


Fault Tolerance Model for Hadoop Distributed System


Soraya Setti Ahmed

(Mustapha Stambouli University, Mascara, Algeria)

 <https://orcid.org/0009-0003-2736-7187>, soraya.settiyahmed@univ-mascara.dz

Yahya Slimani

(ISAMM, Manouba University, Manouba, Tunisia)

 <https://orcid.org/0009-0000-0833-4556>, yahya.slimani@gmail.com

Riadh Frefita

(Esprit School, Pôle Technologique, 2083 El Ghazala, Tunisia)

riadh.frefita@esprit.tn

Abstract: Fault tolerance approaches in distributed systems are essentially based on replication and checkpointing. Each of these approaches has its advantages and limitations. This paper has two objectives: first, it proposes a fault tolerance approach based on the nodes status of a distributed system. For this purpose, it defines 3 nodes status: safety, faulty and potentially faulty. With respect of classical node status (safety, faulty), it introduces a new status that we call potentially faulty. This last node allows to enhance the availability of a distributed system. Second, it discusses the efficiency of the proposed model on two types of architectures: virtual multi-node cluster and a physical multi-node cluster with WIFI connection. Experiments have showed that proposed approach increases the system performance throughput and its fault tolerance level.

Keywords: Distributed Systems, Hadoop, Fault Tolerance, Networks, Node Failures

Categories: B.8, C.4

DOI: 10.3897/jucs.120840

1 Introduction

Fault tolerance represents the capability of any system or equipment to sustain its operation during the presence of a fault [Herault and Robert. 2015]. Systems and equipment with high fault tolerance, depending upon the adopted fault tolerance mechanism, are able to completely or partially sustain their operation upon the occurrence of a fault [Herault and Robert. 2015, Sivagami and Easwarakumar. 2019]. The development of fault-tolerant design requires careful consideration of failures that can be manifested throughout the equipment life cycle, along with their probable causes and consequences. However, the designers must also consider the cost and resource factors needed to achieve the required level of tolerance, reliability, and dependability of a system [Bansal et al. 2011]. It is often misunderstood that a fault-tolerant design should provide complete tolerance to all types of faults. This is not true. A good design should match the degree of tolerance to the criticality of the fault such that the overall optimization of cost and resource efficiencies can be achieved [Bansal et al. 2011, Herault and Robert. 2015]. One of the simple actions that can be taken to increase fault tolerance is by incorporating redundancies in the design. Redundancy simply means the presence of an alternate system (or components)

solution that can take over the intended function should the primary system fail [Koren and Krishna 2020, Kumari and Kaur. 2021].

Distributed systems and their various versions (parallel systems, cloud systems, grids, clusters, etc.) are used on a very large scale and in many different fields. Their contribution in terms of storage space and computing power is obvious [Saadoon et al. 2021, Steen and Tanenbaum. 2023]. Fault tolerance system, in general and specifically in complex systems like distributed ones, is very difficult to implement due to its dynamic nature and complex services in the one hand, and the complexity of distributed systems that are very large, heterogeneous, in the other hand [Herault and Robert. 2015, Raynal 2018, Sreenivasulu et al. 2020]. In addition to its own components, a distributed system uses a network to enable its components to communicate between them [Jalote 1994]. This network is the system's weakest link, since its failure will cause the entire system to break down [Bansal et al. 2011, Koren and Krishna 2020, Kiadehi et al. 2021]. The wide spread of distributed systems in different domains like industry, transportation, telecommunications, and home equipment's has increased the need of reliability and robustness in the several domain applications [Kime 1975, Kiadehi et al. 2021]. This requirement is stronger for critical systems where unforeseen failures may lead to serious consequences [Bansal et al. 2011, Sreenivasulu et al. 2020, Kumari and Kaur. 2021]. Critical infrastructure applications provide services upon which society depends heavily; such applications require constant, dependable operation in the face of various failures, natural disasters, and other disruptive events that might cause a loss of service. So, survivability of critical information systems is an important issue. Survivability is the ability of a system to continue to provide service, though possibly alternate or degraded, in the face of various types of failure and disruption. A fundamental mechanism by which survivability can be achieved in critical information systems is fault tolerance. Fault tolerance provides several benefits for distributed applications including minimized system downtime, enhanced data protection, and increased system reliability. Its use cases span across sectors that require high availability and data integrity, such as financial services, healthcare, telecommunications, and e-commerce. In this context, our proposal represents one approach to ensure these properties. Fault tolerance strategies combine two or more techniques including replication and checkpointing [Kime 1975], which are often the methods used in the implementation of fault tolerance in distributed systems. We can classify these techniques into two categories [Jalote 1994, Herault and Robert. 2015]: reactive fault tolerance and proactive fault tolerance. Reactive fault tolerance strategies try to reduce the effect of failures by triggering when failure occurs. In contrary, proactive fault tolerance try to avoid required treatment assigned to the recovery of failure by replacing proactively suspect components [Koren and Krishna 2020].

This paper addresses the problem of fault tolerance management in distributed systems. As an example of a distributed system, we chose Hadoop because it is an open source, Java-based framework that manages the storage and processing of large amounts of data for applications [Holmes 2014]. Hadoop uses distributed storage and parallel processing to handle big data and analytics jobs, breaking workloads into smaller workloads that can be run simultaneously. Following reference [Bansal et al. 2011], we adopt a strategy of two steps: Detection step and Replica management step. We define two approaches of management fault tolerance in Hadoop framework: **Gossip Hadoop** and **Gray Hadoop**. As mentioned above, the two approaches have both two steps. For the detection step, the two approaches have different behaviors. However, for the replica management step, the two approaches are similar. The remainder of the paper is organized as follows: Section 2 highlights our main contributions. Section 3 describes the model we propose to represent nodes of a distributed system. In Section 4, we describe in detail the proposed

Hadoop-based approach for managing fault tolerance. Section 5 presents and details the results of our experiments. Section 6 concludes the paper and summarizes the level of achievement of the research goal and objectives. It also outlines potential research directions that would build on the results of this study or continue the research initiated in this paper.

2 Contributions

In this research work, we study the problem of fault tolerance management in distributed systems through the Hadoop system [Elkawagy and Elbeh. 2020, Kapil et al. 2020]. To address this problem, we categorize nodes of a distributed system into three classes according to their fault status. In addition to the two classical node status, safety and faulty, we introduce a new node status, which we call *Potentially Faulty Node*. This type of status concerns nodes that are not faulty, but have some potential to lead to a faulty node in the future. According to these three node status, we define three classes of nodes to manage these different node status. To demonstrate the potential of our proposal, we experiment it on different infrastructures with different workloads.

