Università
Ca'Foscari
Venezia

MSc in Computer Science

Master Thesis

Evaluating Performance of Hadoop Distributed File System

Supervisor
Dott. Andrea Marin
Author
Luca Lorenzetto
Student ID 840310

Ca' Foscari
Dorsoduro 3246
30123 Venezia

Academic Year
2013/2014

Contents

1 Introduction **3**
- 1.1 Context . . . 3
- 1.2 Motivations and contributions . . . 4
  - 1.2.1 Brief company history and environment details . . . 4
  - 1.2.2 Why HDFS . . . 5
  - 1.2.3 Contributions . . . 6

2 HDFS: Hadoop Distributed File System **7**
- 2.1 Hadoop . . . 7
- 2.2 HDFS . . . 8
  - 2.2.1 *Namenode* and *Datanodes* . . . 8
  - 2.2.2 Organization of data . . . 10
  - 2.2.3 Data Replication: selection and placement policies . . . 11
  - 2.2.4 Input/Output operations on HDFS . . . 15

3 HDFS Performance **17**
- 3.1 Known Issues . . . 17
  - 3.1.1 Software architectural bottlenecks . . . 17
  - 3.1.2 File Size Problem . . . 18
  - 3.1.3 Single Namenode . . . 18
  - 3.1.4 Communication channels . . . 19
- 3.2 Open problems . . . 19
  - 3.2.1 Replica selection policy . . . 20

3.2.2 High resource popularity . . . 21
3.3 Performance evaluation methods . . . 22
3.3.1 Practical performance analysis model . . . 23
3.3.2 Modelling HDFS using CPN . . . 24
3.4 Considerations . . . 24

4 Modelling Formalism **27**
4.1 PEPA . . . 27
4.1.1 Language outline . . . 28
4.1.2 Syntax and semantics . . . 28
4.1.3 Underlying Markov Process . . . 29
4.2 PEPA$_k$ . . . 30
4.2.1 PEPA$_k$ syntax . . . 31
4.3 Performance Measures for M/M/1 queues . . . 32
4.3.1 Mean number of customers in a queue . . . 34

5 HDFS System Model **35**
5.1 Working Scenario . . . 35
5.2 Model Components . . . 36
5.2.1 Client . . . 36
5.2.2 *Datanode* . . . 37
5.2.3 Complete System . . . 37
5.2.4 Why no *namenode*? . . . 37
5.3 PEPA$_k$ HDFS System model . . . 37
5.3.1 *Datanode* queue . . . 38
5.3.2 Client . . . 38
5.3.3 System . . . 39

6 Placement probability **41**
6.1 Replica placement probability . . . 42

7 Model Analysis **45**
7.1 Mean number of clients requesting from *datanode* . . . 45
7.2 Mean number of clients being served by the entire cluster . . . 46

8 Proposed solution **49**
8.1 Improvements to the system . . . 49
8.2 Balancing algorithm . . . 50
8.2.1 Idea . . . 50

8.2.2 Algorithm 51
8.2.3 Experimental implementation 52

9 Results **55**

10 Conclusions **59**

Abstract

In recent years, a huge quantity of data produced by multiple sources has appeared. Dealing with this data has arisen the so called "big data problem", which can be faced only with new computing paradigms and platforms. Many vendors compete in this field, but at this day the de-facto standard platform for big-data is the opensource framework *Apache Hadoop* . Inspired by Google's private cluster platform, some independent developers created *Hadoop* and, following the structure published by Google's engineering team, a complete set of components for big data elaboration has been developed. One of this components is the *Hadoop Distributed File System*, one of the core components. In this thesis work, we will analyze its behavior and identify some action points that can be tuned to improve its behavior in a real implementation. An hypothetical new balancing algorithm, using a *model driven dynamic optimization approach*, is proposed and its effectiveness is tested.

1 Introduction

In recent years, the increase of the ways for collecting data and the speed at this data are generated, has shown the necessity of finding a way to process them in a feasible time. This is the so called *"Big Data"* problem [1].

1.1 Context

As *Big Data* problem we refer to a set of problems that are related to the elaboration of datasources of huge size (but not only size). *"Big Data"* is any attribute that challenges the constraints of a computing systems. Gartner [1] refers as the big data with the "3V": high volume, high velocity and high variety data that need a new way for being processed. So we can say that big data is not only big for size, but also big for the high volume of data generators (i.e. a small quantity generated by a big variety of sources). The speed and the size of the currently generated data can be explained by this percentages: 90% of the data generated since the beginning of the time has been produced starting from 2010. Also interconnection between the collected data is relevant and finding the correct relation is a challenging problem that requires high quantity of computing resources. Differently from previous business intelligence systems that were using only descriptive statistics for detecting trends and measures, big data is used to reveal relationships, dependencies and predictions using inductive statistics for inferring laws.

For analyzing this amount of data, specific technologies are required. Storing and retrieving data is crucial due to the velocity and volume this data can be generated and needs to be elaborated. Consider, i.e., the famous LHC

experiment at CERN: this is the most famous *"Big Science"* experiment (scientific big data generator). Data are generated at enormous speed of more than 500 exabytes per day. It's evident that commonly used business intelligence or data mining applications are impossible to use with this big mass of data.

In private sector a similar problem has emerged at *Google* around the beginning of 2000's. Their internal engineering team developed an entire framework for dealing with this emerging problem, called *MapReduce*. The framework is still actively used and developed inside Google, but hasn't never released as generally available product. However, in [2] and [3], *Google's* engineering team made public their work, inspiring the start of some alternative implementations. The main and most famous implementation is an opensource one called *Hadoop*, started at *Yahoo! Research* by Doug Cutting in 2005, as support framework for the opensource search engine *Nutch*.

1.2 Motivations and contributions

The analysis done in this thesis work has started from a company request. The Technological Division of InfoCamere asked for explorative survey of the *Hadoop Distributed File System (HDFS)* and its functionalities. After the identification of limitations and issues affecting HDFS, some improvements have been proposed.

1.2.1 Brief company history and environment details

InfoCamere's business is related to the IT services provided to the system of Italian Chambers of Commerce, of which is integral part. In it's history has realized and now manages the IT system connecting each other, through an high speed network, 105 Chambers of Commerce and their 235 branch offices. The main activity is the preservation and the management of the information assets of the chambers, which contains, among other things, the italian business register, the repository financial balances of the registered companies and the italian register of patents and trademarks. Through the service called "Registro Imprese" (Company Register) is possible to gain access to part of the informative assets and to tools used for making administrative practices that needs to be sent to other italian public offices. Another relevant service is the service called "Impresa in un giorno" (Company In A Day) that is a frontend service for all the requests needed for opening a company in

Italy. All the InfoCamere's services are offered to a big user base, that are more than 6 millions of businesses and all the public administrations, both central and local.

1.2.2 Why HDFS

Infocamere's storage infrastructure has over 1600 TB of capacity, with an average usage of about 70%. All this infrastructure is object of a load of more than 23 million transaction per day and over 17 million of daily web contacts. This means that there is the need of a continuous update of the technologies used in the datacenter to maintain an high levels of service.

Recently the technology division started evaluating technologies for storing big data. The natural choice fell on the de-facto standard platform, *Hadoop*. Since at the moment they are not interested in computing part, the evaluation has been restricted only on the file system. The requirements to be satisfied are quite common:

- reliability, avoid data loss;
- fast retrieval of data when required;
- ability to scale easily adding nodes for extending storage capacity and improving general performance and reliability of the system;
- storage of very big quantity of data of any size, starting from very small to huge files.

These requirements are satisfied by HDFS.

An hypothetical use case is as long term storage of company certificates generated by the web application of "Registro Imprese". Every time a user requests the certificate of a company, a pdf is created with the informations at the time of the request. This file has to be preserved for a certain time, since a user can download many time that file and no information has to be updated until the user asks for a new certificate. Due to the high volume of data generated and low number successive requests of the same file, Infocamere wants to store this data on an online filesystem, but with a lower cost per gigabytes than standard NAS storages.

1.2.3 Contributions

With the analysis, some additional limitations and problems to the already presented in literature has been found. The exposed problems are related to the usage of HDFS in a scenario like the one InfoCamere has expressed interest in implementing. For online, or near online, usage of HDFS there are surely required improvements. In this thesis, two problems are exposed and in particular one of the two is approached with a proposed solution. The problems are exposed in 3.2. The possible solution proposed for the second problem makes use of a *model driven dynamic optimization* approach, where the system model is solved for searching a system configuration fitting best the performance objectives the HDFS cluster has to satisfy.

2 HDFS: Hadoop Distributed File System

2.1 Hadoop

Hadoop is a framework for large-scale processing of big datasets on so-called *"commodity hardware"*. The entire framework has been released as opensource and is composed by different modules. One of these is the *Hadoop Distributed File System*.

Hadoop project is a project started with the aim of improving the performance of the *Nutch Web search engine*, implementing a version of the *Google's MapReduce* engine, which has been described in [2]. Together with the *MapReduce* engine they created also a distributed file system called *Google File System*, with the particularity of permitting efficient and reliable access to huge quantity of data. This is not a real file system, but is a "user level" file system, running as a service that can be accessed using API and libraries.

The project was initially developed inside *Yahoo!* that was the first company implementing and using the Hadoop framework in a production environment. In recent years, many companies are implementing Hadoop clusters from small to huge size to process big datasets, frequently also for core business applications. Also other components has been developed on top of Hadoop for increasing possible applications, mainly by companies those main business strictly related to big data, like *Facebook* and many other social network platforms or computing companies like *Cloudera*.

2.2 HDFS

The *Hadoop distributed File System* is the primary storage of an *Hadoop Cluster*. The architecture of this file system, described in [4], is quite similar to the architecture of existing distributed file systems, but has some differences. Main objective of HDFS is to guarantee high-availability, fault tolerance and high speed I/O access to data. Also provides high throughput in access to data, especially in streaming access, since HDFS is designed mainly for batch processing. Another relevant functionality is the ability to grow on demand keeping low the overall cost, due to the commodity hardware that composes the cluster.

In [4] another key goal is highlighted and is the most relevant: high reliability of the system. Since hardware failure in a cluster composed of an high number of nodes is the norm and not an exception, HDFS has been developed keeping in mind this concept. Through heartbeat mechanisms, replication, and replica placement policies, the HDFS guarantees resistance to faults, with quick and automatic recovery from failures and automatic switch of client's requests to working nodes. To simplify this process, a *write-once-read-many* access has been implemented for files, providing an elementary coherence model.

2.2.1 *Namenode* and *Datanodes*

An HDFS cluster is composed by two types of nodes: *namenode* and *datanode*. Usually there is one *namenode* and multiple *datanodes* spanning on multiple racks and datacenters. In following section their role and how they work is explained.

Namenode

The role of *namenode* in the cluster is keeping the *file system namespace*. It contains the file and directory hierarchy and file locations, stored as a tree. Since in HDFS files are divided into blocks and these are replicated multiple times, *namenode* contains also the list of *datanodes* containing each single block. Like on classic file systems, in HDFS files are represented by *inodes* that store permissions and any other interesting metadata like access times and namespace quota.

Namenode is the first node a client contacts when wants to do an input/output operation. The *namenode* replies to client's requests with the list

of *datanodes* it has to interact with, sorted using a *replica selection policy* for reads and a *replica placement policy* for writes. How this policies works is explained in following sections.

Another relevant task of the *namenode* is keeping track of the state of HDFS cluster, receiving the heartbeats of the *datanodes* and verifying if the file system is consistent, checking which blocks need to be replicated and, in case, initiating replication when needed.

