# ASYMMETRIC DISTRIBUTED LOCK MANAGEMENT IN CLOUD COMPUTING

A THESIS SUBMITTED TO
THE FACULTY OF ACHITECTURE AND ENGINEERING
OF
EPOKA UNIVERSITY

BY

ARTUR KOÇI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

JANUARY 2019

Approval of the thesis:

**ASYMMETRIC DISTRIBUTED LOCK MANAGEMENT IN CLOUD COMPUTING**

Submitted by Artur Koçi in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Department of Computer Engineering, Epoka University** by,

Assoc. Prof. Dr. Sokol Dervishi
Dean, Faculty of Architecture and Engineering

Dr. Arban Uka ______________________
Head of Department, **Computer Engineering, EPOKA University**

Prof. Dr. Betim Çiço ______________________
Supervisor, **Computer Engineering Department, EPOKA University**

**Examining Committee Members:**

Prof. Dr. …………….. ______________________
………………. Dept., ………….. University

Prof. Dr. ……………. ______________________
………………. Dept., ………….. University

Assoc. Prof. Dr. ,,,,,,,,,,,,,,,,,,,, ______________________
………………. Dept., ………….. University

Assoc. Prof. Dr. …………………. ______________________
………………. Dept., ………….. University

Assist. Prof. Dr. ……….. ______________________
………………. Dept., ………….. University

**Date:** 05/03/19

**I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.**

Name, Last name: Artur Koçi.

Signature:

# ABSTRACT

## ASYMMETRIC DISTRIBUTED LOCK MANAGEMENT IN CLOUD COMPUTING

Koçi, Artur

Ph.D., Department of Computer Engineering

Supervisor: Prof. Dr. Betim Çiço

Cloud computing have become part of our daily lives. They offer a dynamic environment for costumers to store and access their data at any time in any location. The developments of social networks have led to the necessity to build a solution which is easily accesible and available when required. Cloud computing provide a solution that does not depend on the location and can offer a wide range of services, while being free from failure and errors.

Although there is an increase in the usage of the cloud storage services, there is still a significant number of aspects such as instant servers failures, network partitioning and natural disasters that require to be carefully addressed. Another important point that is vital for a sustainable cloud is the implementation of an algorithm which will coordinate and maintain concurrent access and keep shared files free from errors. One of the main approaches to overcome these problems is to provide a set of servers which will act as a gateway between clients and storage nodes.

In this thesis we propose a new approach which provides an alternative solution to the main problematics related with cloud storages. The approach is based on multiple strategies for eliminating the problem of node failure and network partitioning while providing a complete distributed environment. In our approach, every server acts as a master server for its own requests and can provide service to its clients without interacting with other master servers. The concurrent access is maintained in an asymmetric way through our lock manager algorithm with the least communication among other master servers. According to the state of a specific file, master server can execute any received request without communicating with other master servers and only when additional information is required does further communication occur. In our approach the network partitioning or failure of one or more master servers has no effect on the other part of the cloud. To improve availability, we associate every master server with a failover server which takes up the duty of a master when the master server fails or becomes obsolete. To measure the performance of our approach we have performed various tests and the results are presented in detailed graphs.

**Keywords:** Cloud Computing; Cloud Storage; Distributed Systems; Lock Manager Algorithm; Concurrent Access; Data Availability; Data Durability.

# ABSTRAKT

## MANAXHIMI ASIMETRIK I BLLOKUESVE TË SHPËRNADARË NË RETË KOMPJUTERIKE

Koçi, Artur

Doktoraturë, Departamenti i Inxhinierisë Kompjuterike

Udhëheqësi: Prof. Dr. Betim Çiço

Retë kompjuterike janë bërë pjesë e jetës sonë të përditshme. Ato ofrojnë një mjedis dinamik për konsumatorët për të ruajtur dhe aksesuar të dhënat e tyre në cdo vendndodhje. Zhvillimi i rrjeteve sociale shfaq nevojën për të ndërtuar një zgjidhje e cila është lehtësisht e aksesueshme dhe e gatshme kurdo që kërkohet. Retë kompjuterike sigurojnë një zgjidhje që nuk varet nga vendndodhja dhe ofron një gamë të gjërë shërbimesh, duke qenë të lira nga dështimet dhe gabimet.

Megjithëse ka një rritje në përdorimin e shërbimeve të ruajtjes në re, akoma ka një numër të konsiderueshëm aspektesh, siç mund të jenë dështimet e çastit të serverave, ndarja e rrjetit dhe fatkeqësitë natyrore, të cilat duhet të adresohen me kujdes. Një tjetër pikë e rëndësishme që është jetësore për një re të qëndrueshme është implementimi i një algoritmi që është në gjendje të koordinojë dhe mirëmbajë qasjet e njëkohshme dhe të mbajë skedarët e shpërndara pa gabime. Një nga qasjet kryesore për të kapërcyer këto problematika është ofrimi i një grupi serverash të cilët do të veprojnë si një portë midis klientëve dhe nyjave të ruajtjes.

Në këtë tezë ne propozojmë një qasje të re e cila u jep përgjigje problematikave kryesore që lidhen me depot e reve. Kjo qasje bazohet në disa strategji për eliminimin e problemit të dështimit të nyjave dhe të veçimit të rrjetit duke siguruar një mjedis të shpërndarë të plotë. Në qasjen tonë çdo servër vepron si servër kryesor për kërkesat e tij dhe mund të ofrojë shërbim për klientët e vet pa pasur nevojë të ndërveprojë me servërat kryesorë të tjerë. Qasja e njëpasnjëshme mirëmbahet në një mënyre asimetrike përmes algoritmit tonë të menaxhimit të bllokuesve duke ofruar komunikimin më të vogël të mundshëm ndërmjet servërave kryesorë të tjerë. Sipas gjendjes specifike të një skedari, servëri kryesor mund të ekzekutojë çdo kërkesë të marrë pa komunikuar me servërat kryesorë të tjerë, dhe vetëm kur kërkohet informacion shtesë ndodh komunikim i mëtejshëm. Në qasjen tonë veçimi i rrjetit ose dështimi i një ose më shumë servërave kryesorë nuk ndikon në pjesët e tjera të resë. Për të përëmisuar disponueshmërinë, ne e shoqërojmë çdo servër kryesor me një servër në gatishmëri, i cili merr detyrën e shefit kur servëri kryesor dështon ose bëhet i paarritshëm. Për të matur performancën e qasjes sonë ne kemi kryer teste të ndryshme dhe rezultatet i kemi paraqitur në  grafikë të detajuar.

**Fjalët kyçe:** Reja Kompjuterike; Ruajtja në Re; Sistemet e Shpërndara; Algoritmi i Menaxhimit të Bllokuesve; Qasje e Njëpasnjëshme; Disponueshmëria e të Dhënave; Qëndrueshmëria e të Dhënave.

