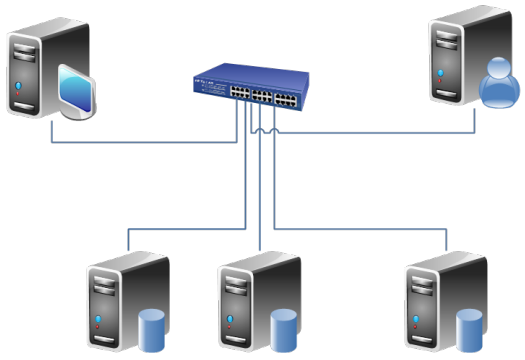


**Figure 5.1:** Testbed Setup

### 5.2.2 Monitoring Methodology

We performed a series of benchmarks in which we compared our implementation (RDMA-Hadoop) versus the original version 0.23.1 which was modified (Normal-Hadoop). In every benchmark, we monitored the behavior of a node using a robust, very informative monitor script. In more detail, we created a generic python monitoring script, which measures every important info of the node. It measures among others CPU usage, memory usage, I/O activity, network bandwidth, context switch. A sketch of how the script works and which UNIX monitor tools were used is given in **Listing 5.4**. The script receives the output of the monitoring commands every second and saves the output into CSV files for analysis. Therefore, in every experiment we had a complete and accurate system profile of the nodes.

```
#PID MONITORING
PIDSTAT = "/usr/bin/pidstat -p %(pid)s -u -d -r -h -l 1 3600"
TOP="/usr/bin/top -p "%(pid)s -b -d  1 -c

#GLOBAL MONITORING
CPU="sar -u ALL 1 3600"
PROCLOAD="/home/X/nfs/benchmark/bin/proc-load2.sh"
MEM="sar -r  1 3600" # or with MEM="free -s 1 -m -t|grep Mem"
SWAP="sar -S 1 3600"
IO="sar -b 1 3600"
IOSTAT="iostat -d 1 -m"
VMSTAT="vmstat -S M 1"
CNTX="sar -w 1 3600"
NET="/home/X/nfs/benchmark/bin/getBW.sh"
OPROFILESTART="/home/X/nfs/benchmark/bin/op-cpu-start"
OPROFILESTOP="/home/X/nfs/benchmark/bin/op-cpu-stop %(dir)s"

...
#Fix the commands
self.cmd_cpu = shlex.split(CPU)
...
#Starting the commands
self.tstart=datetime.datetime.now()
self.ps_cpu = Popen(self.cmd_cpu, stdout=PIPE)
...
#Gathering the outputs & stopping
out_cpu = [outline for outline in self.ps_cpu.communicate()[0].split('\n')][1:]
self.tend=datetime.datetime.now()
...
#PostProcessing outputs & write to files
[dataset, cmdline, headerline]=process_output(out_cpu,"CPU")
info="#Monitoring ["+ cmdline+"]"
info+="\n#Cmd=["+ CPU+"]"
info+="\n#Hostname=["+ HOST+"]"
info+="\n#Time= ["+self.starttime +"] to ["+self.endtime+"]"
print info+"\n"
write_log_file("withSarcpu",headerline,dataset,info)
...
```

**Listing 5.4:** Monitoring Script

Among the monitoring tools, Oprofile [25] is the most important. Oprofile is a low overhead open-source system-wide profiler for linux and it can be used to find CPU usage in the whole system and within processes. In **Figure 5.2**, we present an example CPU-usage graph of a node, which was ordered to perform a MapReduce task (WordCount). As we observe, during this task, the node used fully all the four CPUs, and according to the Oprofile, 580006 CPU cycles were spent in total.