In summary, we can highlight the following main contributions of this paper:

- The proposal of a representation model of nodes in a distributed system.
- The categorization of nodes into three classes according to their faulty status.
- The experimentation of our proposal on different infrastructures with different workloads.

3 Distributed system representation model

In this subsection, we define the model that we use to represent nodes of a distributed system. In this representation model, the nodes of a distributed system are divided into 3 sets: **SN** for *Safety Nodes* (or healthy nodes), **FN** for *Faulty Nodes* and **PFN** for *Potentially Faulty Nodes*:

1. A node N is said to be *Safety* if and only if it is safe and does not contain any faulty components.
2. A node N is said to be *Faulty*, if it fails or has a connection problem with other nodes in the system.
3. A node N is said to be *Potentially Faulty*, if it is not faulty at a time t , but it may fail in the future.

Based on these three categories, we define three node classes that we refer to as **C-SN**, **C-FN** and **C-PFN**. From the fault tolerance perspective, when we start a distributed system these three classes are defined as follows:

- **C-SN** contains all nodes of the distributed system with the hypothesis that these are safety when we launch a distributed system.
- **C-FN** and **C-PFN** are empty.

Fault tolerance refers to the ability of a system (computer, applications, services, network, etc.) to continue operating as per specifications even when one or more of its components encounter a fault [Jalote 1994]. What's more, fault tolerance can happen at different levels such as hardware, software, network, power supply, storage, etc. [Kiadehi et al. 2021]. To assess fault tolerance performance, metrics such as Mean Time Between Failures (MTBF), Mean Time To Repair (MTTR), Availability, and Recovery Time Objectives can be used [Jalote 1994, Kime 1975]. These metrics provide insights into system reliability, resilience, and the impact of failures on overall service availability [Jalote 1994, Herault and Robert. 2015].

All of the metrics described above are continuously calculated and stored in appropriate components in the system. Analyzing the value of these metrics can lead to determining, at a time t , whether a given node is *Safe*, *Faulty*, or *Potentially faulty*. Due to the unpredictable nature of failure, a node of a distributed system may change state during its use. The node status of a distributed system is therefore dynamic. For this purpose, we have defined three types of nodes: *Safety Nodes (SN)*, *Faulty Nodes (FN)* and *Potentially Faulty Nodes (PFN)*. During execution, a node may change its state according to the values of the fault tolerance metrics calculated during execution. As a consequence of this dynamicity, the **C-SN**, **C-FN** and **C-PFN** classes are also able to change their content during execution. For a given node, when its fault tolerance metrics are calculated and analyzed, the fault tolerance manager will assign it to one of these three classes. Therefore, a node may remain in the same class or change classes depending on the current values of its fault tolerance metrics.

4 Hadoop-based approaches for managing fault tolerance in distributed systems

4.1 Background

MapReduce is the most popular data processing technique used to manage distributed applications and services. Hadoop is the standard implementation of MapReduce, which provides capabilities to handle data-intensive applications such as data mining, social data, etc. Hadoop combined with MapReduce gives developers the flexibility to write their applications in any high-level programming language [Holmes 2014, Kapil et al. 2020]. Since in a distributed system, failures of physical and/or logical components, nodes, tasks, jobs, etc. are prevalent, Hadoop handles fault tolerance using master-slave communication through heartbeat messages [Kadirvel et al. 2013, Hu and Dai. 2014, Memishi et al. 2016].

4.2 Management of fault tolerance in Standard Hadoop

The main components of Apache Hadoop are MapReduce and HDFS [Holmes 2014]. Hadoop MapReduce consists of a JobTracker and many TaskTrackers, which constitute the processing master and workers, respectively. The MapReduce workflow is managed by the JobTracker, whose responsibility goes beyond the MapReduce process. For instance, the JobTracker is also in charge of the resource management. HDFS consists of a NameNode and many DataNodes, that is, the storage master and workers, respectively, whereas the NameNode manages the file system metadata, DataNodes hold a portion of data in blocks. The Apache Spark framework uses a master-slave architecture that consists of a driver, which runs as a master node, and many executors that run across

as worker nodes in the cluster. In this framework, we use the Hadoop ecosystem which is composed of four primary modules: (1) Hadoop Distributed File System (HDFS) that provides high-throughput data access and high fault tolerance; (2) Yet Another Resource Negotiator (YARN) that schedules tasks and allocates resources (e.g., CPU and memory) to applications; (3) Hadoop MapReduce, that splits big data processing tasks into smaller ones, distributes the small tasks across different nodes, then runs each task; (4) Hadoop Common (Hadoop Core) that is a set of common libraries and utilities that the other three modules depend on. In a distributed setup, our data are distributed across various worker nodes in an HDFS (Hadoop Distributed File System) setup. The Spark provides a framework to coordinate work among these worker nodes. We choose Hadoop because it is a highly fault-tolerant that has it was designed to replicate data across many nodes. Each file is split into blocks and replicated numerous times across many machines, ensuring that if a single machine goes down, the file can be rebuilt from other blocks elsewhere. The choice between HDFS and cloud-based data systems like ADLS Gen2 and Amazon S3 depends on different objectives and characteristics. While HDFS remains the best choice in traditional big data environments, the cloud-based data systems offer a modern and flexible approach, with some interesting properties like persistent storage, decoupled compute, high accessibility, and cost-effectiveness. The following characteristics can distinguish between data distributed systems:

- Storage Architecture: Distributed File System vs Object-Based Storage: for example HDFS is a distributed file system, designed to store data in blocks across a cluster of machines. In contrary, ADLS Gen2 and Amazon S3 are both object-based storage systems, where data is stored as objects with associated metadata.
- Persistence: HDFS is not inherently persistent, but ADLS Gen2 and Amazon S3 offer persistent storage, ensuring that data remains intact even if clusters are shut down.
- Accessibility and Interoperability: In HDFS, data is bound to a specific Hadoop cluster, and accessing data from one HDFS cluster to another can be challenging. Cloud-based data systems like ADLS Gen2 and Amazon S3 offer greater flexibility.
- Cost Model: HDFS often involves significant upfront capital expenditure for the setup and maintenance of on-premises infrastructure. Cloud storage services like ADLS Gen2 and Amazon S3 follow a pay-as-you-go pricing model, allowing organizations to pay for actual usage, reducing upfront costs and providing scalability.
- Security and Compliance: HDFS relies on traditional security measures, and organizations are responsible for implementing security protocols. Cloud-based data systems offer robust security features.