By design, there is only one *namenode* and data never flow through it. This is a single point of failure, but *namenode* should be never overloaded and should be the most reliable node of the cluster because, without *namenode*, HDFS is totally unserviceable. In recent *Hadoop* releases, there is also a *backupnode*, always up to date with latest namespace status. It receives all the operations done by *namenode* and stores them in local memory. This permits to have the latest, up to date, namespace status when *namenode* fails. This node can be considered as *read only namenode*, since can do all the operations of the *namenode* that don't require knowledge of the block locations (that are known only by *namenode* due to *block report* messages by *datanodes*) [5].

Datanode

A *datanode* is a cluster member with the role of containing file blocks on its local disk, and serving them to clients. It manages its locally connected storage and is responsible for serving read and write requests and managing creation, deletion and replication with the supervision of the *namenode*. *Namenode* instructs the nodes on what to do for keeping the file system coherent to the desired constraints. *Datanode* stores on his native local file system, that has been configured for HDFS, two files for each block: one contains the data itself and the second one contains block metadata, checksum and block's generation stamp. In a cluster there are many of these *datanodes*, usually one for each member node. This permits to use all the local space available in all the nodes of the *Hadoop* cluster. Each *datanode* joins the cluster registering to the namenode the first time. After the registration, it gets its unique permanent id. Once analyzed the stored metadata of each block, it sends a block report to the *namenode*, which updates the block location table. When online, *datanode* periodically sends an heartbeat message to *namenode*. If *namenode* gets no updates from a *datanode* within 10 minutes, considers it failed and start actions for preserving the service level of each block that were contained in the failed *datanodes*.

Datanodes are also contacted directly by clients for reading and writing blocks, so their behavior is critical for the overall performances of the system [5].

2.2.2 Organization of data

HDFS has been designed for being a filesystem that stores files of big size. This big size has to fit with high throughput and reliability requirements. Clients usually deal with this large datasets, requiring frequent and quick access to data. All this prerequisites are satisfied with the data organization structure implemented in HDFS.

Data blocks

Each file is neither stored contiguously, nor on single node. A file is composed by several blocks of equal size, except the last. The default size of these blocks is 64MB, but is configurable at file level. Each block is, possibly, stored on different *datanodes*, permitting higher throughput either when a client tries to access to the entire file (can obtain multiple parallel access to the blocks of the file) or when multiple clients want access to different parts of the same file (parallel requests to different datanodes).

Writing and Datanodes Pipeline

When a client starts writing data, doesn't contact immediately the *namenode* for adding a block, but buffers file on local temporary file. Then, once closed or reached the size of a block, flushes to the cluster the data stored the local temporary file. The client has to contact the *namenode* for opening in write mode the file and then for communicating the closing. Once done that, the file is stored permanently in HDFS. If *namenode* crashes before the close message, the file is lost. Each block stored has to be replicated for guaranteeing reliability and consistency of data. This replication is made through a pipeline when writing a block. During the write process, a client node requires a list of *datanodes* where to put the block and contacts the first, sending it the list. Once started write operation, the first *datanode* writes on it's local filesystem and forwards what it is receiving also to the second *datanode*, and the second datanode does the same to the following node, up to the last node of the list. This offloads the client from replication management, moving this responsibility to the cluster.

2.2.3 Data Replication: selection and placement policies

An HDFS cluster is composed, as said before, of a lot of *datanodes* and has to provide reliable store of large files divided in multiple blocks across these nodes. To meet this requirements, blocks are replicated for fault tolerance.

It is a *namenode*'s task deciding where putting replicas when a block is written or the number of replicas existing for a certain block is less than the value set. It has also to keep track of positions of replicas for redirecting client's read requests to a set of selected datanodes containing the needed block. *Datanodes* are selected using a *replica placement policy* (for writes) and *replica selection policy* (for reads). These two policies are strongly impacted by the network topology.

Network topology

By definition a cluster is composed by multiple nodes that frequently cannot fit in a single rack and, for business continuity needs, can also span over multiple datacenters. Communication channels between racks and between datacenters have different characteristics with respect to internal rack communication. For sake of completion, recent technologies for hi-density computing like *Blade Servers*, introduced an ulterior grouping, called enclosure or chassis[1]. This generates an hierarchy of the communication speed between nodes.

Intra-chassis communication is the fastest, both for speed and latency. On second level we can place internal rack communication: frequently racks have a top of rack switch interconnecting multiple enclosures/servers standing on the same rack. Third level is *inter-rack* communication or *intra-datacenter* communication. Last level is *inter-datacenter*, that can be done on metropolitan or geographical links, with totally different latencies and speeds on the connecting links.

Hadoop doesn't have knowledge of server's location and doesn't make any measurement to infer the cluster structure. So, by default, considers all nodes belonging to the same rack. This has an evident impact on performances: handling a remote node as the same way of a local node can lead to a performance degradation due to links connecting the "calling node" and the "called node". To avoid this, a topology can be passed to *Hadoop* through a script that, given a certain node, returns its position in the cluster topol-

[1]Enclosure (or chassis) is delegated of making all the non-core computing services like cooling, power and blade interconnection to internal LAN switches

ogy. This script permit nodes to know their position in the cluster and allows rack aware mechanisms. Rack awareness impacts the policies of selection and placement of a block.

Topology has multiple layers, like the example in figure 2.1. Top level is a root node connecting all the datacenters. As leafs of the tree there are all the nodes of the cluster.

This tree based structure permits to measure in a very simple way the topological distance between two elements of the cluster. This distance is very important for node selection in the placement/selection policies: the higher is the distance, the lower is the bandwidth interconnecting the two nodes. E.g. in the figure 2.1, it's possible to compute the following distances:

- *distance(D1/R2/H5, D1/R2/H5) = 0* same node;
- *distance(D1/R2/H4, D1/R2/H5) = 2* different nodes on the same rack;
- *distance(D1/R1/H1, D1/R2/H5) = 4* different nodes on different racks and same datacenter;
- *distance(D2/R4/H12, D1/R2/H5) = 6* different nodes on different racks and different datacenters.

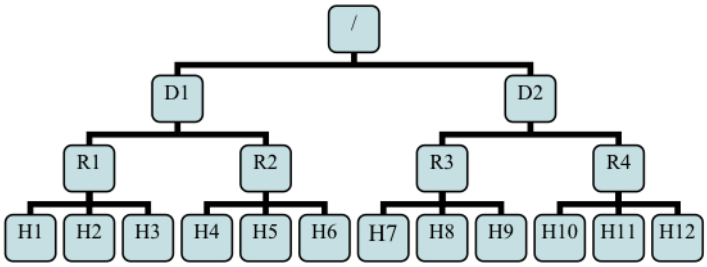

Figure 2.1: An example of Hadoop cluster topology

Replica Placement Policy

The replica placement policy is a set of rules leading to the selection of three nodes (in case of the default replication level) where the replicas of a block will be written. HDFS has a pluggable interface for placing replicas of blocks, so anyone can deploy their rules writing a new class extending the

abstract class provided (`blockmanagement.BlockPlacementPolicy`). A default policy (`blockmanagement.BlockPlacementPolicyDefault`) is already implemented and automatically used. This default policy relies on two simple rules:

1. No more than one replica per node of the same block

2. No more than two replicas per rack of the same block

This allows the blocks to be available on two different racks (or more if the number of replicas is set to a value higher than 3), improving availability and reducing the risk of corruption. The implemented behavior in the default policy places the blocks in this way:

First replica placed on local node if the client is also a *datanode*, else selects a random *datanode*;

Second replica a random rack different from the one containing the *datanode* selected for first block is chosen, and a random node in that rack is chosen;

Third replica a random node on the same rack as the second node different from it is chosen.

Other replicas additional random nodes with restriction that no more than two replicas should be placed on the same rack. This constraint limits the number of replicas to $2 \times number\,of\,racks$

Once the required nodes are selected, they are sorted by proximity to the first replica and a pipeline from the first to the last replica is formed for write 2.2.

The policy works at block level, so multiple blocks of the same file can reside on totally different nodes: there's no relation between where a block is placed and which file the block belongs to. This gives a totally independent random allocation.

This default policy may not be good for all the environments. HDFS's developers suggest [6] some alternative possibilities for the policy, i.e. the usage of *HeatMaps* for placing replicas on less loaded nodes, that could be a good idea for improving performance. At the moment, this possible alternative implementations are not available on main HDFS source code.

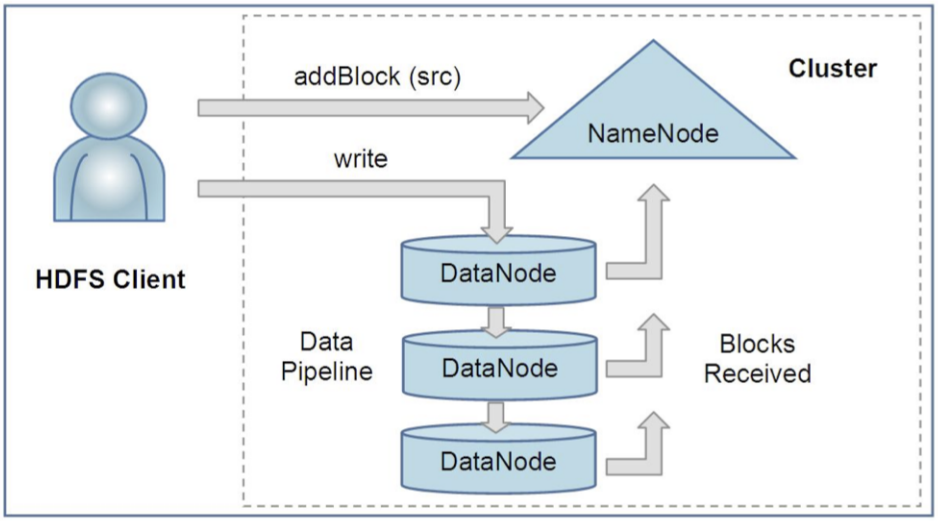

Figure 2.2: Write operation of a block

Replica Selection Policy

The replica selection policy is the rule applied by a *namenode* when a client asks for the location of a block. The *namenode* orders the locations of replicas depending on the client that made the request. The purpose is satisfying reads by addressing the client to the closest node containing the block required. *Datanodes* are preferred in order of proximity: first nodes on the same rack, then nodes on the same datacenter, else a random one (also remote). The sorting algorithm is implemented in `net.NetworkTopology` by the method `pseudoSortByDistance`, which sorts the list of nodes containing a replica in this way:

1. if block is on the local *datanode* (block is stored on the same machine the client is running), swap it with first element;

2. if a node on the same rack as the client is found, swap it with the first element after the local *datanode* or the first element if there's no local *datanode*;

3. if neither, put a random replica at first element and leave the rest of nodes untouched.

If more than one *datanode* are located on the same rack, then one of them is always selected as first local *datanode* until the *namenode* is restarted. This

is due to how the namespace tree is loaded into memory. The list of nodes storing a block is represented by a data structure that has no ordering and, every time is loaded in memory, order may vary.

2.2.4 Input/Output operations on HDFS

To understand better how HDFS manages files, in following sections how read and write operations work is presented.

Read

When a client wants to open a file in read mode, first contacts the *namenode* for fetching the list of the blocks and their location. Locations of each block are sorted by *namenode* before the communication, according to replica selection policy, explained in section 2.2.3. The client then starts with the request of each block, beginning from the first node in the list. Block read can fail for multiple reasons: *datanode* is unavailable, does not longer hosts that block or replica is found to be corrupted when checksum is verified. In fact, when a file is read, the client checks each block's checksum with the checksum value that the *datanode* has sent with the block. If a corrupt replica is found, the client notifies *namenode* and requests the same block from another replica.