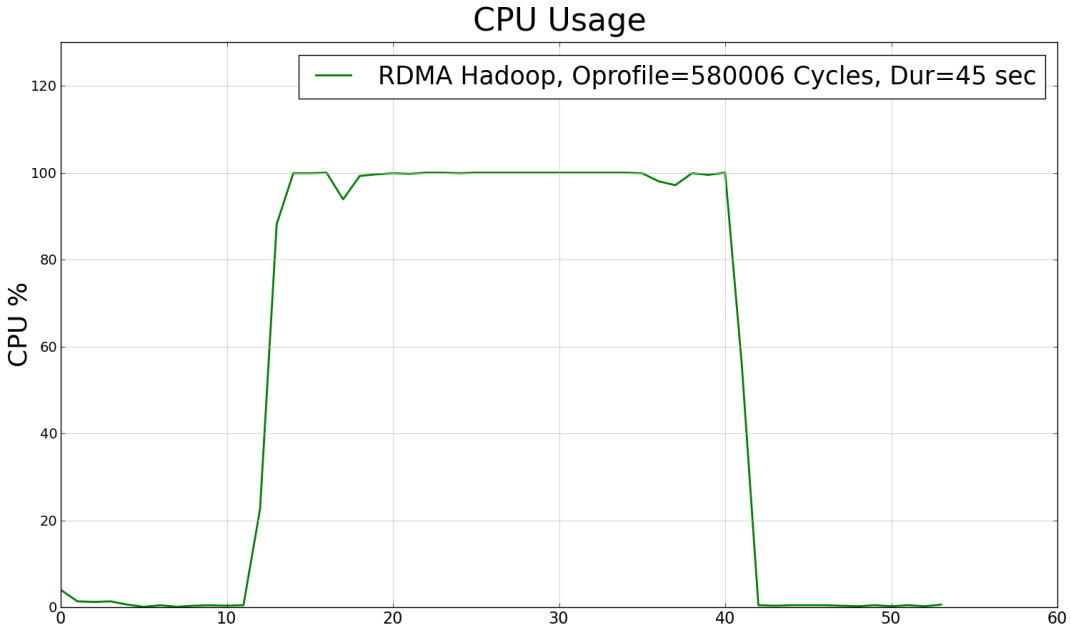



**Figure 5.2:** Example CPU Usage Graph during a WordCount Task

### 5.2.3 Uploading a Big File in the Cluster

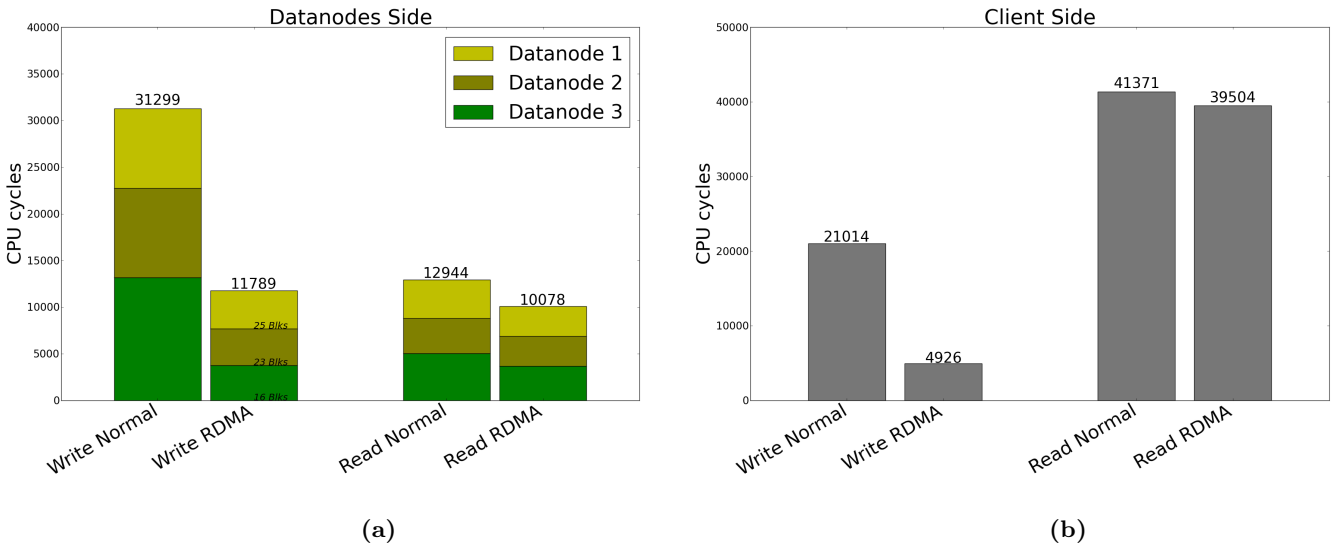
We performed the following benchmark: one external client uploads a big file of size 2048 MB into a cluster consisting of three datanodes and then it reads the same file back (**Listing 5.5**). The test was performed twice, one versus our RDMA-enabled implementation and one versus the normal version. The HDFS blocksize was 32 MB and the replication factor was one.

```
#1. Start Distributed Monitoring
#2. Upload Big File
/usr/bin/time -f "%e" hadoop fs -Ddfs.replication=1 -put file2048MB file2048MB
#3. Stop Distributed Monitoring
...
#1. Start Distributed Monitoring
#2. Download Big File
/usr/bin/time -f "%e" hadoop fs -get file2048MB file2048MB_cp
#3. Stop Distributed Monitoring
...
#Validate
md5sum file2048MB file2048MB_cp
```

**Listing 5.5:** Benchmark 1: Upload/Download Big File to Cluster

The results are given in **Figure 5.3**. Both graphs were created by the output of the Oprofile tool which indicates how many CPU cycles the system did during the test. The benchmark was repeated several times and each time the numbers were similar. Further, the results were compared to the output of the unix command "top" which also measures CPU usage and they were in accordance.

We observed that during write operation there has been a great reduction in CPU usage in both sides. On the datanodes side, in order to save a 64 block-sized file in the cluster, the system spent around 31299 CPU cycles in the normal implementation, while it spent around 11789 CPU cycles in ours (62%). On the client side, the CPU usage was 21014 cycles in normal version versus 4926 cycles in our implementation (70%).




**(a)** **(b)**

**Figure 5.3:** CPU Usage, Write/Read 2048 MB File in a 3DN Cluster (=64blocks of 32MB)

As far as the read operation is concerned, a slight optimization was also observed. On the datanodes side, the CPU cycles were reduced from 12944 in the normal implementation to 10078 in the RDMA-enabled implementation (22%). On the client side the change was insignificant, 41371 vs 39504 (4%).