To manage fault-tolerance, Hadoop is based on the heartbeat principle, because heartbeat messages play a crucial role in ensuring the reliability, availability, and fault tolerance of distributed systems [Kadirvel et al. 2013, Hu and Dai. 2014, Memishi et al. 2016]. In Hadoop, heartbeat is referred to a signal used between a Datanode and Namenode, and between task tracker and job tracker, if the Namenode or job tracker does not respond to the signal, then it is considered there is some issues with data node or task tracker. If the master node does not receive a heartbeat message from a slave node within a configurable timeout value, the slave node is labeled as failed or faulty. At the same time, the successful progress made by the faulty node prior to its failure is neglected, resulting in a huge waste of overprocessing and resource usage. Meanwhile, Hadoop

must wait for the resource scheduler to allocate a free slot to restart the failed tasks that should be running on the faulty node for recovery. This problem encourages researchers to optimize the time spent detecting a failure and recovering from a failure to achieve minimal performance degradation under failure [Hassan and Babar. 2021, Saadon et al. 2021].

4.3 Related works

In this section, we cite some cases and studies of Hadoop's of fault tolerance. In [Memishi et al. 2016], the authors propose an interesting survey on fault tolerance in MapReduce. Samadi et al. in [Samadi et al. 2018] analyze fault tolerance mechanism of Hadoop MapReduce under different types of failures. In [Hassan and Babar. 2021], the authors have carried out a comparative study about fault tolerance techniques in Hadoop. In [Hu and Dai. 2014], authors have proposed an approach to enhance the fault tolerance in a Hadoop cluster. In an another work, [Kadirvel et al. 2013], authors have proposed a management of fault tolerance through early detection of anomalous nodes. In [Saadon et al. 2021], authors have conducted an experimental study for fault detection and recovery techniques on Hadoop MapReduce. Finally in [Asif et al. 2022], the authors have proposed a MapReduce-based model for detecting intrusion using machine leaning technique. This non-exhaustive list of research work on Hadoop MapReduce and many others show the interest of the Hadoop framework for studying fault tolerance problems in distributed systems. Also, these research results along with the increasing importance of MapReduce motivates our goal for improving fault management in Hadoop.

4.4 Our proposal

From a practical point of view, our contribution was deployed on Hadoop framework and we modified some Hadoop methods and classes. Our modifications mainly concern:

- Definition of new variables needed to implement our proposal.
- Some classes and methods of the Hadoop framework were modified. These modifications include: deleting some statements from the original code of Hadoop framework methods; modifying some statements such as *if-then-else* or *for-end*; adding news *if-then-else* and *for-end* statements. The aim of these changes is to modify the behavior of certain Hadoop framework methods to adapt them to our proposals.

In addition, since we are working on the Hadoop framework on multiple virtual machines, we need to:

- Deploy the modified Hadoop framework with our new code on all virtual machines.
- Compile each framework on each virtual machine.
- Set up the communication mechanism between these virtual machines and the Hadoop messaging system.

4.5 A dynamic approach for fault tolerance management

Now, we present our approach to manage faults in Hadoop distributed system. We will begin by presenting the fundamental aspects of our approach before discussing the details of its implementation.

4.5.1 Potentially Faulty Node status

We will discuss here our first contribution, which concerns the introduction of a new node status. The aim of this new node status is to define a preventive method of fault tolerance in Hadoop. This preventive method is provided by a dynamic task replication mechanism and depends on two parameters:

1. The job priority for the job that contains task.
2. The node status that will run task.

To improve fault tolerance in Hadoop, we propose a new node status, which we call *Potentially Faulty Node (PFN)*. Thus, at a given time t , each node will have one of the following node status: *Safe or Safety Node (SN)*, *Faulty Node (FN)*, *Potentially Faulty Node (PFN)*. When we use only two states of a node (*safe* and *faulty*), all faulty nodes are excluded from the system and the entire workload is carried only by the safety nodes. This solution has two disadvantages: (i) as the workload on the nodes increases, performance degrades; (ii) as the workload on a node increases, so does the likelihood that it will fail in the future. By integrating a new node status (*Potentially Faulty Node*), these two disadvantages can be overcome, since the workload will not be fully supported by the safety nodes, but will be distributed between the *safety nodes* and the *potentially faulty nodes*. *Safety nodes* are those that respect the tolerated thresholds of the various fault tolerance metrics. On the other hand, *Potentially faulty nodes* are nodes that are not faulty, but which present a risk of failure in the future. From a fault tolerance point of view, this means that the values of fault tolerance metrics of these nodes are still acceptable, but are currently within a critical confidence interval.

4.5.2 Node status management

Here we will present some characteristics of our proposal. First of all, we partition nodes of a distributed system into three sets as follows: Let's assume that there are n TaskTrackers running. We represent TaskTrackers as a set $TR = \{TR_1, \dots, TR_n\}$, where $n = |TR|$. During the execution, set TR is splitted into three disjoint subsets, namely TS , TF and TP . TS contains *Safety Nodes*, TF *Faulty Nodes*, and TP *Potentially Faulty Nodes*. Initially, all elements TR_i of TR are in TS , while TF and TP are empty. Depending on their status, a node can eventually move from one set to another: for example, from *Safety Node* to *Faulty Node* or from *Potentially Faulty Node* to *Safety Node*, and so on.

4.5.3 Implementation details

The main goal of this paper is to set up a dynamic task replication mechanism as a fault prevention method in case where tasks are executed by a TaskTracker. To achieve this goal, we propose two approaches:

1. *Gray Hadoop*: In this approach, we decide to replicate task when the TaskTracker to which this task is assigned has a *PF* status.
2. *Gossip Hadoop*: In this second approach, we decide to replicate task when the TaskTracker to which this task is assigned has a reputation less than a specific threshold.

Let's consider TR_i be the TaskTracker to which a given task is assigned. For both our proposed approaches, we decide to replicate task when $TR_i \in TP$. This step is called *Replica management* step, and it's the same for the two approaches. However, their implementation needs two steps: *Replica management* step and *Detection management* step in which each approach has a different behavior:

1. *Detection step* in Gray Hadoop: The implementation of this step needs the introduction of two indicators in Hadoop framework:
 - *Neededreplication*: A boolean indicator of replication is added in TrackerStatus class into MapReduce layer.
 - *Nbreplication*: An indicator for authorized number of attempts for a running task is added in TaskInProgress class in MapReduce layer.
2. *Detection step* in Gossip Hadoop: It aims to evaluate the reputation of each node [Bansal et al. 2011]. The reputation is calculated referring to fetch-failure notifications received by the JobTracker. These notifications are sent from a TaskTracker when it tries to execute a reduce task, but it doesn't find the result (output) of a Map task. The system saves all those notifications, also called *Gossip*, to penalize the suspected TaskTracker (TaskTracker that has executed the Map task). Indeed, in the beginning, each TaskTracker is assigned with a default reputation. When the JobTracker receives any fetch-failure notification, it decreases the value of the reputation associated to the suspected TaskTracker. If the reputation is lower than a specific threshold, then the state of suspected TaskTracker change to *PF* and all tasks that are assigned to it will be replicated. The implementation of this concept of *PF* TaskTracker in the case of Gossip Hadoop is achieved by three data structures and one attribute:
 - (a) Structures
 - *Gossip*: It contains the number of fetch-failure notifications *Gossip* which are considered for a specific period. After that, all notifications will be deleted.
 - *GossipList*: List of *Gossip* that are chronologically ordered.
 - *GossipQueue*: Contains a list of TaskTracker that have suspected a given TaskTracker.
 - (b) *TaskTrackerToReputation* attribute: List of reputations for each TaskTracker.
3. *Replica management* step for the two approaches: In this step, we use different managing replica in the two approaches, depending on the task types:
 - Running tasks: In this case, the JobTracker assigns different tasks (Map and Reduce) to TaskTracker.
 - Reception of successful termination of a Reduce task: This case does not need specific treatment.
 - Reception of failed task: If the failed task has other successful saved replicas, we replace data structures of the failed task by the successful one, otherwise we signal that this task is failed and it should be executed again.

We will now describe the behaviour of our proposal in the case where we use nodes with a low degree of reputation. First of all, the probability theory has proved that the gossip protocol theoretically enables all nodes to receive messages at last. rounds, every node receives the message. Clearly, the time complexity is $O(\log n)$. The decentralized distribution of information in the gossip protocol means that each node that has received information only needs to transmit the message to its neighbouring nodes. Compared with the centralized transmission mode, the gossip protocol has a lower likelihood of information blockage in the transmission process, thus reducing network overhead traffic. Based on the node classification model (see section 3), we define a node reputation model to distinguish nodes in terms of their fault tolerance and their ability to withstand faults. To classify nodes according to their reputation, we use the node structuring model (see section), which defines three classes: SN, FN, PFN. By eliminating the nodes in the FN set (faulty nodes), we are left with two sets: SN and PFN. In each of these sets, nodes are ranked in descending order according to two criteria: capacity in terms of computing and storage resources, and reputation score according to the evolution of their degree of reaction to failures (faults) over time. So, each time we need to choose a node to replace a failed node, we proceed as follows: Select a node from the SN set according to resource and reputation score criteria (choose the best). If the SN set is empty, we'll choose a node according to the same criteria from the PFN set. With this selection procedure, we ensure that the chosen node meets both performance and reputation criteria in terms of fault tolerance.

5 Experiments and Results

In this section, we present and discuss the results of our experiments. First, we show experimental results for a virtual infrastructure, namely **Virtual Multi-Node Cluster (VMC)**. Our first goal is to prove that the approaches have an OverHead (*OH*) (see Equation 1 page 81 for definition) less than the standard Hadoop. Second, we choose to experiment our approaches in a physical infrastructure, more precisely in a **Physical Multi-Node Cluster with WIFI connection (PMCW)**. Our aim is to confirm that our approaches are also applicable in the case of a physical multi-node cluster; finally, we want to compare the results with Virtual Multi-Node Clusters (**VMC**).

5.1 Evaluation metrics

To evaluate our approaches, we have chosen to analyze the *OH* (OverHead) cost caused by the occurrence of a failure. This is justified by the main objective of our work, which is to reduce the execution time for a job or a pool of tasks running in a failed node. After a series of preliminary analyses to validate our proposal, we consider the following hypothesis:

- A data size between 100 MiB and 600 MiB with a frequency of 100 MiB.
- Multiple jobs between 10 and 40 with a frequency of 10.

These choices take into account the heterogeneity constraint in a distributed system. In fact, hardware and software characteristics in a distributed system are different from one component to another. Therefore, the choice of data size and the number of jobs that can be run should take this heterogeneity characteristic into account. In addition, to have as

homogeneous nodes as possible, we decided to implement a misconfiguration on the machine that has a limit storage. The formula used to calculate the OH cost is :

$$OH_v = TT_v - ST_v \quad (1)$$

where:

1. v : is the Hadoop version ($v = d$ for Standard Hadoop, $v = s$ for Gossip Hadoop, $v = r$ for Gray Hadoop).
2. OH_v : OverHead in Hadoop version v .
3. ST_v : Execution time when task is successfully completed without any failure in Hadoop version v .
4. TT_v : Execution time when failure is tolerated due to occurrence of failure in Hadoop version v . To configure our test environment, we should set 3 parameters:
 - For all tests, our platform contains only four machines. So it is recommended to use a replication factor equal to 2.
 - The approaches assessment method will be based on the OH_v resulted by a failed task. So, we should try to solve all other causes that could have consequences to this OH_v .
 - Usually, node performances are different in a distributed system (because heterogeneity). As a result, the execution time will be different from one node to another. To solve this problem, we choose to allocate more jobs to more efficient nodes rather than others.

5.2 Benchmarks

Hadoop includes many testing tools and benchmarks that can be find in installation directory with the name “*hadoop-*examples*.jar” [Holmes 2014]. Each benchmark is adapted to test particular property. We have chosen to use *Terasort* benchmark which is the most popular benchmark of Hadoop used to test both layers HDFS and MapReduce. It consists of running a program that sort input data, and it is composed of three elements:

1. *TeraGen*: generate random data to be sorted.
2. *TeraSort*: sort data generated by *TeraGen*.
3. *TeraValidate*: valid data sorted by *TeraSort*. As we will run a lot of tests with different parameters, such as data size and number of jobs, we have written a shell program (see script 1) which can be adapted to these parameters.

To run shell program (see script 1), we should specify four parameters which are:

1. \$1: The test number that identify a given test. This number is used to create needed directories for this test.
2. \$2: Parameter to specify the size of data used in a given test. This number is used to remember the data size in MiB. It’s not used by the framework. However, the data size used by framework is \$3.

3. \$3: The data size with 100 Bytes of unit, which is the block size in Hadoop framework. To have the real data size in byte, this number should be multiplied by 100. So, we obtain the parameter \$2 but in MiB.
4. \$4: Represents the number of jobs to run in a given test. The shell program is mainly composed of three parts: the first part, lines 5 to 8, contains instructions used to prepare tests; it consists of removing some files and running Hadoop services (HDFS and MapReduce layers). The second part, from lines 9 to 13, is a loop that creates directories needed to run tests. The third part, from lines 15 to 21, is a second loop that run many jobs (parameter \$4) with a specific data size (parameter \$3).