Differently from other filesystems, is possible to read a file while is open for writing, due to its simple coherency model that allows only append to an existing file. When the file is open for writing the *namenode* knows only the size of data actually in the cluster, so client gets the list of the blocks that are present before the read request.

Write

As said before, HDFS implements a simple coherency model that allows multiple readers, but a single writer and only append to the file. This model can be summarized as *"single-writer, multiple reader"*. Write operation is exclusive and when a client asks for a write operation has the lease of the file guaranteed only if no write lock is set on that file. Write lock is set by *namenode* every time a write operation is allowed. Lock is guaranteed until the file is closed or a timeout occurs. To avoid the timeout to occur, periodical heartbeats are sent to the *namenode*.

When the client needs to write a block, ask the *namenode* to allocate one with an `addBlock` call (see figure 2.2). The *namenode* replies with an

unique block ID and determines the *datanodes* that will host the replicas. These nodes are sorted in order to minimize the network distance between the client and the last datanode of the list.

Block's data are pushed into the pipeline formed by the nodes using packets of fixed size (typically 64KB). After the packet has been wrote on all the nodes and the client has received all the required acknowledges, starts pushing the next packet. The client also computes the checksum of the block it is writing, and sends it to *datanode* which stores this information with the block, for future verifications.

Data is not visible to other readers until the file is closed, or an explicit flush is requested with the `hflush` operation. All data written up to the flush are then certainly visible by the readers.

3 HDFS Performance

From the public release of the *Hadoop framework*, its performance has been object of study: as said before, the objective of HDFS filesystem is processing massive quantity of data, with high speed I/O. In next section we will see previous works on performances that show some critical points. Then other open problems found during the analysis are exposed.

3.1 Known Issues

3.1.1 Software architectural bottlenecks

In [7], authors identified some bottlenecks connected to the architecture of the platform. First of the presented problems shows that is possible to encounter performance issues due to the access pattern of clients to HDFS. The file system is developed with the target of providing streaming access to files in the filesystem, preferring continuous disk access to random access patterns. Frequently, applications with low computation requirements use this last type of access, decreasing performances. Other performance issues can be related to scheduling algorithm of *Hadoop* in case of big number of small map jobs working on multiple blocks. Since jobs are scheduled every 3 seconds, if the job is very quick, the node will be idle for several seconds, creating the effect of intermittent resource usage. Proposed solutions are the increase of block size, that will allow to require less blocks for each job, or the oversubscription of nodes, that means scheduling more jobs than the available resources, reducing the possibility of idle time. This last proposal may degrade performances in another manner due to resource contention.

Another identified issue is due to portability. All the framework is written in *java* to simplify the portability to multiple operating systems. HDFS assumes that the underlying platform (filesystem and OS) behaves in the best way for *Hadoop*. This is not true, because any platform has its specific behavior, with parameters that are completely out of control of HDFS. In example, I/O scheduling on disk used for storing HDFS data can affect performance. Usually this schedulers are designed for general purpose workloads, trying to share fairly resources between processes. This can be a problem since HDFS requires high bandwidth and schedulers generally makes a tradeoff between bandwidth and latency. In addition also data fragmentation can impact performances when disk is shared with multiple writers. Some filesystems cannot guarantee that an HDFS block can be stored contiguously. The more a *datanode* is running, the higher fragmentation can occur, resulting in a very low on-disk file contiguity.

3.1.2 File Size Problem

HDFS suffers when treating small files. With small file we refer to files those size is significantly smaller than the block size (that is usually 64MB), requiring the same memory usage as a file occupying a full block. The problem is not directly related to the file size, but to the high quantity of this files that will be stored on HDFS. If someone is using *Hadoop* and the average file size is small, probably the number of files is very very high. In [8] the author shows how much memory a *namenode* requires for storing an high number of files. The size calculated is based on a rule of thumb [9], stating that each file or directory stored in HDFS will usually require 150kb of RAM on *namenode*.

Many solutions for this issue are present, but are related to the use of HDFS that the client will make. In [10] and [11], some solutions are presented. In [8] a possible solution for Content Addressable Storage is shown. Many works has been published on this issue, making the developers, in [5], to plan a future revision of *namenode* structure.

3.1.3 Single Namenode

By design, an HDFS cluster is provided of only one *namenode*. This is a known infrastructure issue, that configures *namenode* as single point of failure. Many improvement has been done since the first cluster implementation, providing *secondary (backup) namenode* and *checkpoint namenode* (explained

in [5]). All this improvements had the objective of increasing the overall reliability of the system, since this components work in an active/standby way. Instead, speaking of performances, no improvement is provided, since all the clients will require to contact this single component for their operations on the filesystem. In recent versions, HDFS federation has been added, providing the ability of multiple namespaces on the same cluster. Each namespace has its own *namenode*. With this approach, *datanodes* are used as common storage for blocks by all the *namenodes*, and multiple *namenodes* can work in the cluster without the need of synchronizing each other. This can aid reducing the load of a single *namenode*, splitting the namespace in multiple parts, but clients that are all concurrently working on the same namespace still sends requests to the same *namenode*, possibly overloading it. In [5], HDFS authors in "Future works" acknowledge this problem and say they're planning to explore other approaches such as implementing a truly distributed *namenode*.

3.1.4 Communication channels

Another performance impacting problem that has been approached is related to communication channels. *Hadoop* was born and has been developed to run on commodity hardware, so communications happens using the widely present Ethernet protocol, with TCP/IP. Instead, authors of [12] propose to use InfiniBand [13], a computer network communications link used in high-performance computing and enterprise data centers. InfiniBand has higher performances than the common 10GbE communication channels, and can be used both with IP protocol (IP over InfiniBand) and with more powerful RDMA (Remote Direct Memory Access). With respect to the actual implementation, using JNI and Java Socket, RDMA provides high-throughput, low-latency networking, which is especially useful this type of environment where high quantity of data are massively processed.

3.2 Open problems

Reading architectural and implementation documents about HDFS we found out some possible intervention areas for optimizing HDFS performances. In particular, we focused on the *read* operation, that happens with more frequency than write operations. In next sections the two problems are shown. Modifications proposed on previous works, shown in previous section, are not

taken in account. This because most of them are, in current versions, not implemented and may require deep architectural revision (e.g. implementing fully distributed *namenode*).

3.2.1 Replica selection policy

In 2.2.3, how the replica selection policy works is explained. Even if is not clearly understandable from documentation and HDFS code, we found out that the policy doesn't behaves in optimal way.

The factor influencing the selection of a certain datanode is related only to the topological distance and no other factor are kept in consideration. If the client requesting the resource resides on a *datanode*, this is clearly not a real problem, but the bad behavior appears when more than one *datanode* resides on the same rack of one or more clients. In this case, even if there are more than one *datanodes* at the same topological distance, the *namenode* directs all the clients to the same *datanode*, that could overload. Until a *namenode* restart, the same request from the same node will obtain the same response, since no randomization happens. After the restart, the list of *datanodes* could be sorted in another way, but this not because of particular mechanisms, but only because when the *namenode* restarts, loads from disk the file system namespace, so the list of *datanodes* containing a block can be loaded in memory with a different order. In our scenario, clients are directed to a random node containing the block, but we can consider this behavior not optimal, since the possible *datanode* is randomly selected without considering any other characteristic of that node (i.e. the actual job load or the length of request queue).

This default behavior of replica selection policy could stress more some *datanodes* with respect to other *datanodes*. If *datanode* could share their load with the *namenode*, replica selection policy could take in account also this information when sorting the list of nodes.

Another issue that we found out is that, differently from replica placement policy, read selection policy is not configurable and, for implementing new features or different behaviors, requires modifications to HDFS core source code. This is undesirable because will create a fork from the main core source code that could be lost with updates and require manual intervention every time the cluster is upgraded to newer releases, requiring patching and code adaptation to new version.

A possible solution for the latter problem is implementing a common in-

terface for developing custom policies, like the work that has been done for the replica placement policy and explained in [6].

For the first issue, instead, we need to establish a communication channel between *datanodes* and *namenode* to inform it about the status. In [5], we see that communication from *datanodes* with the *namenode* already happens, and is done through heartbeat messages sent every 3 seconds. These messages has the main objective of communicating the vitality of the node to the *namenode*, that keeps track of the status of the machine and marks it as unavailable it after 10 minutes of missing heartbeats. Heartbeats also carry other informations about overall health status of the server: total vs. free capacity and data transfers currently in progress. Including other informations, like the number of client in queue, we can improve the possibility of balancing better the load between various datanodes that can serve the same resource. In this way the *namenode* can sort the list of *datanodes* according to a *join-the-shortest-queue* policy [14, 15].

The evaluations of possible solutions involving *join-the-shortest-queue* policies for queue management is possible only through simulation. This approach doesn't fit with dynamic optimization, like the method next problem, and will involve an high resource usage for complete model simulation.

3.2.2 High resource popularity

This second problem, which will be analyzed in last part of this thesis work, is related to the growth of requests for a certain resource. As said in previous chapter, the default number of replicas is set to be 3. Considering our scenario, where the replica the client will read is selected randomly, when a resource becomes very popular, there is an high probability that two client will request the same block from the same node. If the resource becomes even more popular, there is the possibility of sensible waiting in queue for being served by *datanode*.

HDFS cluster has a balancer task that runs periodically or on request that has the objective of keeping resource equally distributed on the cluster. When finds that a node has more disk usage than the others on the cluster, starts moving blocks to other nodes to keep disk usage more or less equal in all nodes in the cluster. When a new node is added to the cluster, the balancer moves some block to that node, balancing the resource usage.

Also in this case, no operation are done with the objective of increasing overall system performance. Balancer doesn't have knowledge of resource

popularity and makes equilibrations considering only disk space utilization. Its intervention doesn't help much the overall performances, it could increase or could not (i.e. when moving a popular block to an overloaded node).

Additionally, the balancer only moves data but, instead, could be interesting that, if informed of popularity of a block, could increase the number of replicas of the block, permitting a better equilibration of clients between datanodes.

Like replica selection policy, also balancer is not configurable for taking in account different parameters than disk utilization.

3.3 Performance evaluation methods

To check how the issues impact on the overall performance of HDFS, we need to obtain some measures for the comparison between the implementation with before and after applying contermeasures. A first approach can be the use of a benchmark suite, like in [16]. At the moment the most popular benchmark is called *TestDFSIO*, based on a *MapReduce* job, but other benchmark suites exists, and are shown in [17].

Using benchmarks for checking possible solutions for the issues we identified in previous section is unfeasible, since requires the implementation of the possible fixes in *Hadoop* source code, which is time expensive and can lead to unexpected behavior that can be related to new bugs introduced in software.

Our approach, instead, uses an abstract representation of the system, called "model". Model represents the essential characteristics of the system, so that can be used for reproducing system's performance. With the model we can focus on particular measures of our interest and evaluate how the system behaves when changing some parameters.

Representing the a system with a model requires the in depth analysis of how it works and this, sometime can be unfeasible, maybe due to unavailability of source code or behavior that cannot be easily understood from a simple observation from outside.

Since HDFS is developed as opensource software, source code is available and detailed documentation has been released, showing internal mechanisms of the software. For this motivations, developing a model should be simpler. In next sections two different models existing in literature are presented.