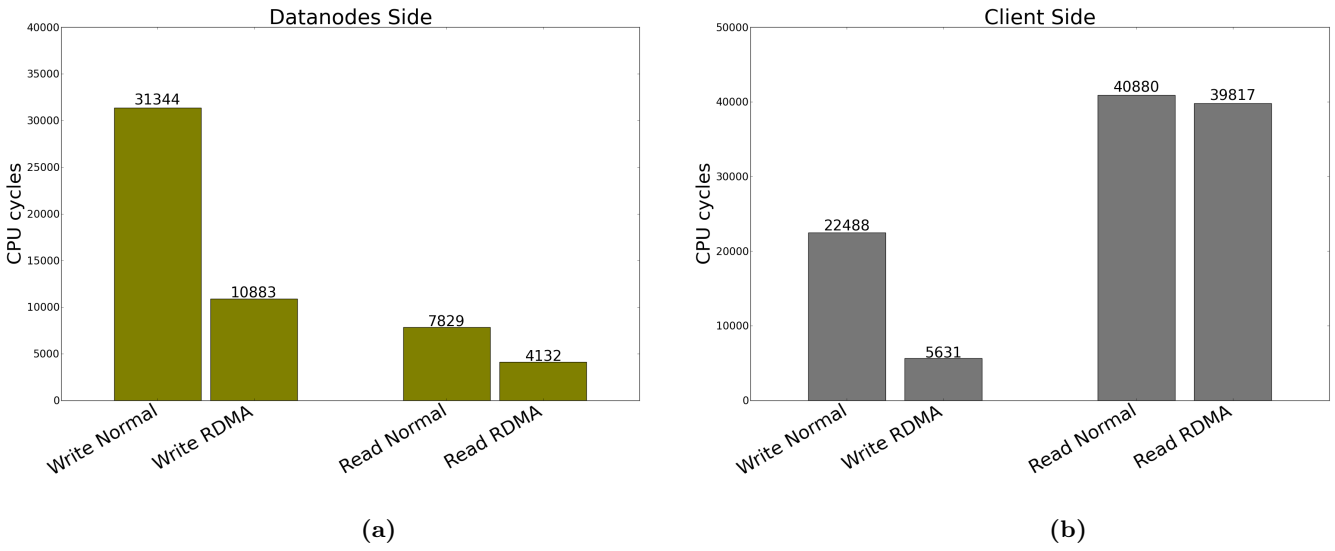
Comparing the systems in a distributed way is difficult for three reasons. Firstly, the cluster datanodes have not the exact same hardware, thus summing up the results is not completely correct. Secondly, the distribution of the blocks to the datanodes is not stable or uniform, some datanodes will receive more blocks than others. Finally, the duration of the tests varies significantly, even between the same test in the same version. For that reasons, we repeated the same benchmark in a one-datanode cluster. In that case we have the same hardware and a unique block distribution. Nevertheless, the duration remains variable, but in this benchmark it does not make difference because we investigate the CPU usage until the task is done and no other process runs on the machine. We got the results, which are shown in **Figure 5.4**, and which verify the previous results. Furthermore, the system profile (CPU and net usage) of the datanode during the write part of the benchmark is given in **Figure 5.5**. From this figure, we observe that the optimization in CPU cycles is translated in an around 9% of the total CPU usage as stated by the output of the "top" unix command.

The reduction during the write operation is due to the fact that we reduced the numbers of copies[1] as it was described in the implementation chapter. Furthermore, normal Hadoop implements a streaming protocol which includes header processing and checksums[2]. This protocol was completely replaced in our implementation, and therefore, a percentage of the CPU was released.

Concerning the reading part, the read operation in the original version is already optimized given that in the datanode side the block is sent using zero-copy. The optimization we observe, takes place due to the protocol dropping. On the client side, no significant changes were made, mainly because we had been restricted to the HDFS API, and hence no significant optimization was expected. Furthermore SoftiWARP does not provide zero-copy on the receiving side.

[1] in client side, the copies were reduced from 4 to 1

[2] the implementation of the protocol is defined here http://svn.apache.org/repos/asf/hadoop/common/trunk/hadoop-hdfs-project/hadoop-hdfs/src/main/java/org/apache/hadoop/hdfs/server/datanode/BlockSender.java