***Dedicated to my lovely family***

# ACKNOWLEDGEMENTS

I would like to express my special thanks to my supervisor Prof. Dr. Betim Çiço for his continuous guidance, encouragement, motivation and support during all the stages of my thesis. I sincerely appreciate the time and effort he has spent to improve my experience during my graduate years.

I am also deeply thankful to my friend Dionis Prifti for his precious contribution to this thesis.

My profound gratitude goes to Niuton Mulleti for his tremendous support and advice.

I am also highly appreciative of the advice and support Fatih, Edmond, Igli, Arban, Nevis and Sokol provided whenever I have required.

Last but not least, I sincerely thank my brother Gëzim, my spouse Blerina and my son Dejan for their sacrifice, understanding, patience, encouragement, love and support during those challenging moments of my studies.

# TABLE OF CONTENT

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF ABBREVIATIONS

The following table describes the significance of various abbreviations and acronyms used throughout the thesis.

| **Abbreviation** | **Meaning** |
|---|---|
| **SNT** | Server Node Table |
| **RLT** | Request Lock Table |
| **LFL** | Locked File List |
| **FD** | File Directory |
| **Mout-T** | Migrate-out Table |
| **Min-T** | Migrate-in Table |
| **ADLMCC** | Asymmetric Distributed Lock Management in Cloud Computing |

# CHAPTER 1

# INTRODUCTION

## 1.1 Introduction

Cloud storage services are becoming one of the main options for consumers to store data that are accessible everywhere, from every device and to provide features to connect to network. Since their presentation, most of the biggest companies have been moving their data in cloud and most of financial transactions worldwide are performed online and are not limited by their location.

The overall storage demands are growing at an exponential rate. With the recent developments, users need to access their personal data from every device, from any location, and they would like to share their data with their relatives in a fast and simple manner[1]. Nowadays exist clouds storage such as Google, Amazon, Microsoft, Facebook, etc. which provide different services and store thousands of Terabytes of user's data. One of the biggest concerns on cloud storage services is reliability of user's data. With the reliability in cloud storage we comprehend a cloud that has implemented fault tolerant solution.

Dealing with user data, cloud platforms have to provide scalability while sustaining high availability as well as a proper solution for maintaining concurrent access on the stored data. To meet with these requirements, in the recent years, different datacenters providing online storage services have been built. These storage datacenters use distributed systems such as Google File System GFS [2] or Hadoop [3], [4], which are the biggest distributed file systems designed to store very large data sets reliably and to provide high throughputs. However, to achieve these goals, and to maintain their availability high, these datacenters management requires lots of resources, energy consumption and hardware maintenance. It is very important to design a sustainable solution which implements all the main parameters of e reliable cloud while using resources efficiently.

In big datacenters, in order to avoid single points of failure, developer's strategy is to design a replication node server per each master node. The file concurrency is avoided by creating snapshots and record appends operation. One of the widely used methods to improve data reliability in distributed systems is to distribute data on several storage devices. When using this method, it is essential to associate distributed data with redundant information.

Reliability in distributed cloud storage is defined as the tolerance of the node failure, while availability means any time access to the files regardless of location and time. Another parameter to be considered in distributed cloud storage is efficiency, which is defined as the redundant information stored in the system.

## 1.2 Quality of the Storage Services

According to its definition, in distributed systems, data is not stored in a single unique local storage node but is spread in several storage nodes. A distributed storage solution is composed of hundreds of storage nodes spread among different geographical areas, meaning that part of a single file is physically stored in various nodes that are in multiple locations[1] [5].

One of the main concerns in distributed systems is that distributed nodes can become unavailable in any moment of time. In spite of that, users require the accessibility of their data to meet the service agreement parameters property. The main properties of online storage services are data durability, data scalability, retrieval time and being free from errors.

- **Data Availability:** The data stored in a distributed system should be available for all the time, even if parts of it, stored in different nodes are offline.

- **Data Durability:** In addition to data availability, distributed system should ensure that, after the data are stored in the distributed storage they are never lost. This means that the system must guarantee that even if parts of the file are lost due to the failure of the storage nodes, they can be repaired in a short period without affecting the overall system.

- **Retrieval Time**: The users can access or download their data as quick as possible. There exist other factors that affect the retrieval time such as bandwidth and network congestion, but the cloud service should ensure that the user will have the least possible time from the system side.

Many distributed storage systems use techniques of replication [6], [7], [8], [9], [10] for increasing data reliability. Replication is a process where the whole file is replicated to a certain number of times in different nodes geographically distributed. In case that one of the nodes fails the remaining copies can be available. Replication technique is a process that consumes a lot of space and bandwidth and commits overhead of the system.

To avoid the old methods of replication erasure codes techniques [11], [12], [13] are introduced. Erasure codes is a technique that divides the original file in fragments called chunks and after encoding redundant information it stores them in different distributed nodes. This is a way to efficiently use cloud storage resources and eliminate the need to store many copies of the same file. In erasure codes technique only one copy of a single file is stored. In case of failure the original file can be retrieved using a group of the total number of the stored number of chunks. However, erasure codes are techniques that consume lots of bandwidth to maintain file consistency. For every failure of a node it is required to download information from every remaining node to construct the lost chunk.

Maximum distance separable (MDS) erasure codes described in [13], [12], [11] are erasure codes techniques that can generate a node by contacting a subset of the group used in erasure codes, increasing thus the total number of nodes tolerated to fail. MDS technique increases the reliability of the cloud storage; however, compared to replication it requires more bandwidth consumption while exchanging redundant information.

Another aspect that needs to be optimized in cloud storages is file consistency during concurrent access. Therefore, sustainable lock management should be deployed. Lock managers [14] are techniques for maintaining file inconsistency and avoiding simultaneous access to the same file. They can be implemented for centralized management [8] and for distributed management [15].

In centralized management, all client requests are directed to a single server that plays the role of a master. In this approach master server can face the problem of completion of the resources and can easily get overhead therefore causing service interruption. In distributed management, lock manager is spread in multiple servers, and in case of failure of one of them the access to storage nodes can still continue from other servers offering users the possibility to have access to their data and provide more reliability.

## Objective of this thesis

Although in Chapter 2 more details will be provided about data reliability and concurrent access in shared resources, the main objective of this thesis is to introduce a distributed lock manager algorithm, which will provide a sustainable solution to the main cloud parameters properties for a reliable cloud to increase data availability and keep the shared data free from errors when concurrent access happens while ensuring the least communication possible among servers.