Script 1

```

1: #/bin/bash
2: # Edited by Riadh FREFITA
3: $1:test number, $2: data size (MiB)
4: $3: data size (100 Byte), $4: number of jobs
5: rm -R /app/hadoop/tmp
6: /usr/local/hadoop/hadoop-2.4.1/bin/hadoop namenode -format
7: /usr/local/hadoop/hadoop-2.4.1/sbin/start-all.sh
8: ps -aux | grep java | awk print $12
9: for $i=1 to $4 do
10:   mkdir -p output$(( $i + $1 ))
11:   mkdir -p input$(( $i + $1 ))
12:   mkdir -p validate$(( $i + $1 ))
13: end for
14: d1='date + %s'
15: for $i=1 to $4 do
16:   hadoop jar *examples*.jar teragen $3 input$1&
17:   hadoop jar *examples*.jar terasort input$1 output$1&
18:   hadoop jar *examples*.jar teravalidate output$1 validate$1&
19: end for
20: d2='date + %s'
21: echo $((d2-d1)) >> resume$2
22: /usr/local/hadoop/hadoop-2.4.1/sbin/stop-all.sh

```

5.3 Experimentations on Virtual Multi-Node Cluster (VMC)

To experiment with our proposal, we first tested it in a virtual infrastructure. As shown in Table 1, we considered a Virtual Multi-Node Infrastructure (VMC) containing only 4 virtual machines installed on VMware Workstation: the performance of the nodes in a distributed system is so different, as mentioned in subsection 5.1. As result, the execution time will be different from one node to another. To solve this problem related to node heterogeneity, we choose to allocate more jobs to Virtual Machines which are more efficient than others, in terms of resources and performance.

In the rest of this subsection, we present some experimental results to evaluate our proposed approaches (Gossip Hadoop and Gray Hadoop). To do so, we first present,

for each number of jobs, OH results for different data sizes (100 MiB, 200 MiB, ..., 600 MiB). Then, we show the distribution of OH for each of them. Finally, we give some conclusions and interpretations. Before presenting and discussing our results, we define the equations used to compute our evaluation metrics:

$$TT_v = ST_v + OH_v \quad (2)$$

$$P\%_v = (OH_v \div TT_v) \times 100 \quad (3)$$

$$GsH = ((OH_s \div TT_s) \div (OH_d \div TT_d)) \times 100 \quad (4)$$

$$GrH = ((OH_r \div TT_r) \div (OH_d \div TT_d)) \times 100 \quad (5)$$

where:

- v : indicates the Hadoop version; $v = d$ for Standard Hadoop, $v = s$ for Gossip Hadoop, $v = r$ for Gray Hadoop
- OH_v : OverHead in Hadoop version v
- $P\%_v$: Percentage of OverHead in Hadoop version v
- GsH : Percentage of OverHead in Gossip Hadoop
- GrH : Percentage of OverHead in Gray Hadoop

In Table 1, we show the OH results for 10 jobs for all variants: Standard Hadoop, Gossip Hadoop and Gray Hadoop. The $ST(Sec)$ row indicates the time taken to complete a job (in seconds), if the job was completed successfully without failing. The first block of three lines, called $Hdp(Sec)$, shows the OH results (in seconds) for Standard Hadoop with failure. The second block of three lines, called $Gsp(Sec)$, shows the OH results in seconds for Gossip Hadoop with failure. The third block of three lines, called $Gry(Sec)$, shows the OH results in seconds for Gray Hadoop with failures. Each block contains three rows. The first one, called TT_v , shows the time required to complete the test (see Equation 2 page 83). The second row represents the overhead caused by a failure (see Equation 1 page 81). The third line is the percentage of overhead in TT_v . The fourth block (STAT) gives some statistics about the percentage of overhead caused by Gossip and Gray Hadoop in the Standard Hadoop. The Gossip Hadoop overhead percentage in Standard Hadoop is shown in the first row of this fourth block (GsH). The second row (GrH) represents the overhead percentage of Gray Hadoop in Standard Hadoop. To analyze and evaluate the results, we show the distribution of OH_v for each approach. In the case of 10 jobs, as shown in Figure 1, for both approaches, Gossip and Gray Hadoop, OH_v is lower than Standard Hadoop. We can easily see that as the data size increases exactly from 400 MiB, the difference between Standard Hadoop and our approaches becomes more and more important. From 400 MiB, Gossip and Gray Hadoop are more and more similar, and they represent about 50% of the overhead of Standard Hadoop (see GsH and GrH in bold in Table 1). In Tables 2, 3 and 4, we show the OH_v results for 20, 30, and 40 jobs, respectively, for all variants: Standard Hadoop, Gossip Hadoop, and Gray Hadoop.

Data		100	200	300	400	500	600
ST (sec)		1441	2614	3487	4082	05101	06203
Hdp (sec)	TT_d	2140	4246	6256	7579	11580	16123
	OH_d	0699	1632	2769	3497	06479	09920
	Pc_d	32.60	38.44	44.26	46.14	55.95	61.53
Gsp (sec)	TT_s	2050	3850	5430	5921	07985	10105
	OH_s	0609	1236	1943	1839	02884	03902
	Pc_s	29.71	32.10	35.78	31.06	36.12	38.61
Gry (sec)	TT_r	2020	3815	5248	5590	07249	08899
	OH_r	0579	1201	1761	1508	02148	02696
	Pc_r	28.66	31.48	33.56	26.98	29.63	30.30
STAT	GsH	0.91	0.84	0.81	0.67	0.65	0.63
	GrH	0.88	0.82	0.76	0.58	0.53	0.49

Table 1: Overhead values for 10 Jobs in VMC infrastructure

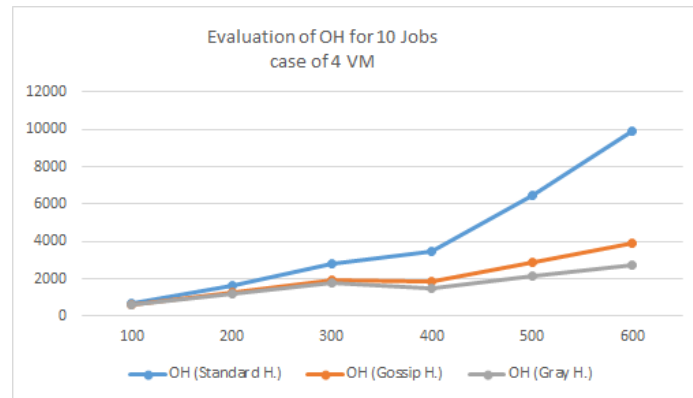


Figure 1: Chart of overhead cost for 10 Jobs in VMC infrastructure

Data		100	200	300	400	500	600
ST (sec)		2904	4610	06625	08104	10209	12516
Hdp (sec)	TT_d	4073	6850	10555	14450	19570	27883
	OH_d	1169	2240	03930	06346	09361	01536
	Pc_d	28.70	32.70	37.23	43.92	47.83	55.11
Gsp (sec)	TT_s	3820	6231	09180	11255	14452	18379
	OH_s	0916	1621	02555	03151	04243	05863
	Pc_s	23.98	26.02	27.83	28.00	29.36	31.90
Gry (sec)	TT_r	3615	5802	08520	10630	13536	17191
	OH_r	0711	1192	01895	02526	03327	04675
	Pc_r	19.67	20.54	22.24	23.76	24.58	27.19
STAT	GsH	0.84	0.80	0.75	0.64	0.61	0.58
	GrH	0.69	0.63	0.60	0.54	0.51	0.49