3.3.1 Practical performance analysis model

In [18], Wu et al., present a model for a generic distributed file system. This model has been derived from HDFS and uses the same nomenclature as HDFS for components of the system. Like the title says, the objective of this model is providing a practical way for performance analysis for any distributed filesystem.

The followed approach is based on the UML (Unified Modelling Language) representation of HDFS. From this representation, the authors have extracted the components of the system and then, for each component, its internal structure. Model is composed by this elements:

- DataNode
- NameNode
- Client
- File
- Block

The first three components are representing the components of software that run the DFS. File and Block, instead, are concepts that are modelled for being consistent with the behavioral model, which describes the two main operations of a DFS: read and write. Both read and write operations are described by the internal action composing the complete operation. Via UML Sequence diagrams, authors model the exchange of messages between client and servers.

Once defined the required message exchange, authors defined a set of "configurable parameters" that are values that are possibly tuned with an impact on performance. With this performance parameters, as set of performance variables has been built. In the paper the variables are the time required for a read or write action. Each variable is composed by other more specific variables defining, in example, the time required for obtaining a block location or the time required for writing a block on the disk of datanode. All this specific variables are function of the configurable parameters involved in the operation.

3.3.2 Modelling HDFS using CPN

This second model, presented in [19], is more expressive than the one presented in the previous section and uses a different approach. The model uses Coloured Petri Nets and CPN-tools for the model analysis [20].

Coloured Petri Nets (CPN) is a formal method evolved from Petri Nets, a method widely used in performance modelling and evaluation. CPN improves standard PN permitting the analysis of complex and large systems with several functionalities: allows hierarchical model construction, differentiation of the tokens using a color and permitting to carry different informations and addition to timing on model. These new functionalities and tools that can be utilized for simulation and analysis of CPN give the researchers the possibility to evaluate both performances and checking logical and functional correctness of a complex system. In fact, one of the advantages of this modelling formalism is the ability to evaluate both performance and logical correctness with the same (or at least very similar) model. For performance analysis CPN can help generating some useful quantitative informations, like queue lengths, throughput and response time.

The model uses the ability of CPN to be constructed by composition and has developed System Model that is composed by the three components of the HDFS architecture: *ClientNode*, *DataNode* and *NameNode*. This components interacts through a "colour set" defining the type of messages exchanged between the components of the system through the network. Each component is defined with a submodel, representing the internal structure of the node, with its internal and external communication flow and messages, defined each in a specific colour set. The complete model can be also represented in graphical form, and is shown in figure 3.1.

3.4 Considerations

These two models developed are quite generic and represent the entire HDFS architecture. In addition, the first one gives only performance measures of time required for an action without considering other measures, like queue lengths or throughput. This is why we developed a new model, focused on the analysis of the issue shown in section 3.2.2 that involves the minimal set of components needed for the problem. This model will be shown and used in the analysis in next chapters.

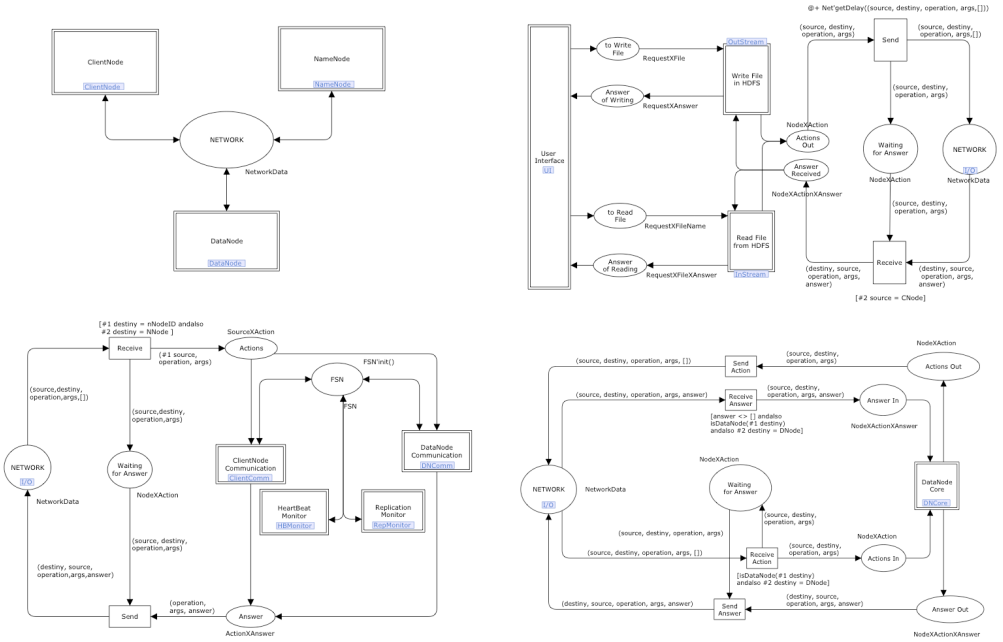

Figure 3.1: Graphical representation of HDFS modelled with CPN. Top Left: Complete System, Top Right: Client, Bottom Left: Namenode, Bottom Right: Datanode

4 Modelling Formalism

As seen in previous chapter, several approaches to representation of systems for performance evaluation exists. As said, existing models of HDFS doesn't fit with requirements of the analysis that has been set for solving the identified problem. In this work other different modelling formalism have been used for a simpler representation and more practical analysis. The core of all the analysis are M/M/1 queues, whose theory is applied for predicting average system usage.

The model described in next chapter will make use of PEPA$_k$, an extension of PEPA language. In following sections these languages are explained.

4.1 PEPA

Performance Evaluation Process Algebra [21] is a modelling language developed by Jane Hillston. This language is inspired by existing process algebras, like CCS, with the objective of integrating performance analysis in system design. *PEPA* takes advantage of the features of process algebras and incorporates the performance modelling features for specifying a stochastic process. The main advantage from process algebras is the compositionality. Compositionality allows to specify model in term of smaller components, like in the model that is being shown in next chapter, and combine this components for building more complex systems.

4.1.1 Language outline

As said before, in PEPA a system is described as cooperating components. These components represents specific parts of a system and represent the active units. Components can be atomic or composed itself by other components.

Components has a behavior defined by the activities in which it can engage. An *activity* is composed by an *action type* and an *activity rate*. With *action type* we represent the possible actions that the component being modelled can do. With *activity rate*, instead, we represent the duration of an activity, which is an exponentially distributed random variable that it is represented by a single real number parameter. An *activity* is so represented by the couple (α, r) where α is the action type and r is the activity rate.

When an activity, i.e. a $= (\alpha, r)$, is enabled this will wait for a certain time determined by the rate r and then the action α is executed. If several activities are enabled, each one will wait for the time determined by its own rate. An *activity* may be preempted, or aborted, if another action has been completed first.

4.1.2 Syntax and semantics

The syntax of PEPA is defined as follows:

$$P ::= (\alpha, r).P \mid P + Q \mid P \underset{L}{\bowtie} Q \mid P/L \mid A$$

More detail are explained below:

Prefix $(\alpha, r).P$ the component $(\alpha, r).P$ first enables and activity with type α and then, at the end of the waiting time, behaves as component P;

Choice $P + Q$ the component may behave either as P or as Q. The choice enables all the current activities of both P and Q. The first activity that completes determines which process the component will behave as;

Cooperation $P \underset{L}{\bowtie} Q$ P and Q are synchronized on the set of action types specified in L. The actions specified in L can be enabled only if both P and Q are ready to enable them. If an activity has an unspecified rate $\top$, the component is *passive* with respect to that action type. This means that the component doesn't contribute to the work for that action (i.e. a message passing channel);

Hiding P/L the component behaves as P but the actions specified in L are not visible to external viewers;

Constant $A \stackrel{def}{=} P$ defines a constant A that behaves as process P.

When multiple activities are enabled at the same time, a *race condition* determines the behavior of the model. All the activities are proceeding, but only the "fastest" is succeeding. Due to the nature of random variables determining the duration of the activities, the "fastest" cannot be determined in advance for every execution. This race condition leads to a probabilistic branching, so we can state that a certain behavior is more or less likely than another. If all the enabled activities share the same rate, they have equal probability to be the "fastest", but if rates are different, is more likely that the "fastest" is the activity with the higher rate. Any other activity will be preempted. This means that the activity will be aborted, or more precisely, interrupted.

We said before that an activity can have an unspecified rate. This activity must be shared with another component of the system where this activity has a specified rate, which determines the rate of this shared activity. In case of multiple passive activities ready to be enabled, these passive actions must have an assigned weight. This weight is a natural number that defines the probability of that passive activity of being enabled. The higher is the weight, the more likely the passive action can be enabled. By default weight is 1.

The operational semantics of PEPA is shown in figure 4.1.

4.1.3 Underlying Markov Process

With the derivation graph of a PEPA model is possible to generate the representation of the modelled system as a stochastic process. Since the rate of activities is assumed to be exponentially distributed, the resulting stochastic process is a continuous time Markov process. With this CTMC is possible to solve the model and find steady state distribution. Steady state distribution permits to derive the probability of the model operating as a certain component of the PEPA model, when stability has been reached. This means that we can obtain the probability that, observing the system at random after it has been running some time, it is behaving in a certain way. From this probabilities is possible to obtain measures like throughput, average delay time and queue lengths.

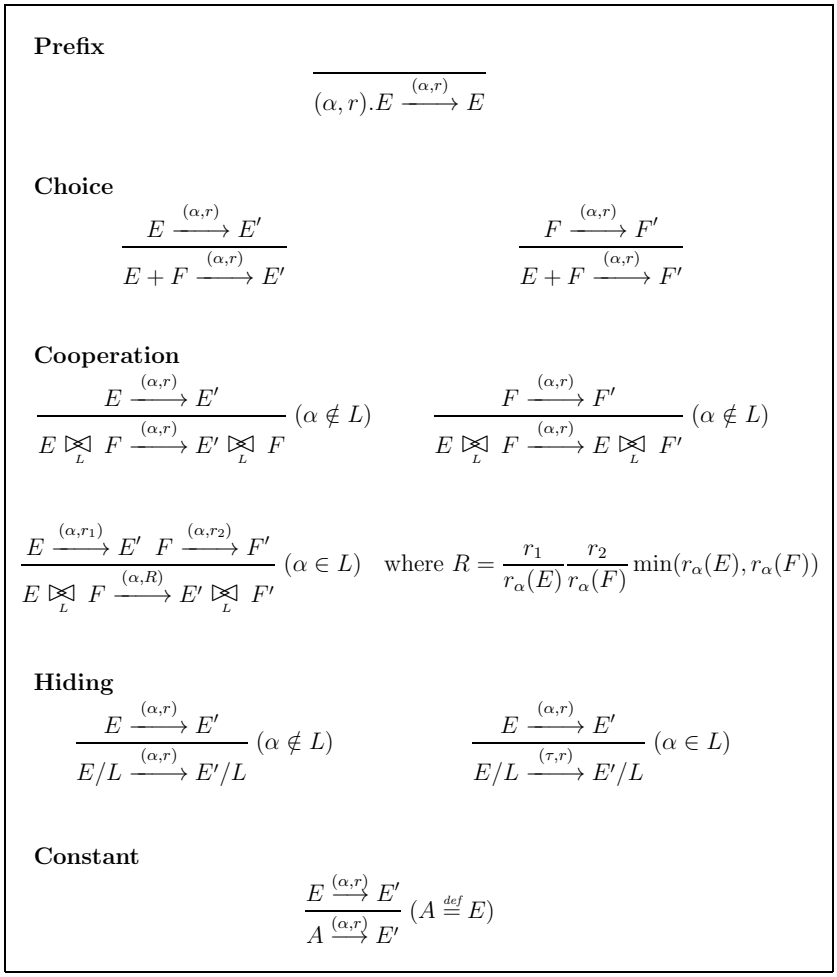

Figure 4.1: PEPA operational semantics

4.2 PEPA$_k$

PEPA$_k$ [22] is an extension of PEPA that makes use of process parameters, implemented in the Möbius modelling framework.