**(a)** **(b)**

**Figure 5.4:** CPU Usage, Write/Read 2048 MB File in a 1DN Cluster (=64blocks of 32MB)

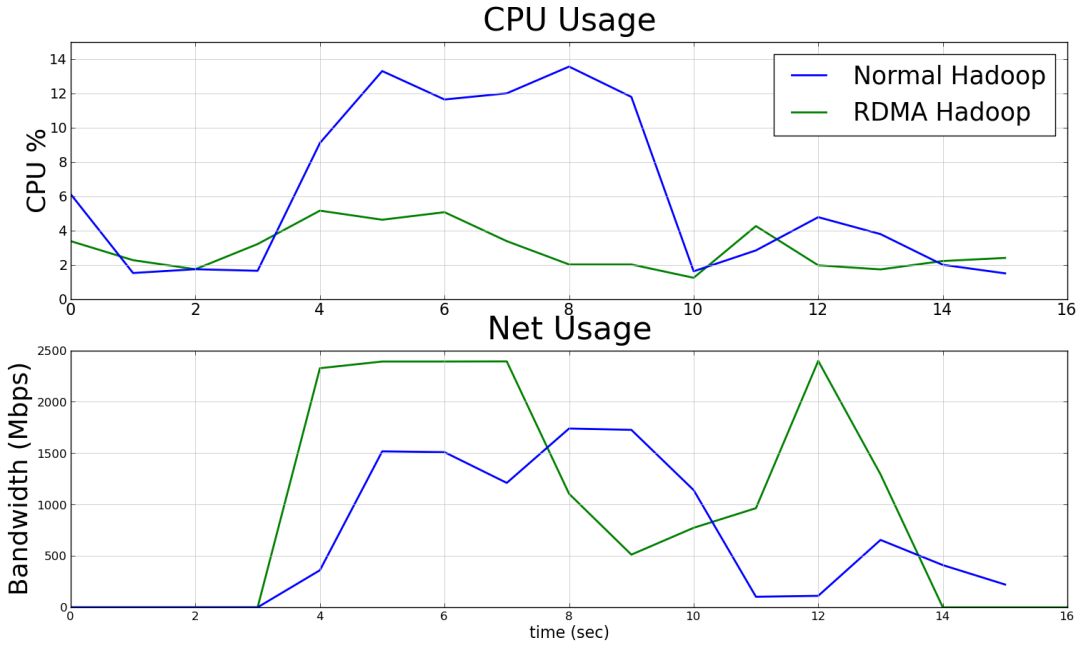



**Figure 5.5:** Datanode CPU/Net Usage, Write Operation of 2048 MB File in a 1DN Cluster

### 5.2.4 Micro-Benchmarks

In this benchmark, we assess the performance in terms of write/read latency. For that purpose, we choose to perform reads/writes micro-benchmarks instead of a real case workload. Micro-benchmarks provided us with a controlled environment. Using real case distributed workload (e.g., a MapReduce task) imposes a number of other variables which define in greater extent the performance and which

have not been examined[3]. Our micro-benchmarks were based on Jira HDFS 1324 "Quantifying Client Latency in HDFS", where synthetic benchmarks, which emulate real HDFS traffic, were introduced.

In the following listings we explain exactly what are the read/write operations we benchmarked, where are the timers, how we create significant workload. Specifically, a Java HDFS client was created, this client performs the following: one write operation (as described in **Listing 5.7**), five read operations (as described in **Listing 5.6**), these two steps are repeated (as described in **Listing 5.8**). Briefly, a client writes a file, then reads the file five times in the row, this iteration takes place a number times to increase the number of measurements. Every time the results were saved in CSV file for analysis. An external bash script (**Listing 5.9**) creates many clients in parallel or sequential. The benchmark took place once in a cluster of one datanode without replication, and once in a cluster of three datanodes with replication factor of three. The file, which is being written and read, is of size 128 MB. The validity of each write was checked in the following way: the content of the file is repeated every 32 MB (equals to HDFS blocksize)[4], thus every block should have the exact same data. If an error occurs, a block with different data will appear in the data folder of a datanode.

```
public void read(String filename){
    FSDataInputStream in = fs.open(new
        Path(filename));

    long start = System.nanoTime();
    byte[] buffer = new byte[1024];
    while (in.read(buffer)>0){}
    end = System.nanoTime() - start;

    latency = (double) end/1000000;
    logger.warn("operation read blocks -
        latency ="+latency+" msec\n");
    s.writeLog2("read",filename,latency);

}
```

**Listing 5.6:** Atomic Read

```
public void writeData(String srcfile,
    String dstfile)
{
    Path src = new Path(srcfile);
    Path dst1 = new Path(dstfile);

    long start = System.nanoTime();
    fs.copyFromLocalFile(src , dst1);
    end = System.nanoTime() - start;

    latency = (double) end/1000000;
    logger.warn("operation write data -
        latency ="+latency+" msec\n");
    s.writeLog2("write","",latency);
}
```

**Listing 5.7:** Atomic Write

```
connect();
for(;c<=iterations;c++){
    String rname=random();
    writeData("/home/X/nfs/file128MB",
        rname+"toRead");
    read(rname+"toRead");
    read(rname+"toRead");
    read(rname+"toRead");
    read(rname+"toRead");
    read(rname+"toRead");
}
disconnect();
```

**Listing 5.8:** Java Client reads/writes

```
#!/usr/bin/bash

for ((i=1; $i<=$1; i++ ))
do

    echo "Client Number x=$x"
    #sequencial
    hadoop jar libRdmaClient.jar  client
    #or parallel
    hadoop jar libRdmaClient.jar  client &

done
```

**Listing 5.9:** Creates many clients

The results are presented in **Figure 5.8** for the write case, and in **Figure 5.6** for the read case. Furthermore, in our implementation we investigated further the delay we measure. By filling the code with timers in critical points, we identified in which parts the delay is divided (**Figure 5.7** and **Figure 5.9**). The explanation of these figures can be found in the sequence diagrams presented in the previous chapter (**Figure 4.5, 4.4**).

[3]A slide presenting in a summarized way the tunning factors of Hadoop can be found here `http://developer.amd.com/zones/java/assets/HadoopPerformanceTuningGuide.pdf`

[4]the file was created by executing : cat file32MB file32MB file32MB file32MB > file128MB