Table 2: Overhead values for 20 Jobs in VMC infrastructure

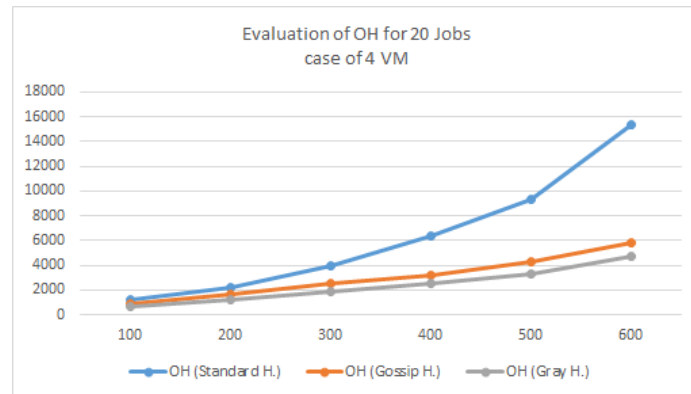


Figure 2: Chart of overhead cost for 20 Jobs in VMC infrastructure

Data		100	200	300	400	500	600
ST (sec)		4640	6593	9957	12117	14890	18707
Hdp (sec)	TT_d	5988	8969	14891	18992	26690	36880
	OH_d	1348	2376	04934	6875	11800	18173
	Pc_d	22.51	26.49	33.13	36.20	44.21	49.28
Gsp (sec)	TT_s	5610	8110	12853	15555	19335	24920
	OH_s	0970	1517	02896	3438	04445	06213
	Pc_s	17.29	18.71	22.53	22.10	22.99	24.93
Gry (sec)	TT_r	5604	8110	12257	14968	18824	23585
	OH_r	0964	1517	02300	2851	03934	04878
	Pc_r	17.20	18.71	18.76	19.05	20.90	20.68
STAT	GsH	0.77	0.71	0.68	0.61	0.52	0.51
	GrH	0.76	0.71	0.57	0.53	0.47	0.42

Table 3: Overhead values for 30 Jobs in VMC infrastructure

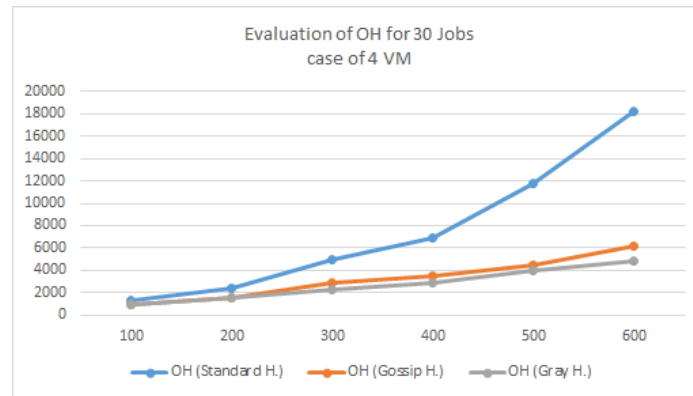


Figure 3: Chart of overhead cost for 30 Jobs in VMC infrastructure

Data		100	200	300	400	500	600
ST (sec)		5558	9648	13611	15950	19985	24622
Hdp (sec)	TT_d	6788	12669	18945	24438	32533	42751
	OH_d	1230	03021	05334	08488	12548	18129
	Pc_d	18.12	23.85	28.16	34.73	38.57	42.41
Gsp (sec)	TT_s	6380	11526	16443	19277	24208	30290
	OH_s	0822	01878	02832	03327	04223	05668
	Pc_s	12.88	16.29	17.22	17.26	17.44	18.71
Gry (sec)	TT_r	6189	10909	15546	18575	23399	29270
	OH_r	0631	01261	01935	02625	03414	04648
	Pc_r	10.20	11.56	12.45	14.13	14.59	15.88
STAT	GsH	0.71	0.68	0.61	0.50	0.45	0.44
	GrH	0.56	0.48	0.44	0.41	0.38	0.37

Table 4: Overhead values for 40 Jobs in VMC infrastructure

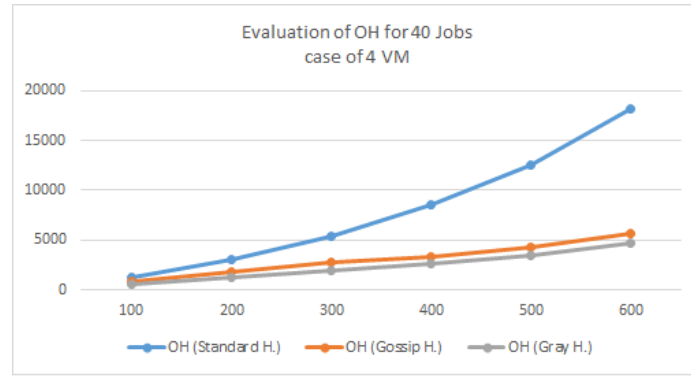


Figure 4: Chart of overhead cost for 40 Jobs in VMC infrastructure

5.4 Physical Multi-node Cluster with WIFI connection

We recall that node performances are so different in a distributed system. As a consequence, the execution time will be different from one node to another. To solve this problem related to nodes heterogeneity, we choose to allocate more jobs to virtual machines that are more efficient than others in the VMC infrastructure. In this subsection, we present some experimental results to evaluate our approaches (Gossip and Gray Hadoop) in the case of Physical Multi-node Cluster with WIFI connection. To do so, we present OH_v results for each data size (100 MiB to 600 MiB) and for different number of jobs (30 and 40).