Our solution is implemented in a complete distributed manner and it is centralized server-less architecture. Our design architecture is more akin to a peer-to-peer based architecture by providing all features of a client-server based architecture. We call servers that participate in the distributed storage system master node and storage node or simply nodes, all of which are used to store user`s shared files. All master nodes are equally treated and have the same right on shared files. As in peer-to-peer based architecture [8], [16], [17], master nodes communicate with one another to maintain the concurrent access. In contrast to peer-to-peer architecture, our solution does not perform periodic updates or update all masters any time a file is modified to maintain file consistency.

We propose a model without any kind of global updates and the only communication occurring is message exchange in the moment that one of the master nodes cannot decide itself for the consistency of a certain shared file. Another novelty proposed in this thesis is the elimination of periodic communication between master nodes to maintain their health status. In our proposal there is only one periodic communication which happens between master server and its own failover server to inform about its health status. Failover servers are backup servers associated with each of the master servers and it is used to increase cloud availability. If a master node does not send periodic health information, failover status takes over the master node duty.

## 1.3 Distributed Storage models, challenges and paradigms

As it has already been described, when concurrent access happens in distributed data file consistency [18] is maintained by implementing a centralized management model or distributed management model.

**Challenge 1:** A better understanding of the centralized management

A most common method used in centralized management systems is the method of master slave model. The servers are grouped in master server and slave server. All incoming requests from users are directed to master server and he is responsible for redirecting the request to slaves.

Hadoop [3], [4] is one of the biggest distributed file systems designed using centralized management. The core components of Hadoop are Namenode which stores the file system metadata and Datanodes, which are responsible for storing applications data. Any client who wants to read a data block firstly contacts Namenode for finding the location of data block and then reads the data block from Datanodes.

Another distributing file system using centralized management is Google File System GFS [2]. The architecture of GFS is composed of single master server and multiple chunk servers. Master maintains the file system metadata such as namespace, access control and the current location of the file. The client interacts with the master only for metadata operation and the data-bearing commination goes directly to the chunk server avoiding the overhead of the master server. For avoiding single point of failure, master server has its replication server. File concurrency is avoided by creating snapshots and record appends operation.

Another aspect that needs to be addressed for the centralized management is that they use technique of replication [6], [7], [8], [9], [10] for increasing data reliability. Replication is a process in which the whole file is replicated a certain number of times in different nodes; in case that one of the nodes fails, other copies can be available. The first of the two main concerns to be addressed while this method is implemented is that, as all the requests are directed to a single server, the master server might fail or go offline. This is the case when the entire system enters in failure mode and until the master server is back there is service interruption. The second concern is related to the use of replication technique which is a process that consumes a lot of space and bandwidth causing overhead of the system.

**Challenge 2:** A better understanding of the distributed management

In distributed management, management is spread in multiple servers and in case of failure of one of them, access can continue from the other servers. One solution for maintaining the file consistency in distributed cloud storages is provided by [19]. The offer lock service named Chubby consists in two main components: a cell that is a set of servers also called replicas, and a library that client applications link against. Client Library maintains all the communications between client application and servers. Most of the distributed management systems use [20], [17] conflict resolution procedures for maintaining file consistency. Other algorithms applied in distributed system for maintaining data consistency [21]   enable every node to have information for every other node. They use the technique of reapplication for increasing availability of their stored data. The main concern in distributed system remains the information exchanged between server nodes in order to maintain data consistency and avoid incorrect result when concurrent access happens.

Comparing the two models, both centralized management and distributed management have their positive and negative aspects. The main drawback of the centralized management is related with the fact that the master server becomes a single point of failure. To minimize this effect most of the new approaches offer the solution to provide one master server that acts as the primary server or so-called online mode and the secondary passive servers that are in offline mode. Both servers exchange communication messages and in case that the passive server does not get any response from the active server, it announces its presence and becomes active.

In distributed management this problem is solved by having a set of servers where each of the servers has the duty of master server for its own requests. The requests from users to access any data are directed to the closest server and by using different approaches they maintain the consistency of the shared data. The main drawback with this mode consists in the exchange of the messages to maintain data consistency and consumption of a lot of bandwidth and resources.

Taking into account the above explanation, can we develop a model which will address all the concerns raised by keeping high data reliability?

i) Can we avoid the existence of a single master server or a set of servers which consume lots of bandwidth to receive updates from each other for maintaining their consistency and provide a model where every server acts as master server unaware of the existence of the other master servers?

ii) How can we implement a fully distributed model in a way that every server acts as master server and decrease the number of the exchanged messages among servers without decreasing the system performance?

iii) How to design a system with such requirements and avoid simultaneous read/write in the distributed data stored and keep them free from errors?

iv) How can we analyze and provide sustainable results if this solution can be implemented in cloud with different sizes?

v) What are the drawbacks of implementing this model and how to minimize the impact of these drawbacks?

## 1.4 Contribution of This Thesis

In this thesis we introduce a new distributed lock manager algorithm named asymmetric distributed lock management in cloud computing that will provide a sustainable solution to the main concerns that are faced by distributed storage systems. Lock manager algorithm we introduce addresses the main problematics related with storage systems such as data durability, data scalability, retrieval time, being free from errors and provides a stable solution to the concurrent access in the shared files.

I) With asymmetry of our lock manager we refer to the fact that our algorithm maintains file concurrent access locally without requiring extra communication among other servers and only when further information about the situation of the file is needed, will extra communication with other parts of the clouds occur.

II) Our approach reduces the number of the messages exchanged among servers to maintain system performance. In our solution, the number of messages exchanged will depend on the situation of the file requested and it differs from case to case. Based on the situation of the data requested for being accessed, the number of messages exchanged can vary from zero to a certain number in the cases that servers need to collaborate with other servers.

III) Compared to preview proposals, the novelty of asymmetric distributed lock management in cloud computing depends on the statement that there is no single server which plays the role of master server; there exists a collection of master servers, which independently of each other can execute different tasks achieving high availability of the cloud storage.

IV) All servers work independently of each other and do not keep trace of the other servers, part of the same cloud or trace of the tasks they are performing, increasing every server autonomy and reducing bandwidth consumption.

V) Algorithm sets no limitation to the number of master servers providing high scalability. According to its structure, new servers can be added, or old servers can be removed without affecting the availability of the system.

VI) Another novelty in ADLMCC is provision of a complete distributed lock management, thus avoiding the necessity for a reference server, which sends periodic updates for maintaining cloud performance. Every server contains its own lock manager which performs its tasks in separation from other lock managers and maintains data consistency of the shared data under its command.

VII) In asymmetric distributed lock management in cloud computing, the failure of any of the servers has the least impact on the overall performance and affects only the local users that are accessing it.

## 1.5 Thesis Organization

In the following section, we provide a short summery of the rest of the chapters of this thesis.