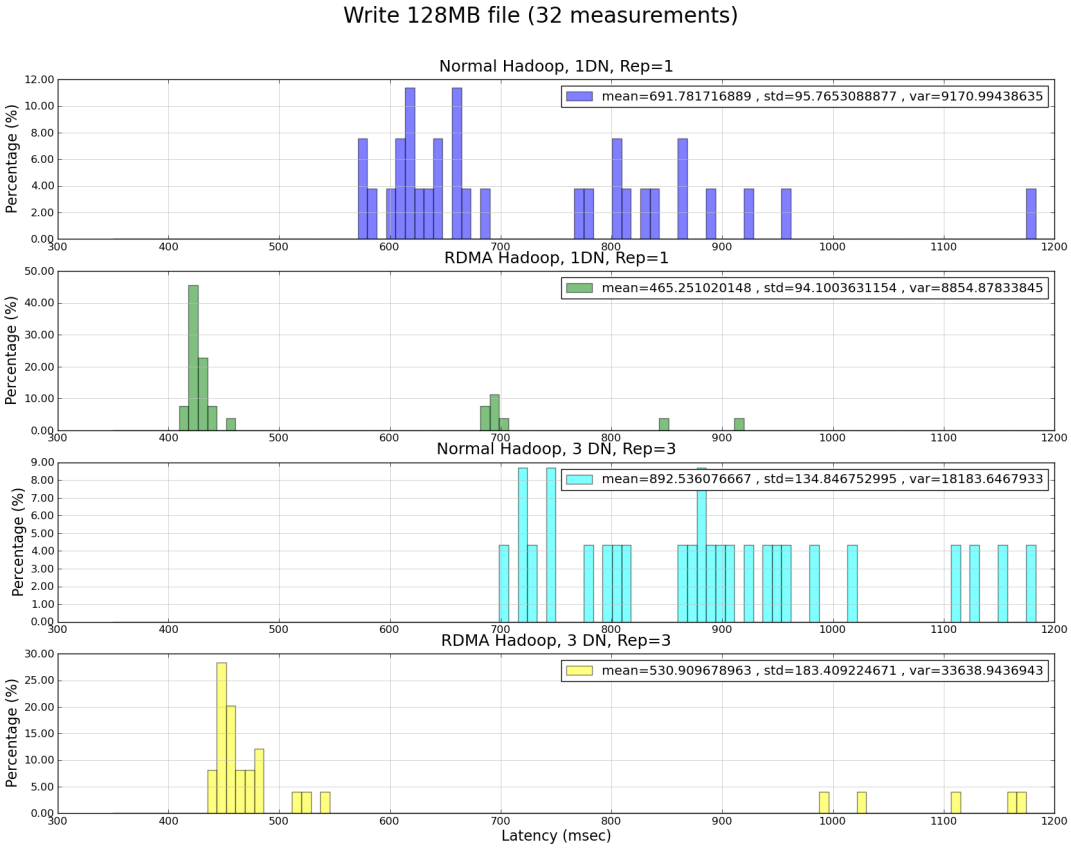



**Figure 5.6:** Write Latency Histograms

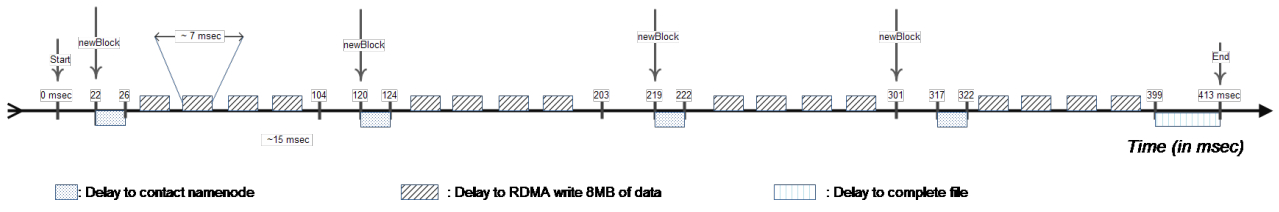



**Figure 5.7:** Write Delay Details (128MB File in 32MB block in 8MB packets)

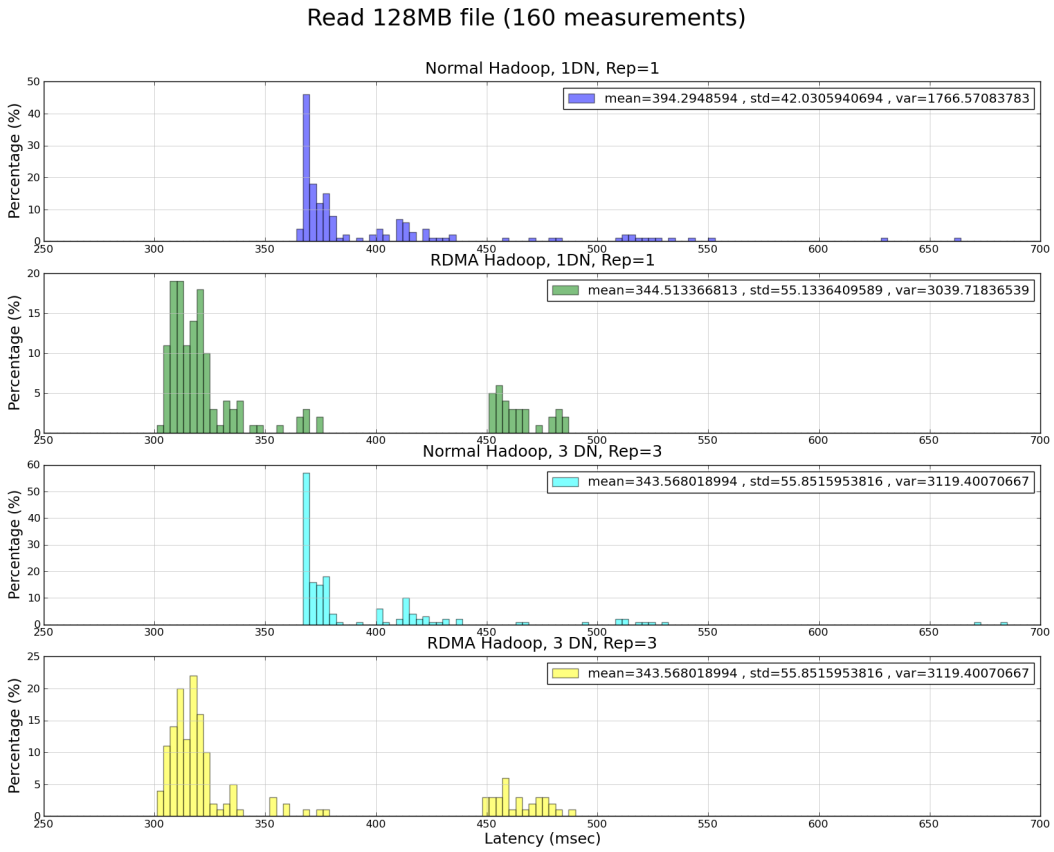



**Figure 5.8:** Read Latency Histograms

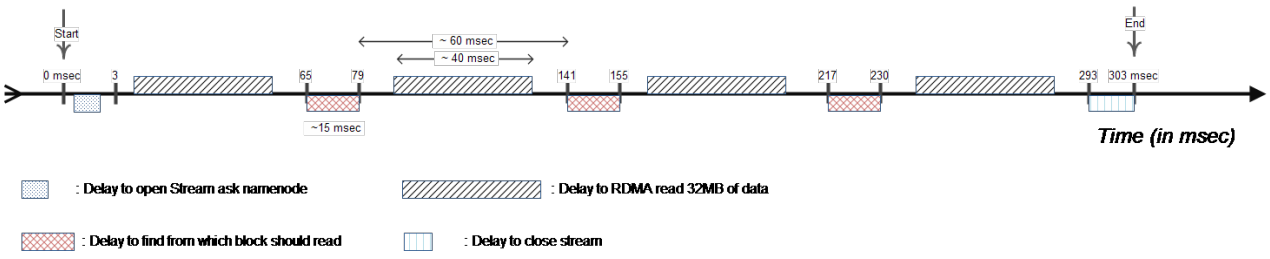



**Figure 5.9:** Read Delay Details (128MB File in 32MB block)

In the write case, we observe around 196 ms gain, which is an around 30% optimization. Executing the write operation using replication does not add a significant delay, nevertheless we get some large values (> 1200 ms), which indicate that few errors occurred. In the read case, the gain is around 50 ms, which is an around 12% optimization. Given that no specific hardware was used the results are satisfactory.

In this benchmark, we should pay attention that in the normal Hadoop version the data is not read from the disk, but it is read from the memory due to the OS caching. Through the monitoring script, we verified once that the disk is not being used during the read operation. Nevertheless, because comparing an in-memory-based to a disk-based implementation, would be completely false, we repeated the benchmark storing the Hadoop data in a ramdisk. In that way, without doubt everything is in memory. In the **Listing 5.10**, we show how the ramdisk was prepared. We got the exact same results as before.

```
$ sudo mount -t tmpfs -o size=256M tmpfs /home/X/hadoop/hddata