Möbius Framework and Tool

The Möbius Framework provides an abstraction of various modelling formalisms, permitting a multi-formalism model definition. Components of the system can be defined with different modelling languages and composed in a more complex model. This is possible to an AFI (Abstract Functional Inter-

face), a common interface between all the modelling formalisms supported by the framework.

The Möbius software, was developed by Professor William H. Sanders and the Performability Engineering Research Group (PERFORM) at the University of Illinois at Urbana-Champaign. The tool permits a hierarchical composition of systems. The modeler starts specifying *atomic* models, in any formalism, which are the base of the system. Then these models can be connected in a *composed model*, which is a set of atomic, or in turn composed, models. Once defined the complete model, performance variables has to be specified. These performance variables are the performance values that we're interested in. Is possible, in addition, to use some global variables to create some experiments to work on, i.e. evaluate how the performance variable changes when an action rate is increased.

Model can be then solved using both analytical solvers, for steady state and transient measures, and discrete event simulator.

4.2.1 PEPA$_k$ syntax

The syntax of PEPA$_k$ is the following:

$$S ::= (\alpha, r).S \mid (\alpha!e, r).S \mid (\alpha?x, r).S \mid S + S \mid \text{if } b \text{ then } S \mid A_S \mid A_S[\underline{e}]$$
$$P ::= P \underset{L}{\bowtie} P \mid P/L \mid A \mid A[\underline{e}] \mid S$$

where A ranges over the set C of process constants, A_S over a set of $C_S \subset C$ of sequential process constants, x over a set of X process parameters, e is the syntax of an arithmetic expression over X and b is a boolean expression over X.

The additions of PEPA$_k$ over PEPA are the following:

Formal parameters now processes can be instantiated with parameters

Guards operations can be guarded and they can be present in the current status of the process only if the guard is satisfied

Value Passing values can be communicated between components via activities

Using the operational semantics of PEPA$_k$, shown in figure 4.2, is possible to show that any PEPA$_k$ model lead to a PEPA model with an equivalent performance behavior.

$$
\begin{aligned}
[\![P[\underline{x}] \stackrel{\text{def}}{=} Q]\!]_E &= \{P_{\underline{i}} \stackrel{\text{def}}{=} [\![Q]\!]_{E'} : i_1, \ldots, i_n \in T, E' = E[i_1/x_1, \ldots, i_n/x_n]\} \\
[\![P \underset{L}{\bowtie} Q]\!]_E &= [\![P]\!]_E \underset{\mathsf{refine}(L)}{\bowtie} [\![Q]\!]_E \\
[\![P/L]\!]_E &= [\![P]\!]_E/\mathsf{refine}(L) \\
[\![P + Q]\!]_E &= \begin{cases} [\![P]\!]_E & \textit{if } [\![Q]\!]_E = 0 \\ [\![Q]\!]_E & \textit{if } [\![P]\!]_E = 0 \\ [\![P]\!]_E + [\![Q]\!]_E & \textit{otherwise} \end{cases} \\
[\![\mathsf{if}\ b\ \mathsf{then}\ P]\!]_E &= \begin{cases} [\![P]\!]_E & \textit{if } \mathsf{eval}(b, E) = \mathsf{true} \\ 0 & \textit{otherwise} \end{cases} \\
[\![(\alpha !\mathsf{e}, r).Q]\!]_E &= (\alpha_{\mathsf{eval}(\mathsf{e},E)}, \mathsf{eval}(r, E)).[\![Q]\!]_E \\
[\![(\alpha ?\mathtt{x}, r).Q]\!]_E &= \sum_{\{\alpha_i : i \in T\}} (\alpha_{\mathtt{i}}, \mathsf{eval}(r, E)).[\![Q]\!]_{E[i/x]} \\
[\![(\alpha, r).Q]\!]_E &= (\alpha, \mathsf{eval}(r, E)).[\![Q]\!]_E \\
[\![A[\underline{e}]]\!]_E &= A_{\mathsf{eval}(e_1,E),\ldots,\mathsf{eval}(e_n,E)}
\end{aligned}
$$

Figure 4.2: PEPA$_k$ operational semantics

As example, in the paper, a simple PEPA$_k$ model of a M/M/s/n queue is presented:

$$
\begin{aligned}
Queue[m, s, n] \stackrel{def}{=} &\text{ if } (m < n) \text{ then } (in, \lambda).Queue[m + 1, s, n] \\
&+ \text{ if } (m > 0) \text{ then } (out, \mu * min(s, m)).Queue[m - 1, s, n]
\end{aligned}
$$

Then the translation in plain PEPA is shown, demonstrating the possibility to transform from PEPA$_k$ to PEPA keeping the same behavior.

$$
\begin{aligned}
&Queue_{0,s,n} \stackrel{def}{=} (in, \lambda).Queue_{1,s,n} \\
&Queue_{i,s,n} \stackrel{def}{=} (in, \lambda).Queue_{i+1,s,n} + (out, \mu * i).Queue_{i-1,s,n} \text{ for } 0 < i < s \\
&Queue_{i,s,n} \stackrel{def}{=} (in, \lambda).Queue_{i+1,s,n} + (out, \mu * s).Queue_{i-1,s,n} \text{ for } s \leq i < n \\
&Queue_{n,s,n} \stackrel{def}{=} (out, \mu * m).Queue_{n-1,s,n}
\end{aligned}
$$

4.3 Performance Measures for M/M/1 queues

When analyzing a model, first thing to determine is what measures to extract. This measure has to provide a relevant value that has to be useful for

comparisons between different systems.

An M/M/1 queue [23, 24] can be represented as a *birth-death* process, a *continuous time markov chain* with very special structure, since transitions are possible only between nearest neighbors (moving from i to i+1 or i-1, with the exception of 0 that moves only to 1).

Figure 4.3: graphical representation of M/M/1 Queue

Using steady state analysis, is possible to compute some performance measures concerning the M/M/1 queue, including:

- Mean number of customers in the system
- Mean queue length
- Average response time
- Average waiting time

These measures are strongly related to the rates of arrival time (λ) and the service time (μ). In the following analysis a single measure is evaluated. The measure chosen is the mean number of customers in the system.

Within a queuing system a client can be present in the queue, waiting to be served, or may be receiving service, so obtaining a measure of the number of customers in the system can give a good measure about the load of the system. This measure is preferred over the other measures due to Little's Law [25, 23]:

$$L = \lambda W$$

where W is the response time of the system and L is the mean number of customers. Using this law is possible to focus only on the mean number of customers in the system and evaluate then the response time. Is easy to see that the response time is directly proportional to the mean number of customers, so having a low L gives a low average response time.

4.3.1 Mean number of customers in a queue

Utilization

Before defining the mean number of customers in a queue, *utilization factor* has to be defined. The *utilization factor* is the measure of average use of a service facility. It's represented by the variable ρ and is defined as follows:

$$\rho = \frac{\lambda}{\mu}$$

where λ is the admission rate and μ is the service rate. The existence of a steady state solution requires that the value of ρ is less than 1, or, more practically, $\lambda < \mu$. Note that in case of $\lambda > \mu$, the system size will keep increasing without limit. In this case is considered unstable. It is not properly true that the given formula for ρ always defines the utilization, because sometimes customers can be refused admission ("lost" or "rejected" customers, i.e. due to queue full in case of limited capacity), resulting in an admission rate less than λ and, consequently, an utilization lesser than ρ.

In the following analysis is assumed that any request to join the queue is accepted.

Mean number in system

Given that the system is stable ($\rho < 1$), the average number of customers in a M/M/1 queue is defined as:

$$L = \frac{\rho}{1 - \rho}$$

In this analysis, the value L is referred as $E[DN_{rt}]$, indicating that is the average number of customers in the queue system of *datanode* DN_{rt}.

5 HDFS System Model

??

In this thesis the working scenario is quite uncommon. HDFS typical usage is coupled with at least *MapReduce* component of *Hadoop* stack. In this way processes benefit from the *Hadoop* approach of moving tasks next to the data. Instead, here the working scenario involves only the filesystem component, that will be used by clients residing out of the the cluster to store data.

Considering the issue presented in 3.2.2 and the models shown previously, a specific model has been developed. Differently from the model using CPN presented in section 3.3.2, that represents the entire system with all the required interactions between components, the model that is being shown and used from this point forward, focuses only on the interaction between the client and the datanode.

5.1 Working Scenario

A sample of analyzed scenario is represented in figure 5.1.

More generally, we suppose to have one rack containing only clients and other racks containing nodes running HDFS. Each rack can contain a limited number of servers, usually up to 16 if we consider as rack a single blade chassis. Also number of racks may vary, depending on the file system disk space needed.

Due to this structure, the topological distance between clients and server is uniform. This makes the policies explained in 2.2.3 to use random selections,

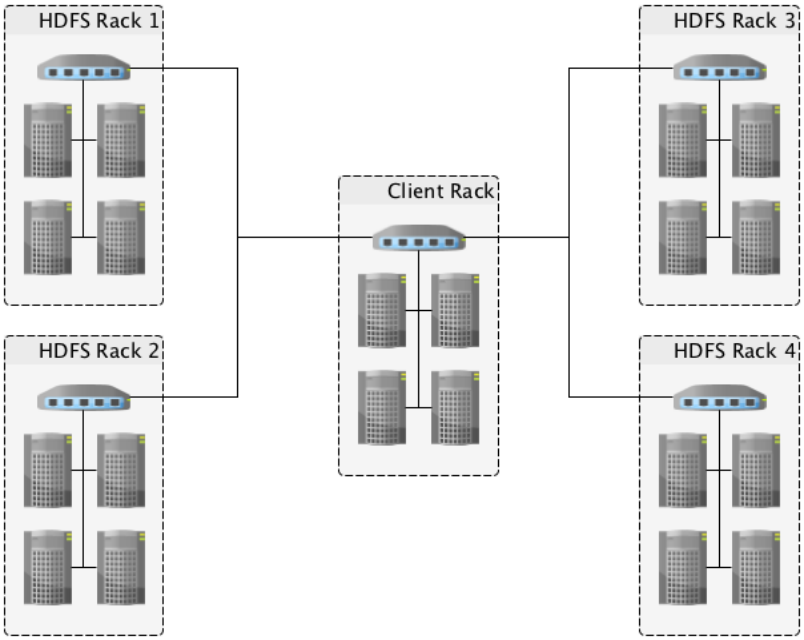

Figure 5.1: Example of possible scenario

leading to a particular situation.

5.2 Model Components

The model represents a set of clients and the *datanodes* in the system, each one represented by a queue. The models relies on the interaction between clients and queue: each client can enqueue its request on a certain queue and the request is then processed by *datanode* owning that queue.

5.2.1 Client

In this model a client doesn't really represents a real client that needs a certain set of resources for computing, but is a representation of the requests for a certain resource. So it is possible to group all the requests for a certain resources on a single client making requests with a certain rate λ_b. This rate is the arrival rate that can be derived by observing the requests in the system from various nodes.