**Chapter 2** Distributed Storage Systems Definition, Existing models and Background

We provide a discussion of the existing background in distributed storage systems and we discuss the existing solution provided as well as their millstones through a comprehensive analysis. We focus on the most important factors that affect the quality of distributed storage such as: redundancy, availability and data retrieval time as well as relations among each other.

**Chapter 3** Lock Management in Cloud Computing

In this chapter we discuss the main principles of the lock management in cloud computing, the existing models and paradigms. We give a short description of concurrent access, compare the benefits of each model and analyze their drawbacks.

**Chapter 4** Asymmetric Distributed Lock Management in Cloud Computing: design and functionality

In this chapter we introduce the asymmetric distributed lock management in cloud computing algorithm by giving a detailed explanation of its design, the relationship and collaboration with other distributed lock managers and its functionality. We explain all the components of the asymmetric distributed lock management in cloud computing algorithm and explain the way how lock manager manages the received user requests, maintains data consistency and keeps files free from errors. We present the main novelties of our algorithm and its drawbacks.

**Chapter 5** Implementation of the asymmetric distributed lock management in cloud computing algorithm

In this chapter we describe the implementation of the asymmetric distributed lock management in cloud computing algorithm, the environment and the way how it works. We describe all the modules and the way we collect data which we use in the next chapter for analyzing the performance of the asymmetric distributed lock management in cloud computing algorithm.

**Chapter 6** Performance of the asymmetric distributed lock management in cloud computing algorithm

In this chapter we analyze and discuss the results archived from asymmetric distributed lock management in cloud computing algorithm. We make a comprehensive analysis to find out its conceivable implementation and performance evolution in different cloud computing usage load.

# **Chapter 7** Conclusions

This chapter presents the conclusions that have been achieved by our work and suggestions for future research.

# CHAPTER 2

# DISTRIBUTED STORAGE SYSTEMS: DEFINITION, EXISTING MODELS AND BACKGROUND

## 2.1 Storage Systems Architecture and Design

In this chapter our aim is to introduce distributed storage systems and the important role they play in the design of the online applications. In addition, we give a summary of developments and improvements of the distributed storage systems over years and define the problematics that inspired me and served as motivation for the research done in this thesis.

## 2.2 Local Storage Systems

Every computer system is in need of a mechanism to store personal data and program data persistently[1]. The first dedicated hardware designated to provide data persistence to computers, were hard disk drives. Hard disk drives were first invented by IBM in 1956[11]. Since that time, the hard dick has known enormous developments and nowadays we cannot imagine a computing device without a hard disk. In recent years hard drives have started to be replaced by solid state drives due to their higher speed and increased durability, but they are not still a popular choice due to their price and capacity limitations. However, these local hard disks represent a major problem for being prone to fatal errors and losing data that are locally stored. The traditional solution used to protect those local stored data against fatal failure has been to replicate the stored data in external devices like magnetic tapes or optical devices. Unfortunately, the management of these external

[1]https://www.thinkcomputers.org/the-history-of-the-hard-drive/

backups becomes very difficult and it presents poor scalability and read/write performance. According to that, data management by domestic users was very difficult and many of them faced the problem of losing their personal data. However, enterprise users that could not face the risk of losing their data have been investing in more sophisticated and expensive solutions like RAID[2] storage.

RAID storage is a data storage virtualization technology that combines multiple physical disk drive components into one or more logical units. The main advantages of RAID storage consist in increasing data redundancy, performance improvement, and high availability. Based on RAID configuration, faulty hard disks can be replaced without losing their data and the replacement process can be done without necessarily turning off the storage. Since its first presentation RAID technology has undergone a process of improvements providing different redundancy standards solutions. RAID storages provide weak scalability, and this was the main reason that led to the design of the distributed storage systems that provide a sustainable solution for high availability, redundancy and scalability storages.

## 2.3 Distributed Storage Systems

Distributed storage systems are a collection of storage resources from different dedicated storage devices or computers to build a large storage service. Distributed storage systems are widely used in the most recent cloud storage service and provide high availability, flexibility and durability of the users' stored data. Even though there exist different kinds of distributed systems [1], distributed storage systems can be defined as below:

**Definition 1** A distributed storage platform is an aggregation of distributed computing systems, composed of multiple independent storage nodes that interconnect over a computer network. The main purpose of distributed storages is to combine all these

[2] RAID – Acronym for Redundant Array of Independent Disks

storage nodes and provide a storage service that different applications and users can access over network.

**Definition 2** A storage node [1] is a network component that combines one or more physical storage devices to construct a unique storage component. A physical storage node can refer to different devices such as laptop, desktops, network attached storage (NAS) or any of storage components from data stores.

Due to its distributed nature, distributed storage systems have to face some challenges that did not exit to local storage networks such as: node failure detection, data redundancy, data maintenance, distributing data strategies, bandwidth and parallel access.

(i) **Node failure detection.** A node failure happens when node storage loses connection with the distributed storage system. The failure can occur due to the power outage or when node storage can enter in error mode and becomes unavailable. Distributed storage systems have to implement a mechanism that detects these failures and guarantees that there will not be a disruption of the normal operation mode of the system.

(ii) **Data redundancy:** In distributed storage systems data are spread among different storage nodes and are prone to loss when one or more distributed storage nodes become inactive due to power outage or storage node failure. To prevent the loss of data distributed storage systems needs to implement redundancy schemes and spread data over distributed nodes so that they can recover in any time.

(iii) **Data maintenance:** Data Maintenance refers to the process of regenerating data stored in one of the storage nodes that has permanently failed either by assigning these data to a new storage node or by redistributing them among existing storage nodes. Distributed storage systems have to provide a mechanism that will automatically carry over this process repair data before they become permanently inaccessible.

(iv) **Distributing data strategies:** Distributed storage systems needs to implement a mechanism that will distribute the data among all storage nodes, will provide load balancing and will try to avoid bottleneck when users access any popular content.

(v) **Bandwidth and parallel access:** Considering the fact that bandwidth is not infinite and there is a bandwidth limitation, distributed storage systems should provide strategies to reduce communication exchanged for the maintenance and redundancy process and provide a proper solution that will eliminate parallel access in distributed data that can occur when different applications or users try to write the same object simultaneously.

## 2.4 Distributed Storage Models

Distributed storage systems design should provide solutions that meet all the requirements described in the preview section. Basically, distributed storage systems that meet all these requirements are designed for large data centers. Large data centers such as Google File System GFS [2], Hadoop[3], [4] and IBM's GPFS [22] are examples of such large data centers, and provide high data availability and data scalability. The model of these distributed systems is to have a few master nodes and thousands of storage nodes. Storage nodes are used to store raw chunks of data, while the master node acts as directory service for the file systems and stores the metadata of storage nodes raw chunks. Data redundancy is maintained by storing a minimum of 3 replicas of each file and provides maintenance and repair of the lost data immediately after the detection of storage failure. However, keeping 3 copies of each file consumes a lot of storage and makes the data center very expensive. Another aspect of these models is consumption of the bandwidth used to create replica copies. Implementation of erasure codes has shown a way to reduce storage consumption compared to the old style of the 3-way replicas.