Data		100	200	300	400	500	600
ST (sec)		6120	11090	18550	22840	24590	31900
Hdp (sec)	TT_d	8500	16592	29644	40985	47088	64900
	OH_d	2380	05502	11094	18145	22498	33000
	Pc_d	28.00	33.16	37.42	44.27	47.78	50.85
Gsp (sec)	TT_s	7926	14875	25358	31408	33954	44493
	OH_s	1806	03785	06808	08568	09364	12593
	Pc_s	22.79	25.45	26.85	27.28	27.58	28.30
Gry (sec)	TT_r	7217	13280	22450	27949	30130	39517
	OH_r	1097	02190	03900	05109	05540	07617
	Pc_r	15.20	16.49	17.37	18.28	18.39	19.28
STAT	GsH	81.39	76.75	71.75	61.62	57.72	55.65
	GrH	54.29	49.73	46.42	41.29	38.49	37.92

Table 5: Overhead values for 30 Jobs in VMC infrastructure with WIFI connection

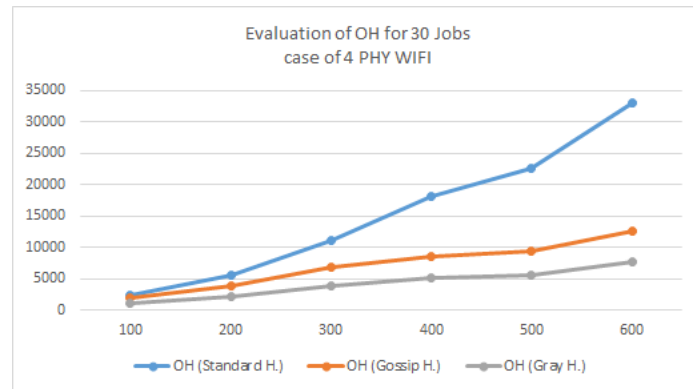


Figure 5: Chart of overhead cost for 30 Jobs in VMC infrastructure with WIFI connection

Data		100	200	300	400	500	600
ST (sec)		08180	14784	24726	31310	32880	42910
Hdp (sec)	TT_d	10125	18853	35420	48621	55900	75580
	OH_d	01945	04069	10694	17311	23020	32670
	Pc_d	19.21	21.58	30.19	35.60	41.18	43.23
Gsp (sec)	TT_s	09524	17351	30005	38205	41250	54810
	OH_s	01806	03785	06808	08568	09364	12593
	Pc_s	22.79	25.45	26.85	27.28	27.58	28.30
Gry (sec)	TT_r	07217	13280	22450	27949	30130	39517
	OH_r	00932	02022	03642	06127	06962	09979
	Pc_r	10.23	12.03	12.84	16.37	17.47	18.87
STAT	GsH	73.45	68.54	58.26	50.70	49.27	50.22
	GrH	53.25	55.75	42.53	45.98	42.42	43.65

Table 6: Overhead values for 40 Jobs in VMC infrastructure with WIFI

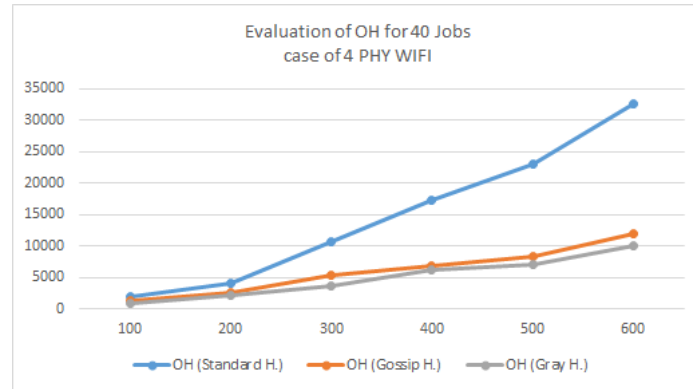


Figure 6: Chart of overhead cost for 40 Jobs in VMC infrastructure with WIFI connection

Then, we present the distribution of OH_v for each of them in Tables 5 and 6. In these tables, we show the OH_v results for 30 and 40 jobs, respectively, for all variants: Standard Hadoop, Gossip Hadoop, and Gray Hadoop. In the case of 30 and 40 jobs, as shown in Figures 3 and 4, OH_v is lower for both approaches (Gossip and Gray Hadoop) than for Standard Hadoop. We can easily see that the difference between Standard Hadoop and our approaches becomes more significant as the data size increases. From 400 MiB, Gossip and Gray Hadoop are more and more similar, and they represent approximately 50% of overhead of Standard Hadoop (see GsH and GrH in bold in Tables 5 and 6).

5.5 Results and Discussions

Through our various experiments, we were able to highlight some interesting results regarding fault tolerance in distributed systems such as Hadoop. In the following, we summarize the most significant results.

1. Regardless of the data size and number of jobs, Gossip Hadoop and Gray Hadoop present both an overhead lower than Standard Hadoop.
2. When increasing the size of data, Gray Hadoop overhead is constantly lower than that of Gossip Hadoop. Moreover, there are very close.
3. When increasing the number of jobs, Gray Hadoop overhead is constantly lower than that of Gossip Hadoop. Because the overheads of both approaches (Gossip Hadoop and Gray Hadoop) are very closed, we decide to compare the results of Standard Hadoop with those of our proposed approaches (Gossip Hadoop and Gray Hadoop).
4. When increasing the size of data, the difference of overhead between Standard Hadoop and the average overhead of our proposed approaches becomes important. We can easily notice that increasing the data size, both approaches significantly improve the time needed to tolerate failure, comparatively to Standard Hadoop. The difference is more important for large data size (500 MiB and 600 MiB) than for small data size (100 MiB and 200 MiB).
5. When increasing the number of jobs, the overhead difference between Standard Hadoop and the average of overhead for our two approaches becomes more and more constant.
6. Indeed, for 30 and 40 jobs, the distribution of the difference of overhead between Standard Hadoop and the average of overhead for our two approaches are very close. However, the two distributions of difference of overhead for 10 and 20 jobs are clearly different.

All these remarks are summarized in Table 7.

#Job	AVR of ST.WF	AVR of SH.WF	AVR of GS.WF	AVR GR.WF	AVR
10	2.78	3.93	4.01	4.73	3.86
20	2.82	4.11	4.37	4.81	4.03
30	2.81	4.56	5.16	3.53	4.01
40	2.81	4.09	4.63	4.21	3.93
AVR	2.81	4.17	4.54	4.32	3.96

Table 7: Comparison table

where :

- ST.WF : ratio (Average -AVR-) of ST in Virtual Multi-Node Cluster by the same one in Physical Multi-Node Cluster with WIFI connection

- SH.WF : ratio of OH_v in Standard Hadoop in Virtual Multi-Node Cluster by the some one in Physical Multi-Node Cluster with WIFI connection
- GS.WF : ratio of OH_v in Gossip Hadoop in Virtual Multi-Node Cluster by the some one in Physical Multi-Node Cluster with WIFI connection
- GR.WF : ratio of OH_v in Gray Hadoop in Virtual Multi-Node Cluster by the some one in Physical Multi-Node Cluster with WIFI connection

According to results shown in Table 7, we can deduce that Virtual Multi-Node Cluster presents a overhead two times lower than Multi-Node Cluster with WIFI connection. That means that virtual infrastructure is always a best way to run a distribution Hadoop application than a physical one with a slow connection.