$ mount|grep hadoop
 tmpfs on /home/X/hadoop/hddata type tmpfs (rw,size=256M)
$ dd if=/dev/zero of=/home/X/hadoop/hddata/file128MB  bs=8M count=16
 16+0 records in
 16+0 records out
 134217728 bytes (134 MB) copied, 0.0768747 s, 1.7 GB/s
$ dd if=/dev/zero of=~/file128MB  bs=8M count=16
 16+0 records in
 16+0 records out
 134217728 bytes (134 MB) copied, 0.178931 s, 750 MB/s
```

**Listing 5.10:** Ramdisk

### 5.2.5 Scalability

In this benchmark, two aspects of the implementation are investigated. Firstly and most important, whether a datanode can serve simultaneously in the same performance many HDFS clients. Secondly, whether a physical node can handle many client instances at the same time. For that reason, we executed the following test: a file (128 MB in 32 MB block) was saved in a single datanode. A variable number of clients try to read the file from the node. The implementation of the client is the same one that was described in the previous benchmarks. Basically, we are running the script in **Listing 5.9** in many machines and with a variable number of instances. We monitored the latency as a function of the clients. Furthermore, we monitored the link utilization, because it gives a clue up to which point the datanode is expected to serve fast the clients. The results are shown in **Figure 5.10**. The cluster setup was in that case one namenode, one datanode, and client instances distributed in 3 machines.