As described in chapter 1, implementing such a distributed file system requires large, well-provisioned and well-managed data centers that require lots of resources, which makes it so expensive that only big enterprises can afford. To reduce the cost of these large data centers, a new style of online storage service has been developed which integrates storage resources from different data centers or even user storages resources into provider's storage. These new trends can be implemented in homogenous systems and between heterogeneous systems.

According to their design architecture, distributed storage systems are divided in two main categories: client-server and peer-to-peer. In client server architecture an entity can behave either as a client or as a server but cannot be both, while a peer-to-peer architecture is completely symmetric and each entity is capable of acting both as a  client and as a server [23].

In client-server based architecture there exists a server which provides service to client requests [24], [2], [25], [26], [27]. In client-server based architecture, there is no ambiguity of who is in control; the server is a central point and is reasonable for data consistency, authentication, replication and providing service to the clients. Client-server based architecture can be categorized into two main groups: Globally Centralized and Locally Centralized [28].
In Globally centralized architecture there is only one server which is responsible for everything that is subject to single point of failure. In other words, this is highly centralized architecture that provides limited scalability.

To address problems related to globally centralized architecture, locally centralized architecture has been introduced. In locally centralized architecture, instead of a server that is responsible for everything, it is a group of servers that share responsibilities among each other providing therefore more resiliency to outages and being more scalable [24], [2], [25], [26].

However, even a locally centralized architecture is vulnerable to failure and limited scalability due to its inherited bottleneck architecture. Client-server based architecture is suitable for monitoring and controlling activities happening in the system and provides high consistency.

To meet new challenges related with operation held in an untrusted environment such as internet, a new wave of systems implementing peer-to-peer architecture has been released. In peer-to-peer based architecture, every node is a potential for being a server or a client and leaves or joins the system as they wish. The benefits of peer-to-peer architecture rely on the fact that peer-to-peer architecture is resilient to outages, provides high scalability and is unrestricted to the public use.

According to their design architecture, peer-to-peer storage can be classified into three main categories: globally centralized, locally centralized and pure peer-to-peer. Each of these categories can have a different degree of centralization that varies from being globally or locally centralized, which means that they have some kind of centralization, to peer-to-peer with no centralization all.

Nasper [29] is an example of globally centralized architecture of peer-to-peer systems, which provides a central server that contains detailed information for every other peer and their respective files. In this model peers are required to contact the central server forgetting information about the other peers. In this model we have a limited scalability and the system itself is a single point of failure.

To overcome the problem of the single point of failure local centralized architecture of peer-to-peer systems has been created [30], [31], [32], [33]. Instead of a central, local centralized architecture has a few masters called super nodes which maintain a repository of metadata through which a group of local users can perform requests and get updates. Super nodes communicate among each other to provide ways for local groups to issue requests to a remote super node rather than broadcasting to all community. Being a centralized point for a group of users, super nodes create a local point of centralization architecture, but they avoid being a single point of failure.

Pure peer-to-peer architecture provides equality between all nodes. The equality between nodes ensures that the model provides the highest scalability among three architectures and it is very adaptive to different environments. Seen from this aspect, pure peer-to-peer architecture is the best choice but being implemented in an asymmetric environment such as internet it becomes very challenging. Most of the users contain asymmetric internet connection biased toward download. This is discouraging users from sharing their resources, which decreases the performance of peer-to-peer systems. Freenet [34], Free Haven [35] and Ivy [36] can be mentioned as examples of pure peer-to-peer systems.

The choice of architecture design plays the biggest role on the distributed systems functionality. Selected architecture determines the boundaries, scalability, effectiveness performance in specific environments and functionality such as routing, consistency and security. While a centralized architecture is suitable to control the environment and functions, but it may lack the scalability functionalities, peer-to-peer is more suitable to be implemented in dynamic environments offering the ability to provide unparalleled scalability and distributed management.

Distributed systems using peer-to-peer based architecture provide high scalability and perform well in dynamic environment. However, the biggest concern in peer-to-peer systems remains reliability of the distributed storage since user resources are not available at all time as in the centralized datacenters which are equipped with dedicated storage resources. In the recent years researchers have been proposing many different peer-to-peer storage solutions and decentralized client-server system to overcome these problems and herein we will discuss the most important ones:

- **Farsite** [8] is a distributed file system designed by Microsoft, which in its logic works as a centralized server but its physical realization is spread among a network of untrusted desktop computers. The main idea of Farsite is to integrate user`s desktop computers resources with the company. The main achievement of Farsite is that it provides centralized management with location transparent to provide logically centralized, secure, and reliable file-storage service.

- **pStore** [37] is a peer-to-peer distributed systems that provides a secure peer-to-peer backup system. The main focus of pStore is to allow users to securely backup and restore their data in an untrusted peer. The backups are performed in an incremental way giving users the possibility to restore a later version of their file.

- **OpenNebula** [38] is a research project used to virtualize and manage heterogeneous datacenters. As a distributed system, OpenNebula is an open source management tool, which is vendor neutral and can combine existing virtualization technologies to provide automated provisioning, elasticity and multi-tenancy, which is an architecture where a single instance of software serves multiple clusters.

- **Cassandra** [15]is distributed storage systems used to manage a very large amount of users data distributed among many servers, while providing high availability with no single point of failure. It manages thousands of nodes spread in different data centers and maintains high availability through multi-dimensional map indexed by key. Cassandra is implemented by Facebook for maintaining the consistency of their users' data. The core distributed systems techniques used in Cassandra are partitioning, replication, failure handling, scaling and membership. Any read/write requests for a key are directed to any of the nodes in Cassandra, and then it is the node that determines which replica is responsible for that specific key.

- **Glacier** [39] is a peer-to-peer distributed storage system used to manage users backup data. It uses a combination of replication and erasure codes to ensure high availability and low-down storage usage. The core architecture of Glacier is composed of three layers: primary store, aggregation layer and application layer. The primary store layer is used to ensure efficient read and write access and to provide short-term availability of data by masking individual node failures. The aggregation layer aggregates small objects prior to their insertion into Glacier for efficiency. To provide data durability, when a large aggregate has accumulated, or a time limit is reached, Glacier erasure codes the aggregate and places the fragments at randomly selected storage nodes throughout the system. In order to protect data against Byzantine failures, Glacier uses application layer to renew leases for all objects they care about once per lease period.