Modelled client fires requests and doesn't wait for the reply from the server. This behavior makes impossible to measure the throughput and other performance measures of a client, but this is not required in this work's analysis.

5.2.2 *Datanode*

Datanode is represented as a server with a queue. Each one can receive in its queue requests for blocks that it is hosting. If it is empty (not hosting any block), datanode doesn't get any request in its queue.

Elements are removed from the queue with a serve action at constant rate, μ. The rate has been set as constant since blocks are supposed of fixed size and service time from internal storage is considered equal for each block. This service rate is the same for all *datanodes* in the cluster, because are considered to be of homogeneous type.

5.2.3 Complete System

The complete system relies on the interaction between the "client" components and "*datanode*" components. The interaction is made through action synchronization. Each "client" is synchronized with each "*datanode*" on the request for a certain block.

5.2.4 Why no *namenode*?

Namenode has been excluded from the model because some considerations on its structure has been done. As said by developers, even if is a known bottleneck, modifying the operating structure of this component of HDFS cluster requires a deep architectural review that is not easy to do. Also, for the emerged issues that are being faced in next chapters, the role of *namenode* is quite marginal and can be eventually codified by modifications in the interactions between *datanode* and client.

5.3 PEPA$_k$ HDFS System model

Using the languages explained in previous chapter and according to the details shown in 5.2, a model of multiple clients interacting with *datanodes* has been developed using PEPA$_k$.

5.3.1 *Datanode* queue

Instead of representing the *datanode* only as a consumer of requests, the following PEPA$_k$ process represents it as a queue with a server. In fact, the focus of following analysis is on the average number of customers in the system, which can be considered as a measure of how much the server's resources are requested.

$$DN_{rt}[m] \stackrel{def}{=} \text{if } (m < n) \text{ then } \sum_{i:b_i \in DN_{rt}} (read_i, \top).DN_{rt}[m+1]$$
$$+ \text{if } (m > 0) \text{ then } (serve, \mu).DN_{rt}[m-1]$$

The various $read_i$ operations are passive, since their rate depends from the rate of the clients which are sending the requests. The presented process, with the support of guards introduced by $PEPA_k$, represents an M/M/1 queue with finite size n which permits enqueuing when there are empty places and proceeds with the serve action only if there are elements to be served. The queue is represented with finite capacity only to have a finite number of states in the system.

5.3.2 Client

The client, instead is a very simple process. As said in 5.2, the client process doesn't really represents a real client, but represents the requests that are sent for a certain resource. Simplifying, we can say that every client of the system can request only a certain resource. Like said previous, for this modelling choice, the client doesn't wait for the response from the *datanode*.

$$Cl_i \stackrel{def}{=} (read_i, \lambda_i).Cl_i$$

In the client process, i represents id of the block being requested and λ_i is the rate which this requests arrive in the system. This last parameter is related to the popularity of the block. The client carries out the $(read_i, \lambda_i)$, having a duration that is exponentially distributed with parameter λ_i. This means that the client will require some time Δt, drawn from the distribution (which is inversely proportional to λ_i), to do the $read_i$ operation. So, the higher is the request rate, the lower is the time between successive $read_i$ requests in the system.

5.3.3 System

The system is composed by multiple cooperations. Clients are grouped on a process Cls, where they proceed concurrently without interaction, represented using the cooperation on an empty set of actions ($\|$ operator). Same is done with all the *datanodes*. The two new processes are then combined with a cooperation on the set of actions containing all the $read_i$ operations that are present in the model.

$$Cls \stackrel{def}{=} (Cl_1 \| ... \| Cl_j)$$
$$DNs \stackrel{def}{=} (DN_{11} \| ... \| DN_{rt})$$
$$System \stackrel{def}{=} Cls \underset{\{read_i\}}{\bowtie} DNs$$

6 Placement probability

Due to the particular scenario being analyzed in this work, the replica placement policy used in write process makes some particular choices, different from the default one made in common *Hadoop* utilization scenario. In next section, the probability of choosing a node during a write operation is shown. This probability formula further shows the particularity of the working scenario that has been took as reference.

In table 6.1 is shown the notation used in the probability and for the entire analysis in this work.

$R_1, \ldots, R_k$	racks available in the system
k	number of racks in the system
DN_{rt}	with $1 \leq r \leq k$ and $1 \leq t \leq D_r$; datanode t on rack R_r
D_r	number of datanodes on rack R_r
C_b	number of replicas of a block
$N_{C_b} = \lceil \frac{C_b+1}{2} \rceil$	number of racks needed for storing C replicas
$X_1, \ldots, X_{N_{C_b}}$	r.v. racks selected for placing replicas in order of selection
$b_1, \ldots, b_n$	blocks present in the system
$b \in DN_{rt}$	block b has a replica stored on node DN_{rt}
λ_i	request rate for block b_i
μ	service rate of datanodes
ρ_{rt}	utilization of DN_{rt}. $\rho_{rt} < 1$
$E[DN_{rt}]$	average number of clients in node DN_{rt} (queued or served)
$E[\sigma]$	average number of clients for the entire system

Table 6.1: Notation used in the analysis

6.1 Replica placement probability

The following formula represents the probability that a node has been selected for storing a replica of a given block b.

The random variable F_{rt} is defined as follows:

$$F_{rt} = \begin{cases} 1 \text{ if node } t \text{ on rack } R_r \text{ has the requested block} \\ 0 \text{ otherwise} \end{cases}$$

and shows if a block is contained or not in the *datanode* DN_{rt}.

In simple words we can explain the probability of containing a given block in this way:

$Pr\{F_{rt} = 1\}$ = probability of R_r of being selected as first rack and node D_{rt} selected in that rack for storing the block + probability of R_r being selected as second rack and node D_{rt} selected in that rack for storing the block + ...+ probability of R_r being selected as last rack and node D_{rt} selected in that rack for storing the node.

The probability can be so written in this way:

$$\begin{aligned} &Pr\{F_{rt} = 1\} = Pr\{F_{rt} = 1 | X_1 = r, X_n \neq r \; \forall 1 < n \leq N_{C_b}\} \\ &\times Pr\{X_1 = r, X_n \neq r \; \forall 1 < n \leq N_{C_b}\} \\ &+ \sum_{n=2}^{N_{C_b}} Pr\{F_{rt} = 1 | X_n = r, X_{n'} \neq r \; \forall 1 \leq n' \leq N_{C_b} \wedge n \neq n'\} \\ &\times Pr\{X_n = r, X_{n'} \neq r \; \forall 1 < n \leq N_{C_b} \wedge n \neq n'\} \end{aligned}$$

In detail the single components of the formula:

$$Pr\{F_{rt} = 1 | X_1 = r, X_n \neq r \; \forall 1 < n \leq N_{C_b}\} = \frac{1}{D_r}$$

$$Pr\{F_{rt} = 1 | X_n = r, X_{n'} \neq r \; \forall 1 \leq n' \leq N_{C_b} \wedge n \neq n'\} = \frac{2}{D_r}$$

The last rack selected has a different formula:

$$\begin{aligned} &Pr\{F_{rt} = 1 | X_{N_{C_b}} = r, X_n \neq r \; \forall 1 \leq n < N_{C_b}\} = \\ &= \begin{cases} \frac{2}{D_r} \text{ if } N_{C_b} = \frac{C+1}{2} \text{(odd number of replicas)} \\ \frac{1}{D_r} \text{ otherwise (even number of replicas)} \end{cases} \end{aligned}$$

This due to the the placement policy. I.e. 4 replicas will require $\lceil \frac{4+1}{2} \rceil = 3$ racks and replicas will be placed in this way:

- X_1 one replica,
- X_2 2 replicas,
- X_3 one replica.

Since the in the working scenario all the racks are at the same topological distance, is possible to show that the probability of selecting a certain rack is equal, however is the replica that is being placed on:

$$Pr\{X_1 = r, X_n \neq r \; \forall 1 < n \leq N_{C_b}\} =$$
$$Pr\{F_{rt} = 1 | X_n = r, X_{n'} \neq r \; \forall 1 \leq n' \leq N_{C_b} \wedge n \neq n'\} = \frac{1}{k}$$

With this probability we can see that the hierarchy of the topology has impact on the distribution of the resources among the *datanodes*. Instead, with this configuration scenario, there's no difference between selecting certain nodes from a rack or from another.

7 Model Analysis

General performance measures shown in 4.3 cannot be used directly on the model shown in chapter **??**. This because system's queues could be potentially being used by multiple clients. This requires a computation of the arrival rate λ that is dependent from all the clients that can contact that single queue.

Additionally, using the mean number of clients of each queue as global measure is not a good idea and quite messy. So a single measure for all the system has to be computed for a better evaluation and comparison.

7.1 Mean number of clients requesting from *datanode*

Using the previously shown formulas, is possible to define the formula for $E[DN_{rt}]$. The formula for the mean number of customers in the system is quite simple and depends only on the definition of ρ.

Differently from a common queue that has only one arrival rate, the PEPA$_k$ model has a different λ rate for each block requested. Since every *datanode* can contain more than one block, there are different arrival rate in the queue.

So the utilization formula ρ has to be revised, in order to include all this rates. To introduce changes, the working scenario and the replica selection policy (shown in 2.2.3) have to be taken in account.

Due to the structure of the working scenario, the replica selection policy falls back to a particular behavior, where the replica is randomly selected. So the probability of a replica for being addressed by the client is the same for

every block. This gives an useful hint for the modification of the utilization formula. Since the probability of a replica of being selected is uniform, is possible to weight each entry rate with respect to the probability of that queue of being selected. There are then multiple utilization formulas, one for each queue, since each one may contain different blocks having a different incoming rate. The general utilization formula is the following:

$$\rho_{rt} = \sum_{i:b_i \in DN_{rt}} \frac{\frac{\lambda_i}{C_i}}{\mu}$$

The mean number in system formula remains more or less the same, but is different for each *datanode* process:

$$E[DN_{rt}] = \frac{\rho_{rt}}{1 - \rho_{rt}}$$

7.2 Mean number of clients being served by the entire cluster

The mean number of client being served by the entire cluster could be calculated, in a basic system of multiple queues, as the mean value of each mean number of clients being server by each queue. In this case, instead, the question is more complex. First of all we had to consider that, since every request for a block is not served by a single *datanode*, the average number of clients that can be found in queue requesting a certain block, depends from multiple *datanodes*. Second thing is that every block could have different request rate from the others and so, its average queuing hasn't to be considered as the same way as the other blocks having lower (or higher) rates.

So two types of weight has to be inserted in the formula:

- the weight of each datanode with respect to the number of replicas of the block;
- the weight of each averaged queue length for a block with respect to the total number of requests for that block are made.

The first weight is very easy to implement and can be written in this way:

$$E[b_i] = \sum_{DN_{rt}:b_i \in DN_{rt}} \frac{1}{C_i} E[DN_{rt}]$$

The result of this formula returns the average queuing found for a certain block b_i.

The second weight, instead, is a little bit more complex. For identifying the contribution of each averaged queue of a block, is necessary to establish a measure for each block. Since each block request is identified by the rate, is possible weight of each $E[b_i]$ in this way:

$$E[\sigma] = \sum_i \frac{\lambda_i}{\sum_x \lambda_x} E[b_i] \tag{7.1}$$

The resulting formula for computing $E[\sigma]$ is the following:

$$E[\sigma] = \sum_i \frac{\lambda_i}{\sum_x \lambda_x} \sum_{DN_{rt}:b_i \in DN_{rt}} \frac{1}{C_i} \frac{\rho_{rt}}{1-\rho_{rt}} \tag{7.2}$$

This formula will be used from here and out as performance measure for the entire system. With this formula will be then possible to compare different system configurations. Due to its direct relation with the average waiting time, we should expect that a system σ' behaving better than another system σ will have $E[\sigma'] < E[\sigma]$.