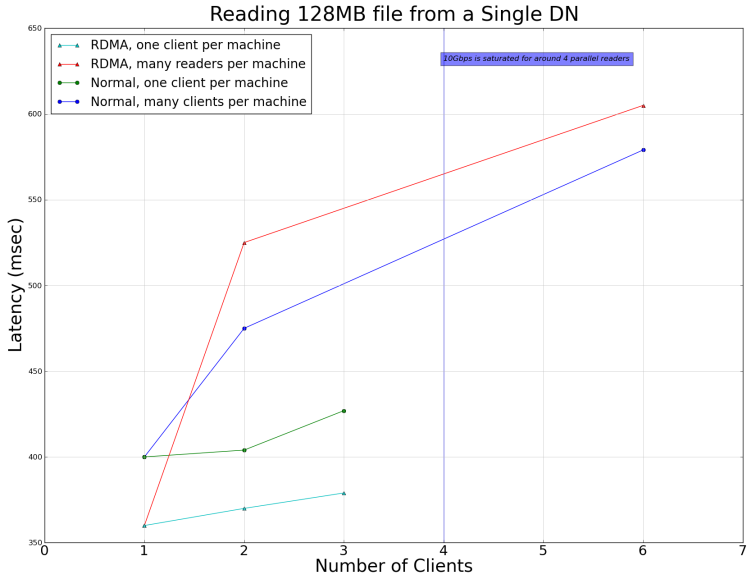



**Figure 5.10:** Scalability: Latency Vs Number of Clients

Concerning the first aspect we can say that a datanode can serve efficiently the readers. We reach this conclusion because the latency, when we set up one client per physical machine, remains almost the same up to the point the link is saturated. As far as the second question is concerned, we observe that a second instance of a client in the same machine increases the delay in the normal Hadoop as well as in our implementation. Nevertheless, in our implementation the impact of a second a client is by far significant (almost doubles the delay). The conclusion is that the client implementation requires further optimization in order to support scalability. However, further benchmarks on a larger scale should be conducted.

### 5.2.6 Read under Saturation

Our implementation targets at enabling the framework to efficiently perform networking in a 10 Gbps network fabric. In order to assess the CPU performance we did the following benchmark: A number of clients reads for a large time period a file from single datanode. This number is greater than the value which was found in the previous benchmark and saturates with certainty the link. The cluster has one namenode, one datanode, three client machines and around six client instances. In this test, we are not investigating the performance, we just want to measure the CPU usage, when the link is saturated. The results are given in **Figure 5.11**.

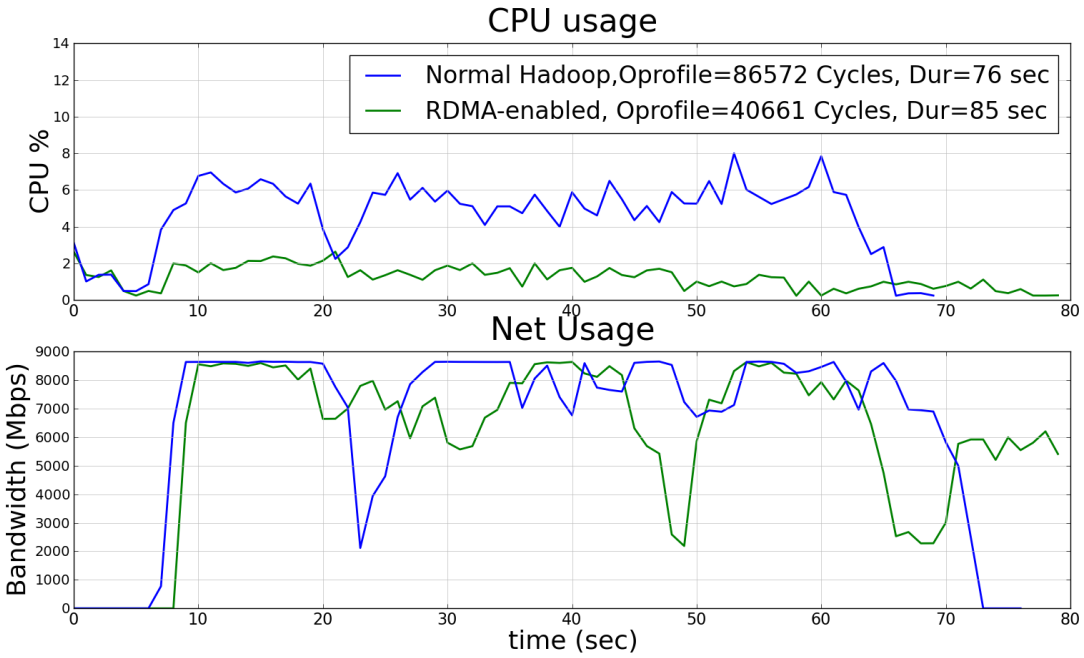



**Figure 5.11:** Datanode Performance under Saturated 10Gbps Network

The observation is that this specific datanode (4x Intel(R) Xeon(R) CPU E5540 @ 2.53GHz) requires approximately 6% of its total CPU power to maintain a data transfers at 10 Gbps. In our RDMA-enabled implementation the percentage is decreased to around 2%. The more accurate results from Oprofile (86572 cycles versus 40661 cycles) indicated the 4% improvement.

# Chapter 6

# Conclusion

## 6.1 Summary

In this master thesis, a prototype of an in-memory RDMA-enabled HDFS system was built. We introduced a range of modifications to the HDFS, which enabled it to work natively in memory. Furthermore, the socket-based communication model was replaced by the RDMA model taking into account the different semantics. Our prototype is designed to work in a cluster of commodity hardware using a software-based RDMA implementation, which enables the system to scale beyond the total cluster physical memory. The implementation was tested and validated in a real commodity cluster.

Evaluating our system was a difficult task, because in a distributed system, there is a wide range of variables and parameters defining the overall performance. However, through a number of experiments in which our RDMA-enabled version was compared to original version, we received interesting results in favour of our system. In particular, during core functions of the HDFS, we decreased the required CPU and reduced execution times.