- **PAST** [40] is a peer-to-peer distributed storage systems designed to provide immutable data build due to the fact that modified files cannot be written with the same file as its original. The PAST system is composed of nodes that can route users request to create or modify a file. The persistence and durability of files is ensured by replication. Each object stored in PAST is replaced multiple times and is geographically distributed. PAST distributed systems provide high availability of files by obliging every block that contains replica copy of each file to send heartbeat to a node responsible to monitor data availability and any time there is a failure of a specific node, the system automatically restores it with all its objects.

PAST provides a very efficient routing scheme that makes possible that clients are served from the closet location node increasing thought resilience of the system.

- **The Bayou** [41], [42] is a peer-to-peer distributed system designed at Xerox PARC to support data sharing among mobile users. The Bayou System is a platform of replicated, highly available and variable-consistency of mobile databases which build collaborative applications. In Bayou architecture every device plays the role of a client or a server. Every device that can hold an entire copy of one or more data items is called server, and clients are called the one who reads or writes these data items. The availability of the data is maintaining thought replication process and each data item are replicated in many servers. Any time a file is modified all copies of a database are converging towards the same state and will be the final state if there are no new updates using "anti-entropy" protocol[43].

- **Amazon S3** [44], [45] is one of the biggest distributed storage systems composed by a large number of data centers nodes distributed across multiple locations, which provides high data availability and data durability for their users. Amazon S3 architecture is composed of two-layer namespaces: buckets and data objects. At the top layer there are buckets which contain a unique global ID and servers for many purposes such as allowing users to organize their data, identifying users and their rights and preparing auditing reports. The user security is maintained through distributed hash tables (DTH). Each bucket can store an unlimited number of data. Each object is composed of a name, an opaque blob of data and metadata consisting of a small set of predefined entries and up to 4KB of user-specified name/value pairs.

## 2.5 Definitions

In order to be considered a good service, every system has to provide some parameters and functionalities that meet the customer requirements. These parameters are essential for the service provider and should not be lower than the agreed condition. The main

functionality related to costumer is systems availability. Even though what is seen from costumers is availability, developers have to optimize other parameters and functionalities such as scalability, data durability, redundancy, concurrent access and retrieval time.
In this section we provide a formal definition of all these parameters and functionalities, which we are going to use throughout this thesis.

- **Data availability:**[46], [47] is the notion used to describe that costumer data are available and accessible at a required level of performance as agreed between service provider and costumer. The level of performance should be kept from normal through disastrous period. Data availability in cloud storage is maintained through redundancy including where data are stored and how they can be retrieved.

- **Data Durability:**[48], [49], [50] is defined as the duration of time the system is able to provide access to its stored data, which means the ability of the systems to maintain data available even in disaster mode. It is very important to monitor properly and not ignore data durability during evolution process especially when the replicas in the system are susceptible to failures of a more permanent nature. It seems quite similar to the notion of availability but, while availability deals with accessibility when all replicas are non-operational, durability deals with permanent loss of the replicas.

- **Data Redundancy:** [51], [52] In distributed storages, data redundancy is a condition created when the same portion of data is stored in two separate places. The process can be understood as two different spots in multiple software environments or platforms. Anytime we have repetitive data, the process of data redundancy is needed. There exist two ways for performing data redundancy: in the first way the same piece of data is replicated in multiple copies and stored in different places. If one or more copies are lost due to storage failure, the other copies are used. This method of data redundancy is very simple to implement but requires a lot of storage. The second type of data redundancy refers to the erasure codes technique. A positive type of data redundancy works to safeguard and promote data, which is the technique where one file is divided in exact numbers of pieces and each piece is stored in different locations. These pieces should be equipped with redundant information to make possible restoration of original file if one or more pieces get lost due to failure or storage outages.

- **Concurrent Access and retrieval Time**: [19], [46], [53], [54]. Concurrency is a property of distributed systems and represents the fact that multiple events are occurring at the same time. According to [55], each distributed system may have several independent processes, each of them being executed on its own without interaction with one another. In addition, these processes may perform some kind of interaction among them. Assuming that each of the processes wishes to write the shared data, or one wishes to read and the other to write, may result in wrong result. To avoid wrong results, it is needed to provide mechanisms to control the different flows of execution via coordination and synchronization, while ensuring consistency[56].

  Another aspect is related with retrieval time [57], [58], which is defined as the elapsed time from the point at which a costumer or user requests to access his data to the point at which the distributed storage system can reply the users request with requested data. The retrieval time is the most important parameter that can define distrusted system quality. Retrieval time involves all other parameters such as availability, consistency and durability.

# CHAPTER 3

# DISTRIBUTED DATA MANAGEMENT IN CLOUD COMPUTING

## 3.1 Distributed Lock Managers

In distributed storage systems, lock managers [14] are techniques which maintain file consistency and avoid simultaneous access to the same file. Distributed lock managers run an instance on every node and are used to provide nodes in distributed systems to cooperate and synchronize access in shared resources. It is important that every node run an instance of the lock manager to effectively coordinate and synchronize access in shared data. The main functionalities provided by lock managers are to: provide mutual exclusion, notify nodes holding a lock for specific data so that the same data is requested by another node, return information about locks, etc. Lock management in distributed storage systems is applied according to two main approaches: centralized lock management and distributed lock management [59].

In centralized lock management [60] there is e single node designated to control and manage all established transaction and locking table mode for all shared resources, acting as a coordinator for all other nodes of the distributed systems. In addition, a unique node maintains locking information and schedules lock management for all other nodes. All requests for accessing shared resources are directed to that node, and then it is the node which decides the priority and the time slots that requests have. In centralized management, concurrent access of the shared resources is resolved, and shared resources are free from errors. However, existence of a single node designated to control and manage all transactions is a single point of failure and becomes a bottleneck for locking requests, which is the main disadvantage of this method.

In distributed lock management every node is equipped and runs an instance of lock manager. Lock manager instance of the node is responsible for managing and maintaining lock status of each of the files stored. When files are replicated in multiple copies to increase redundancy, if a request is attempting to perform a read or a write operation, all other copies have to be exclusively locked before the files are updated. Distributed lock management is a method that improves the availability and scalability, resolves the bottleneck issue but increases communication overhead because each update on a file has to be propagated in every other copy. Distributed lock management has to implement voting algorithm [61], [14], [54] and agree on the way how they will maintain consistency of a shared file, otherwise none of the nodes can perform update on the files.