8 Proposed solution

In 3.2.2, the problem of resource increasing their popularity has been presented. A possible solution for this issue has been introduced. The proposal is to increase the number of replicas of the block, permitting better equilibration between *datanodes*.

Looking at *balancer* behavior and at communication messages between *datanodes* and *namenode*, is possible to see that improvement margins exist.

The resulting balancing algorithm implements a *model driven dynamic optimization*, which solves a system model to determine which actions to take for improving the performances.

8.1 Improvements to the system

To improve the performances, like said previously, a new balancing mechanism has to be implemented. At the moment the system is lacking of a performance focused balancing mechanism, since the only modifications to positions of blocks are done in case of *datanodes* fault or due to rebalance of *datanodes* disk usage.

In actual HDFS implementation, heartbeats carries, among the other informations, also the number of data transfers currently in progress, with load balancing objective. In the upcoming solution, not only the number of data transfers in progress are required, but also the number of clients waiting to be served is required. Combining the two values, the number of clients requesting a block from a *datanode* is obtained.

Another measure required is the request rate for each block. This measure

can be collected directly from *namenode*, since each request for a block is first addressed to it. These request rate are the single λ_b values of the system in a certain moment.

Once obtained this measures and extracted from the namespace the number of replicas of a certain block, is possible to use the formula 7.1 to compute the average number of customers being served by the entire cluster.

In the presented issue, a resource becomes popular and very requested, providing so an increase of its λ_b value. In addition, the growth of requests will lead to an increase of the number of customers waiting and being server by the single datanodes, growing consequently the value of $E[\sigma]$. As said in precedence, due to Little's Law, this is an increase of the response time of the system. In order to keep the overall system response time more or less constant, a balancing algorithm has to be executed for identifying possible interventions to be undertaken.

8.2 Balancing algorithm

8.2.1 Idea

The idea behind this new balancing algorithm is to spread the increased number of requests to an higher number of replicas. Is necessary so to compute a set of actions to be performed in order to move the new system's measures close, or better under, a certain threshold. Is necessary so to determine an objective threshold.

Threshold

One of the objectives of the balancing is to intervene to reduce the impact of the increasing requests. To avoid to keep track of historical measures of the status of the system, the target threshold has to be computed with the data available at the moment of intervention.

The objective can be so determined using the formula 7.2, with a particular set of inputs. Since we have a λ_i value that has increase since b_i has become mode popular, this value is replaced in the list of all the rates by the average value of all the remaining rate of the system. In this way, in computation of $E[\sigma]$ the contribution of the block b_i is more or less the same of the other blocks.

Constraints

Obviously, replicating every resource everywhere can for sure provide a good improvement of the system, but this is certainly not a good solution in term of efficient resource usage. In addition, some architectural constraints exist, blocking this type of intervention:

- No mode than two replicas of the same block per rack are allowed
- No more than one replica per rack of the same block
- No less than two racks should contain a replica of the block

This type of constraints limit the possible interventions of the balancer, that has to take in account this architectural rules.

In addition, while developing the balancer, another constraint has been added. The balancer has to use the less possible space to improve the resource usage. In the implementation, in fact, when a maximum number of replicas is reached for a certain block, further steps will require to decrease the replicas of this block and the increase of replicas of another block. In this case, less disk space in the cluster is used.

8.2.2 Algorithm

The algorithm is presented as pseudocode in box algorithm 1. Is a brief representation and doesn't cover the exceptional case that, when tried all the replications, the $E[\sigma]$ value is still greater than the threshold set.

When adding a new replica, the algorithm places it on the less loaded node of the cluster respecting the placement constraints. The algorithm prefers empty nodes, if existing.

When removing a replica, instead, starts removing it from a node hosted on the rack having the higher number of blocks.

Is possible to see that the algorithm moves with little steps to find out a good optimization. The algorithm is a *local search* and this is why the optimization found could not be the best possible optimization. *Local search optimizations* are known to suffer of *local minimum*, but for the objective of this balancing algorithm this is not a real problem.

8.2.3 Experimental implementation

For testing purposes the algorithm has been implemented in a program using *python* language.

The program, given the cluster components (number of racks, number of nodes per rack and default number of replicas per blocks), generates a possible configuration of the system with a certain number of blocks. The configuration is done by applying the default replica placement policy of HDFS.

Then randomly selects one of the blocks, and changes its rate, simulating the increase of popularity of that block. At this point applies the balancing algorithm. At the end compares the old system configuration with the new generated applying the balancer's improvements.

The comparison is done by ranging by steps the λ rate of the popular block from a certain λ_{min} to λ_{max} and evaluating the $E[\sigma]$, of pre- and post-balancing system, at each step. The results are then shown on a plot where is possible to compare the two behaviors.

The program deals also with cases where, even if all the possible actions are applied, the resulting system is still above threshold. For this case the program saves the configuration giving the best optimization and returns as result if $E[\sigma]$ doesn't decrease below threshold.

For speeding up computation, multiprocessing is used in the loop over all the blocks for identifying the best way to proceed. Additionally, in order to allow reproducibility of executions, the operations involving random choices (replica placement policy, popular block selection, random increase of block popularity) uses a random number generator biased by a seed. With this seed, multiple runs of the script with the same parameters will provide the same result.

```
identify popular resource b_i;
compute E[σ];
compute target threshold using λ_i = avg(λ_x : x ≠ i);
for every block do
    copy actual datacenter status;
    increment by one the replica of the block;
    evaluate new E[σ'];
end
best_block, new_dc = block id with the lowest E[σ'], datacenter
status after increase of one replica for best_block;
backward = False;
while E[σ'] > threshold do
    if number of replicas for best block >= max number of replicas
    then
        old_best_block = best_block;
        for every block do
            copy actual datacenter status;
            increment by one the replica of the block;
            evaluate new E[σ'];
        end
        best_block, new_dc = block id with the lowest E[σ'],
        datacenter status after increase of one replica for best_block;
        backward = True;
    end
    if backward then
        remove a replica of old_best_block
    end
    add replica to best_block;
    compute new E[σ'];
end
apply new configuration to the system;
```

Algorithm 1: Pseudocode of balancing algorithm

9 Results

With the program presented in previous chapter, some experiments has been conducted. Input values for the experiments are shown in table 9.1. As number of nodes per rack has been selected the value 16, which the usual number of servers that can be hosted on a blade chassis. This is similar to a possible real implementation. By default the service rate of *datanodes* has been set to 10, supposing that fetching resources from disk and returning back to client is an operation happening very quickly.

Experiment	n_rack	n_per_rack	n_block
1	20	16	500
2	16	16	100
3	13	16	200
4	8	16	250
5	6	16	120
6	3	3	3

Table 9.1: List of experiments

Experiment's results are shown in table 9.2. The rate is set by default to 0.5 for all blocks, supposing a system with fain resource usage.

Is possible to see that, in every experiment, the balancing gives an improvement of the performance measure ($E[\sigma]$). The improvement can also be seen in plots comparing the behaviors of the system before and after the balancing. Plots are shown in figure 9.1.

Is possible to see that all the optimizations, except one, give a resulting balanced system whose performance measure, in correspondence of the rate

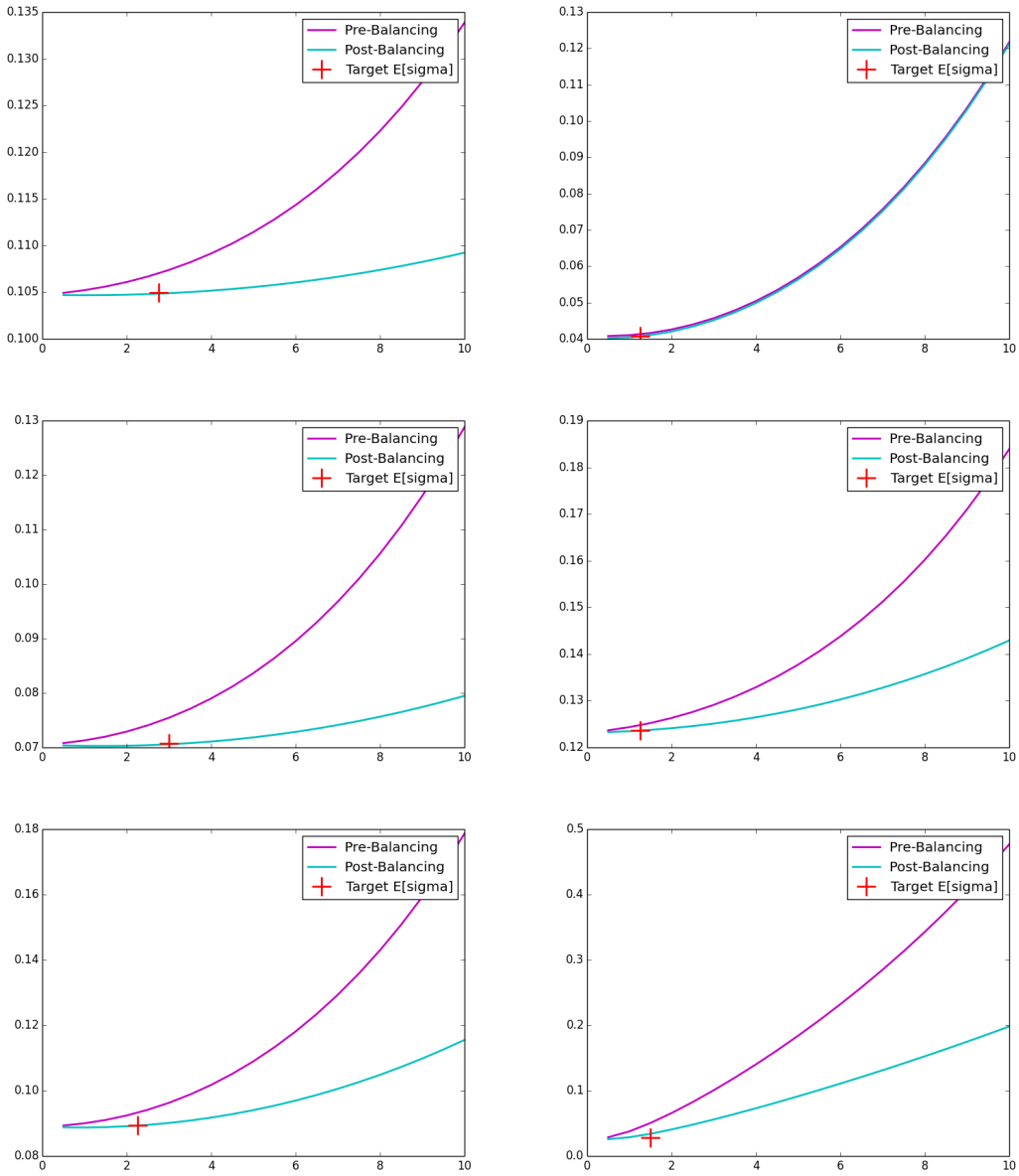

Figure 9.1: Behavior of the systems before and after the balancing

Experiment	Block	rate change	$E[\sigma]$	threshold	$E[\sigma']$
1	181	+2.25	0.107015	0.104927	0.104844
2	36	+0.75	0.041277	0.040791	0.040705
3	72	+2.5	0.075447	0.070799	0.070552
4	90	+0.75	0.124758	0.123667	0.123608
5	43	+1.75	0.093224	0.089324	0.089288
6	0	+1	0.050426	0.028638	0.033944

Table 9.2: Result of experiments

calculated when the block becomes popular, stays below the threshold set. In plots the threshold point is represented by the red +. What has been said can be seen in figure 9.2

The case of experiment 6 As can be seen from the detail 9.2, the balancing algorithm applied to the experiment 6 hasn't reached the result of changing the configuration of the system in order to stay below the threshold. This is due to the constraint inserted to the balancing algorithm. Since there are few racks, the balancer hasn't the possibility to add more than 6 replicas for that block. Once reached this number of replicas, the balanced system reaches the best $E[\sigma]$ and then any further action moves to an higher value. So the program returns this system configuration because no other operation has returned a better $E[\sigma]$.