## 6.2 Future work

Modifying the HDFS distributed file system to fit exactly into Hadoop ecosystem as the original version was a challenging task. Clearly, there is still future work to be done. This work can be divided into the tasks which are required to complete the HDFS functionality, and into the tasks which will contribute further to the performance.

First, to the best possible extent, the core functionalities and mechanisms of HDFS were kept. Nevertheless, the implementation requires significant programming work to be considered complete. An indicative list follows:

- Complete all the error mechanisms that exist in HDFS: An example of what we have not implemented as taken from the HDFS documentation [15] "During reading, if the DFSInputStream encounters an error while communicating with a datanode, it will try the next closest one for that block. It will also remember datanodes that have failed so that it doesn't needlessly retry them for later blocks".

- Complete all possible actions that can be ordered by a client or the namenode. For example, the transfer block command or the deletion command given by namenode to a datanode has not been implemented. Completing this functionality includes extensive validation testing in a large cluster.

- Monitoring/Controlling Memory usage. In the normal Hadoop, namenode can monitor for disk usage, can use a specific percentage of the disk, etc. The same functionality should be ported to our version.

- Make the data persistent. In our implementation, the blocks are saved to the disk with a name different than the block ID. This mapping is not saved persistently. In order to be able to reboot the system and maintain the data, further implementation is required (e.g., rename the blocks during shutdown).
- Implement efficiently the random read operation.

From performance and scalability point of view, there are a number of ideas that can be implemented.

- Hadoop's performance, as we have seen, depends heavily on data locality. Our implementation focuses on the case where the client reads/writes the data from a remote location. The local case should be investigated and modified as well. This task will require HDFS API modification, e.g., when a client wants to read locally, HDFS should return a memory address and make sure that the client process can access this address[1]. The above optimization is crucial in order to observe performance optimization in a MapReduce task.
- Small file issues: in our system, datanode preallocates a block-sized buffer in memory in order to accept a block from a client. If the incoming data is smaller than the block size, then the rest of the buffer remains unused. Given that during a MapReduce task, several files of size around 100 Bytes are saved into HDFS, this is an important scalability issue. Therefore, mapped byte buffers should be resized to fit the data releasing the rest of the memory.
- During every operation, clients access an information stored in the namenode. The time the namenode needs to respond is quite significant and depends on how many clients are connected simultaneously. Given that everything in namenode is in RAM, there might be a way to have a client access the information through RDMA one-sided operation faster and without putting an extra burden on the namenode.

[1] datanode and MapReduce client are two different processes which should share the memory

# Bibliography


[1] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. In SIGOPS OSR. Stanford InfoLab, 2009.

[2] A. Donnelly G. O'Shea A. Rowstron, D. Narayanan and A. Douglas. Nobody ever got fired for using hadoop on a cluster, 2012.

[3] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[4] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In Proceedings of the 7th USENIX conference on Networked systems design and implementation, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[5] Pavan Balaji. Sockets vs rdma interface over 10-gigabit networks: An in-depth analysis of the memory traffic bottleneck. In In RAIT workshop 04, page 2004, 2004.

[6] Jiuxing Liu, Dan Poff, and Bulent Abali. Evaluating high performance communication: a power perspective. In Proceedings of the 23rd international conference on Supercomputing, ICS '09, pages 326–337, New York, NY, USA, 2009. ACM.

[7] Softiwarp: www.gitorious.org/softiwarp.

[8] Animesh Trivedi, Bernard Metzler, and Patrick Stuedi. A case for rdma in clouds: turning supercomputer networking into commodity.

[9] Apache Hadoop: hadoop.apache.org.

[10] Jeffrey Shafer, Scott Rixner, and Alan L. Cox. The hadoop distributed filesystem: Balancing portability and performance.

[11] Rajagopal Ananthanarayanan, Karan Gupta, Prashant P, Himabindu Pucha, Prasenjit Sarkar, Mansi Shah, and Renu Tewari. Cloud analytics: Do we really need to reinvent the storage stack?

[12] Gluster FS: http://www.gluster.org/.

[13] Using lustre with apache hadoop: http://wiki.lustre.org/images/1/1b/hadoop_wp_v0.4.2.pdf.

[14] Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, and Dhiraj Sehgal. Hadoop acceleration through network levitated merge. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pages 57:1–57:10, New York, NY, USA, 2011. ACM.

[15] Tom White. Hadoop: The Definitive Guide. O'Reilly Media, third edition, 2012.

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. Commun. ACM, 51(1):107–113, January 2008.

[17] Slide deck on mapreduce from google academic cluster, tinyurl.com/4z16f5. available under creative commons attribution 2.5 license.

[18] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[19] Philip Frey. Zero-Copy Network Communication: An Application Study of iWARP beyond Micro Benchmarks, 2010.

[20] Rdma consortium completes verbs specifications: ww.rdmaconsortium.org.

[21] Philip Werner Frey and Gustavo Alonso. Minimizing the hidden cost of rdma.

[22] Wimpy nodes with 10gbe: Leveraging one-sided operations in soft rdma to boost memcached.

[23] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In Proceedings of the ACM SIGCOMM 2009 conference on Data communication, SIGCOMM '09, pages 39–50, New York, NY, USA, 2009. ACM.

[24] jverbs: Java/ofed integration for the cloud: https://www.openfabrics.org/ofa-documents/presentations/doc_download/517-jverbs-javaofed-for-cloud-computing.html.

[25] Oprofile: A system profiler for linux: http://oprofile.sourceforge.net.