## 3.2 State of art

In this section we are going to discuss the existing distributed lock managers` algorithms and their features, and in the discussion section, we will present a comparative analysis among existing algorithms and our algorithm. One solution for maintaining the consistency proposed by [62] consist in an algorithm in which the maintenance of the shared file consistency is done both locally or globally and depends on the type of the clients request and depending to the situation of the shared file. The lock manager algorithm consists of a local lock manager and global lock manger, which respectively handle the information processed from its own node and information of the entire lock space for the clustered shared lock. When any of the server nodes gets a request from a client, it gets a new local pointer in its own lock table. If the local pointer is equal to its own number it proceeded the request without any further information, otherwise it requests another lock server from the global lock table. If the returned lock server has failed, it requests alternative lock server from alternate cluster table. The global lock manager is fully distributed, and it is synchronized through all servers that are part of the same cluster.

To implement a solution which synchronizes lock management among servers in the same cluster or among different clusters, it is required to have a reliable connectivity among lock manager servers, used for exchanging messages between servers. To avoid network connectivity, the solution offered by [63] implements the distributed lock algorithm in the control plane of the cloud computing environment and from there can be accessed by distributed applications executing on resource instances in order to maintain locks on resources that are accessible by those distributed applications using different API calls. The lock management can be embodied in accordance with the type of the resources that are shared. For distributed systems that provide virtual computing services to clients, it implements a distributed lick manager which exposes an API to users and, anytime a component of distributed lock manager receives a request to access a shared resource, the component of the distributed lock management performs the lock operation and communicates with other lock components to share the lock state information through control plane network. In some other embodies the distributed lock managers can communicate over network for managing locks on the shared resources.

Another efficient lock manager algorithm that provides high consistency in distributed systems called Chubby [19] has been deployed. Chubby consists of two main components that communicate via remote procedure call protocol: a cell that is a set of servers also called replicas, and a library that client applications link against. Client Library maintains all the communications between client application and servers. The replicas use a distributed consensus protocol for electing the master that is responsible for reads and writes for the database, while replica copy database from master. After being elected, the master periodically sent updates to the replicas for retaining the position of being master.

The model gives a solution to the single point of failure but still requires a lot of bandwidth during the communication for maintaining the state of the replica servers.

In order to increase the efficiency of large-scale distributed systems, according to the approach offered by [64], the cloud is divided in three logical levels where the upper level stands for user interface and library and interacts with middle level. The middle layer is composed of servers that are referred to as nodes, every node has three layers: global index, local index and data chunks. Each node has local indexes for their own data and global index for keeping a set of shared local indexes by each node and is unique in all nodes. The global index is responsible for maintaining the consistency for read/write request of the data chunks stored in the cloud

Paxos consensus algorithm [6], [65] is another algorithm used to maintain distributed lock management in distributed system which is design in layers. The fault-tolerant replica log-based layer sits at the bottom of the protocol stuck and is responsible for maintaining a local copy of the log. Paxos algorithm ensures that that all replicas have the identical sequences of entries in their local log. Another layer of Paxos algorithm is fault-tolerant replicated database which includes a local copy of the database at each replica. At the top layer of Paxos algorithm stand Chubby which is responsible for managing and synchronizing the database. The basic concept of Paxos algorithm stands on the fact that it uses consensus to choose a replica to be coordinator. The coordinator selects a value and broadcasts it to all replicas. The replicas can accept or reject this message. Once the majority of replicas access the message the consensus is reached, and this replica is selected as coordinator. After being elected the coordinator is responsible for coordinating the tasks and maintain lock management. In case that the coordinator fails, the remaining replicas can elect again another coordinator offering high scalability and solving the problem of partitioning.

## 3.3 Voting Algorithm in Distributed Cloud Computing

In a distributed storage system, it is necessary to implement a set of rules and regulations which will fairly distribute the permission rights among participants. Voting [66] is defined as the procedure through which power is distributed among participants from the ballot box. In distributed systems voting refers to the way nodes communicate with each other to decide for acting or not upon a change occurred or requests made for a specific task [67]. For example, as described in [68] voting algorithm works as follows: A client sends a request to a voting node, the voting node informs all other voting nodes about the request. The voting nodes perform the necessary action to the request and send a reply to the client. The client has to wait for a minimum number of replies with the same result from voting nodes for taking or not permission for the requested task. Another approach where voting algorithm is used is to avoid mutual exclusion[69] in shared resources. Mutual exclusion is a mechanism to restrict the simultaneous access in shared resources. Based on the way implemented, voting algorithms can be static and dynamic voting algorithms.

In static voting [70] node votes once assigned remain unchanged for all the process and the nodes do not keep trace of the current state of the votes or of the system. In one of the earliest static voting algorithms proposed in [71] the votes were fixed and the distributed system has been expected to be fully synchronized with no failure node. A process has to wait for the majority of the votes from all nodes in order to enter in critical section. However, in static voting-based approach, if network partitioning occurs due to node failure, the system is not able to adopt new votes without interrupting system availability.

Contrary to static voting, in dynamic voting, nodes keep track of the state of other nodes. In case of node failures due to network partitioning, new votes are assigned to form at least a group of active nodes and keep the system active. One of the dynamic voting

Part of this chapter has been published in [70] which has received the best paper award approaches given in [72] provides two methods for reassigning votes after network partitioning occurs: group consensus and autonomous reassignment method. In the former method, the nodes of the majority group decide about resigning new votes either by distributed algorithm or by sectioning a node coordinator who will distribute votes to nodes. In autonomous reassignment method, each node takes its own decision for changing or not its vote. Nonetheless, before deciding for the final state of its vote, the node is required to take the majority of the total votes.

## 3.4 Data Replication and Concurrent Access Management

Data replication is a process used to maintain multiple copies of data or shared resources and to ensure consistency among redundant resources. The smallest replicable unit is an object. By definition, an object can vary from a field of data and a data table to a file and copy object stored in other locations, which are called replica. The remote locations where an object is stored are called nodes.

In distributed systems, there are different types of replication such as disk storage replication, database replication and file replication. In database replication technique multiple copies of the same database are created and there exists a master/slave relationship between main database and other copies. Any update on the main database is reflected on the other copies maintaining thought data consistency. To increase availability and avoid single point of failure, the multiple master approach is used. In this approach, the complexity of consistency maintenance is increased.

Storage replication is a process of replicating a block of data from a disk to another one in a different location. File-based replication is the process of logically replicating the files in multiple locations and maintaining synchronization among copies. Data replication is implemented to solve main issues in distributed systems related to availability, communication overhead, fault tolerance and scalability.

Two of the main problems with replication process are related with deciding Where and When [73] to perform replication. Where refers to the decision of what replicas will be updated and the parameter When refers to the time when to propagate updates to nodes. The strategies for selecting these two parameters mostly depend on the type of the connection among nodes, architecture of the system and the type of replication occurring such as synchronous or asynchronous replication.