6 Conclusions and Future Work

In this paper, we have presented two approaches to fault tolerance management in Hadoop distributed systems. The proposed approaches aim to improve the management of fault tolerance in both virtual and physical infrastructures. The main idea is to establish a dynamic task replication mechanism as a preventive method of fault tolerance. The first approach, called Gray Hadoop, is based on the introduction of a new state of nodes, which we call *Potentially Faulty Node (PFN)*. Hence, Gray Hadoop approach decides to initiate a dynamic task replication mechanism if and only if task is assigned to *PFN* status nodes. However, the second approach called Gossip Hadoop based on node reputation, decides to initiate a dynamic task replication mechanism if and only if task is assigned to node with a high fault tolerance reputation. In this work, we have considered only task failures and have conducted our experiments on virtual and physical infrastructures. As a perspective, we first want to consider two other types of failures, like master failure and slave failure. In the future, we also want to test our proposal on a large number of nodes to study its scalability and its ability to handle a growing amount of workload.

References

- [Asif et al. 2022] Asif, M., Abbas, S., Khan, M.A., Areej, F., Khan, M., Lee, S.W.: “MapReduce based intelligent model for intrusion detection using machine learning technique”; Journal of King Saud University - Computer and Information Sciences, 34, 2, (2022), 9723-9731.
- [Bansal et al. 2011] Bansal, S., Sharma, S., Trivedi I.: “A detailed review of techniques in distributed system”; International Journal on Internet & Distributed Computing System, 1, 2, (2011), 33-39.
- [Elkawkagy and Elbeh. 2020] Elkawkagy, M., Elbeh, H.: “High Performance Hadoop Distributed File System”; International Journal of Networked and Distributed Computing, 8, 3, (2020), 119-123.
- [Hassan and Babar. 2021] Hassan, A., Babar, N.: “Analysis and implementation of reactive fault tolerance techniques in Hadoop: a comparative study”; The Journal of Supercomputing, 77, 7, (2021), 7184-7210.
- [Herault and Robert. 2015] Herault, T., Robert, Y.: “Fault-Tolerance Techniques for High-Performance Computing”; Springer-Verlag, (2015).
- [Holmes 2014] Holmes, H.: “Hadoop in Practice”; Second Edition. Manning Ed. (2014).

- [Hu and Dai. 2014] Hu, P., Dai, W.: “Enhancing Fault Tolerance based on Hadoop Cluster”; *International Journal of Database Theory and Application*, 7, 1, (2014), 37-48.
- [Jalote 1994] Jalote, P.: 1994. “Fault Tolerance in Distributed Systems”; Prentice-Hall Englewood Cliffs, NJ, (1994).
- [Kadirvel et al. 2013] Kadirvel, S., Ho, J., Fortes, J.-AB.: “Fault Management in Map-Reduce Through Early Detection of Anomalous Nodes”; *10th International Conference on Autonomic Computing, ICAC’13*, San Jose, CA, USA, June 26-28, (2013), 235-245.
- [Kapil et al. 2020] Kapil, G., Agrawal, A., Attaallah, A., Algarni, A., Kumar, R., Khan, R.-A.: “Attribute based honey encryption algorithm for securing big data: Hadoop distributed file system perspective”; *PeerJ Computer Science* 6:e259, (2020), <https://doi.org/10.7717/peerj-cs.259>
- [Kiadehi et al. 2021] Kiadehi, B.-K., Rahmani, A.-M., Molahosseini A-S.: “Increasing fault tolerance of data plane on the internet of things using the software-defined networks”; *PeerJ Computer Science*, 2021 7:e543, (2021), <https://doi.org/10.7717/peerj-cs.543>
- [Kime 1975] Kime, C.R.: “Fault-Tolerant Computing: An Introduction and a Perspective”; In *IEEE Transactions on Computers*, C-24, 5, (1975), 457-460.
- [Koren and Krishna 2020] Koren, I., Krishna, C.-M.: “Fault-Tolerant Systems”; 2nd Edition, Elsevier, (2020).
- [Kumari and Kaur. 2021] Kumari, P., Kaur, P.: “A survey of fault tolerance in cloud computing”; *Journal of King Saud University, Computer and Information Sciences*, 33, 10, (2021), 1159-1176.
- [Memishi et al. 2016] Memishi, B., Ibrahim, S., Pérez, M.-S, Antoniu, G.: “Fault Tolerance in MapReduce: A Survey”; In *Resource Management for Big Data Platforms (Book)*, Springer, (2016), 205-240.
- [Raynal 2018] Raynal, M.: “Fault-Tolerant Message-Passing Distributed Systems: An Algorithmic Approach”; ISBN: 9783030068035, (2018).
- [Saadoon et al. 2021] Saadoon, M., Hamid, S.-HA., Sofian, H., Altarturi, H., Nasuha, N., Azizul, Z.-H., Sani, A.-A., Asemi, A.: “A Experimental Analysis in Hadoop MapReduce: A Closer Look at Fault Detection and Recovery Techniques”; *Sensors (Basel, Switzerland)*, 21, 11, (2021), 3799.
- [Samadi et al. 2018] Samadi, Y., Zbakh, M., Tadonki, C.: “Analyzing fault tolerance mechanism of Hadoop Mapreduce under different type of failures”; *4th International Conference on Cloud Computing Technologies and Applications (Cloudtech)*, Brussels, Belgium, (2018), 1-7.
- [Sivagami and Easwarakumar. 2019] Sivagami, V.-M., Easwarakumar, K.-S.: “An Improved Dynamic Fault Tolerant Management Algorithm during VM migration in Cloud Data Center”; *Future Generation Computer Systems*. 98, (2019), 35-43.
- [Sreenivasulu et al. 2020] Sreenivasulu, G., Srinivas, P.-VS., Goverdhan, A.: “A Distributed Fault Analysis (DFA) Method for Fault Tolerance in High-Performance Computing Systems”; In: Bansal, J., Gupta, M., Sharma, H., Agarwal, B. (eds) *Communication and Intelligent Systems. ICCIS Lecture Notes in Networks and Systems*, Springer, Singapore, 10, (2020).
- [Steen and Tanenbaum. 2023] van Steen, M., Tanenbaum, A.-S.: “Distributed Systems, 4 edition”; Maarten van Steen (ed), (2023), ISBN10: 9081540637.