The only possible change action that can be done is extending the system adding more racks. In this way, more replicas can be added to the system.

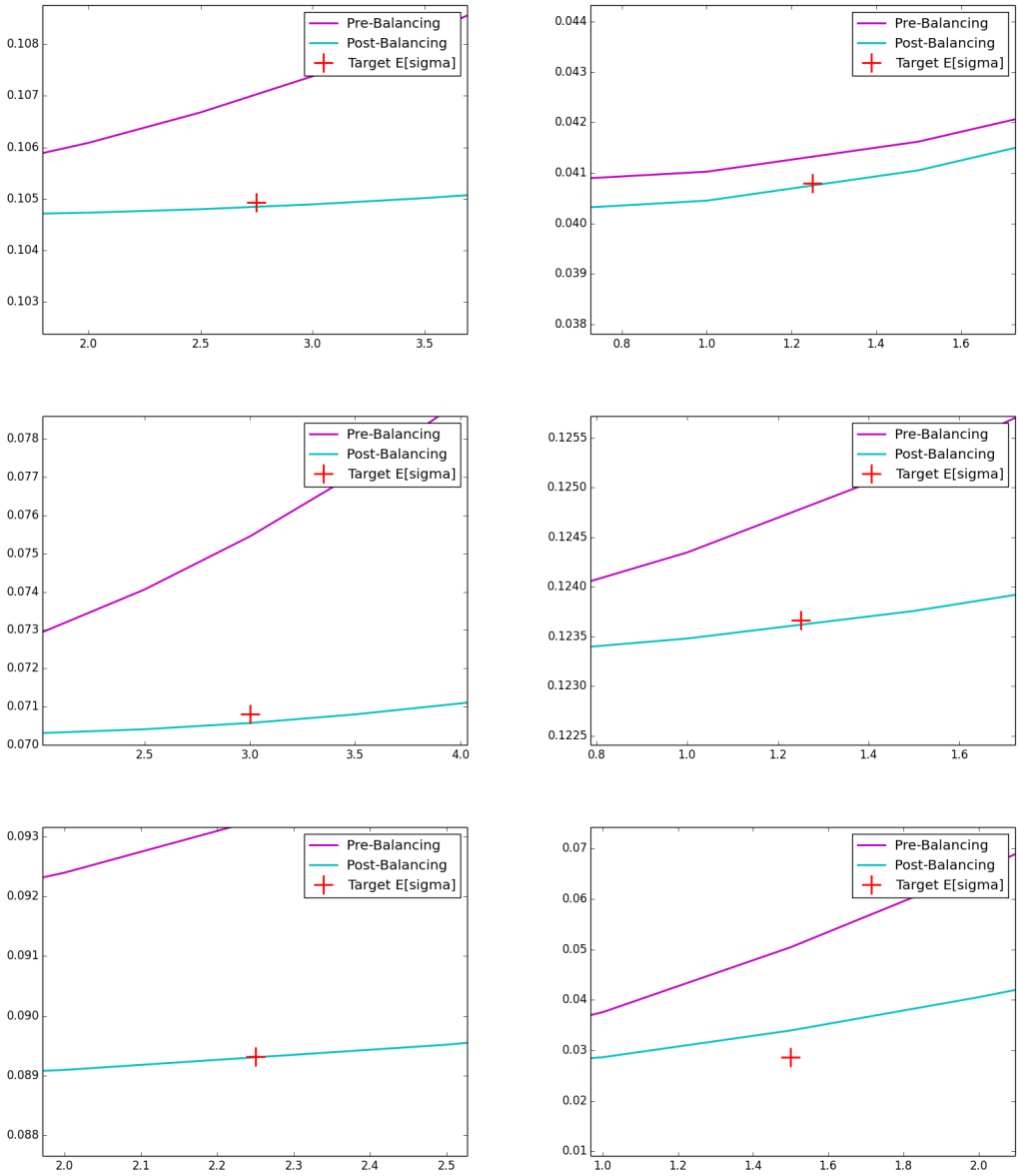

Figure 9.2: Threshold details

10 Conclusions

In this thesis work, the *Hadoop Distributed File System* has been analyzed. This filesystem, whose main objective is storing large quantity of data, has been studied under a particular scenario representing a possible setup on a real environment. Many performance limiting issues has been presented, depending mainly on the architectural design of the filesystem. Even if is mainly involved in batch operations, good system performances are still a requirement for production environments.

The problem analyzed in the thesis work is related to the quite static position of blocks inside the cluster. HDFS runs periodically a *balancer*, whose objective is to keep the disk space utilization almost uniform across all the *datanodes*. *Balancer* doesn't keep in account performance measures when modifying the status of the system. Its objective is only moving block from a server to another.

In the thesis, instead, a new job for the *balancer* is proposed. The new task the *balancer* has to deal with is the modification of number of replicas of blocks, in order to respond to an increase of resource utilization. Starting from a model of the system, focused on fewer components than the complete one existing in literature, a performance measure for the entire system has been identified. Using this measure as way for evaluating impacts of changes in the system configuration, the new balancer uses a *model driven dynamic optimization* for deciding the set of changes that has to bring in order to reach a performance objective. It begins with a study of the actual status of the system and then will intervene to try to move the system to a new configuration where the impact of increase resource popularity is mitigated. The balancing algorithm will work until a computed threshold has been reached

or no more changes to the system are possible. This hypothetical balancing algorithm has been implemented in a program for experimentation and results has been presented in last chapter. The result is that, starting from a ordinary system configuration and block placement status, the balancing algorithm intervenes on the block's replication giving a new possible distribution that gives a better performance measure with respect to the starting system. The new configuration, additionally, will respond better than the old one also to further increases of the popularity of the same block, providing a better performance measure in every situation.

The new role of the balancer seems, in this scenario, useful for spreading the requests to an higher number of *datanodes* and reducing the probability of node and resource contention. This could be also useful in an ordinary *Hadoop* cluster, where HDFS and *MapReduce* share the same cluster nodes. Applying the principle of "moving computation is cheaper than moving data", the job scheduler of *MapReduce* will have more nodes where to start the computation, possibly reducing the single node load and sharing better and more easily the computations across the cluster.

Possible future work could include the extension of the *balancer* to a more generic HDFS configuration, measuring also the improvement of job performances due to reduced node contention. This will require, however, in addition to a modelling approach like the one used in this paper, also the implementation of *balancer* task in *Hadoop*. This will permit the verification also using common HDFS benchmarks, like *TestDFSIO*[17].

Bibliography

[1] Gartner says solving 'big data' challenge involves more than just managing volumes of data. [Online]. Available: http://www.gartner.com/newsroom/id/1731916

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/1165389.945450

[4] D. Borthakur, "The hadoop distributed file system: Architecture and design," 2007. [Online]. Available: https://svn.eu.apache.org/repos/asf/hadoop/common/tags/release-0.16.3/docs/hdfs_design.pdf

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, May 2010, pp. 1–10.

[6] D. Borthakur. Hdfs block replica placement in your hands now! Monday, September 14, 2009. [Online]. Available: http://hadoopblog.blogspot.it/2009/09/hdfs-block-replica-placement-in-your.html

[7] J. Shafer, S. Rixner, and A. Cox, "The hadoop distributed filesystem: Balancing portability and performance," in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, March 2010, pp. 122–133.

[8] C. Vittal. (2013, February) Scalable object storage with apache cloudstack and apache hadoop. Slide 26. [Online]. Available: http://archive.apachecon.com/na2013/presentations/26-Tuesday/Cloud_Crowd/Chiradeep%20Vittal%20-%20Scalable%20Object%20Storage%20with%20Apache%20CloudStack%20and%20Apache%20Hadoop/S3_HDFS_apachecon.pdf

[9] D. Borthakur. (2008, June) Maximum number of files in hadoop. [Online]. Available: http://www.mail-archive.com/core-user@hadoop.apache.org/msg02835.html

[10] T. White. (2009, February) The small files problem. [Online]. Available: https://blog.cloudera.com/blog/2009/02/the-small-files-problem/

[11] L. Jiang, B. Li, and M. Song, "The optimization of hdfs based on small files," in *Broadband Network and Multimedia Technology (IC-BNMT), 2010 3rd IEEE International Conference on*, Oct 2010, pp. 912–915.

[12] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High performance rdma-based design of hdfs over infiniband," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 35:1–35:35. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389044

[13] Infiniband. [Online]. Available: http://en.wikipedia.org/wiki/InfiniBand

[14] V. Gupta, M. H. Balter, K. Sigman, and W. Whitt, "Analysis of join-the-shortest-queue routing for web server farms," *Performance Evaluation*, vol. 64, no. 9–12, pp. 1062 – 1081, 2007, performance 2007 26th International Symposium on Computer Performance, Modeling, Measurements, and Evaluation. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166531607000624

[15] H.-C. Lin and C. Raghavendra, "An analysis of the join the shortest queue (jsq) policy," in *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*, Jun 1992, pp. 362–366.

[16] N. Islam, X. Lu, M. Wasi-ur Rahman, J. Jose, and D. Panda, "A micro-benchmark suite for evaluating hdfs operations on modern clusters," in *Specifying Big Data Benchmarks*, ser. Lecture Notes in Computer Science, T. Rabl, M. Poess, C. Baru, and H.-A. Jacobsen, Eds. Springer Berlin Heidelberg, 2014, vol. 8163, pp. 129–147. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-53974-9_12

[17] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media Inc./Yahoo Press, 2012.

[18] Y. Wu, F. Ye, K. Chen, and W. Zheng, "Modeling of distributed file systems for practical performance analysis," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 1, pp. 156–166, Jan 2014.

[19] L. Aguilera-Mendoza and M. Llorente-Quesada, "Modeling and simulation of hadoop distributed file system in a cluster of workstations," in *Model and Data Engineering*, ser. Lecture Notes in Computer Science, A. Cuzzocrea and S. Maabout, Eds. Springer Berlin Heidelberg, 2013, vol. 8216, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41366-7_1

[20] L. Wells, "Performance analysis using cpn tools," in *Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools*, ser. valuetools '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1190095.1190171

[21] J. Hillston, *A compositional approach to performance modelling*, 1996.

[22] G. Clark and W. H. Sanders, "Implementing a stochastic process algebra within the möbius modeling framework," in *Process Algebra and Probabilistic Methods. Performance Modelling and Verification*. Springer, 2001, pp. 200–215.

[23] W. J. Stewart, *Probability, Markov chains, queues, and simulation: the mathematical basis of performance modeling*. Princeton University Press, 2009.

[24] L. Kleinrock, "Queueing systems. volume 1: Theory," 1975.

[25] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models.* Prentice-Hall, Inc., 1984.

)