In synchronized replication attempts to update all replicas in a single transaction, which means that all replicas are changed any time update synchronization happens, maintain consistency state between replicas. The major drawback with synchronous replication is related to fact that if one of the replicas fails, then the all transaction fails making this method not suitable to be implemented in distributed system.

In contrast to synchronous replication, asynchronous replication is the method that relies on optimistic consistency assumption [74] and non-optimistic consistency assumption [75]. Optimistic consistency, also known as eventual consistency, refers to the strategy of replication when replicas can diverge. Optimistic replication is a method implemented under the assumption that if no new update is done to a file then all accesses to that file will return to the last update state and replicas are synchronized only when part of the system has been lost of a certain time. In the non-optimistic approaches, it is assumed that conflicts will occur, and a propagation strategy needs to be implemented for preventing update conflicts. The main disadvantage of asynchronous method is that update procedure is not stopped due to a failed replica. The failed replica is propagated at a later time.

Another problem to properly be addressed during data replication process is concurrent access. In single master method there are one master copy of data stored in a location with full permission and multiple copies in another location with read permission only. In single master method shared file consistency process is simplified since it is only master server updated, which then spreads updates to other replicas. However, this method is potential for failure due to the existence of single point of failure. In multiple master replicas, there are several masters possessing a master copy of data with permissions to perform updates on the shared data. Clients can issue request for accessing shared data from each of the servers. Each time one of the copies is updated other copies need to be updated requiring a high level of consistency level on master server to prevent concurrent access.

We can configure replication to adopt full or partial replications which mostly differ in the dynamics of the data in the network. In full replication, every object in master copy is replicated in a certain number of copies and each replica has the same updated copy that the master has. In this approach communication overhead is very high and makes it very difficult to maintain consistency in a distributed system due to bandwidth limitations. Another drawback is related with the space used to store each of the replication copies. To decrease bandwidth and space usage, replication method can be optimized using partial replication[76]. In partial replication, nodes replicate a subset of the entire data. The principles for choosing which subset of data to replicate vary mostly according to the type of distributed systems.

## 3.5 Cloud Computing Platforms

With the new developments cloud computing is finding implementation in many aspects of our life. In the new era of smartphones, people are always online and need fast, and easy access to their data wherever they are. Nowadays most business transactions are carried out through online platforms. The usage of social media is a new approach of information for most people and old traditional media resources have been adapted to this new approach. In the recent years, lots of research has been carried out to improve cloud computing services for satisfying the costumers’ needs. There exist different definitions of cloud computing. John McCarthy in the 1960s was one of the first who foresaw that computing facilities could be provided to common users as a utility [77]. However, the term cloud computing was first used in 2006 by Google's CEO Eric Schmidt to describe the business model to provide services from internet. Since then the term cloud computing has become very popular and is used mostly as a marketing model to describe different business approaches and ideas. Many authors have tried to standardize the definition of cloud computing. For example, a work presented in [78] compared more than 20 definitions carried out from different authors, trying to standardize the term cloud computing. In this thesis, to define the term  cloud computing we refer to The National Institute of Standards and Technology (NIST) [79] which in our opinion gives the essential characteristics of cloud computing.

- **NIST definition of cloud computing** *Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*

The reason why exist different definitions of Cloud Computing is because cloud computing is not new technology, but it is a new model to carry out the operation of bringing together different kinds of technologies to provide a new model to do business differently. The main technologies used by cloud computing are: Grid Computing, Utility Computing, Virtualization, and Autonomic Computing.

- **Grid Computing** [80]: Grid Computing is defined as a collection of computers or computing resources distributed among several places to reach the same objective. The grid computing can be understood as distributed systems with high workload that involves a large number of files. Grid Computing are forms of distributed systems composed of various geographically distributed nodes acting as cluster and providing a high-performance task.

- **Utility Computing** [81] is a service provisioning model used by cloud service providers to offer computing resources and infrastructure management to their costumers according to their needs. One of the main benefits of utility computing is that resources to costumers are offered in accordance with their need of usage and not in a flat way, which is a model that maximizes the efficiency of resource usage.

**Virtualization** [82] Virtualization is the way  to create virtual resources such as desktops, servers, storage, operation systems or network while partitioning physical computer and storage resources. The main advantage of Virtual resources is transforming old computing model to prepare scalable computing resources that efficiently use hardware resources

- **Autonomic Computing** [83] Autonomic Computing is a self-management model which  is implemented using characteristics of the  human autonomic nervous system. Autonomic Computing is designed to make decisions for adapting to changes and performing tasks such as Self-configuration, Self-optimization, Self-healing and Self-protection using high-level policies. Autonomic Computing constantly monitors and optimizes its status to automatically adapt itself in case of condition changes.

## 3.6 Cloud Computing Architecture

Cloud computing is composed of two main sections: front end and back end and the two of the more network components connected through the network. The front end is desktop user computers and application used from users to see and interact with the cloud, while back end is cloud systems which includes servers, storage, network and various computers. The architecture of cloud computing is divided in layers in order to make it simpler for developers to add new features to one of the layers without effecting functionalities of other layers. Each layer of cloud computing architecture is strongly attached to layers standing below and above it. The main layers and services of cloud computing include Application Layer, Platform Layer, Infrastructure layer and Servers and Storage Layer.

- **Application Layer:** The application also referred to as "Software as a Service (SaaS)" layer, is the top layer providing high abstraction. It consists of cloud applications, which are different from other traditional applications in that they offer business applications, web services and multimedia which perform scaling features to achieve better performance and high availability. As an example of services offered in Application Layer we can mention Google App [84], Facebook, YouTube, Salefroce.com, etc.

- **Platform Layer** "Platform as a Service (PaaS)" layer is configured on top of infrastructure layer and is used to build operation systems and application networks. Platform layer is used to build platforms used for user authentication, web servers, Application software configure network access, right management, different API, etc.

- **Infrastructure Management Layer** Infrastructure management layer, also referred to as "Infrastructure as a Service (IaaS), is responsible for the virtualization process of physical resources. They use virtualization technologies and create virtual resources such as desktops, servers, storage, operation systems or network while partitioning physical computer and storage resources. Main virtualization technologies in use are s Xen [85], KVM [86] and VMware [87]. The main service provider of IaaS includes Amazon EC2 [88], GoGrid [89] and Flexiscale [90].

- **Server and Storage Layer** is the lowest layer where the level of abstraction is low and comprises physical resources which include servers, storages, switches and communication ways among them. In server and storage layer we can virtualize storages, perform monitoring, manage physical resources, and issue different upgrades. In most of the literatures servers as a layer have not been studied as a separate layer but have been attached to Infrastructure as a Service Layer.

According to their usage each of the layers mentioned above is a standalone solution and is a business model which provides services to customers. Somehow cloud layers are like OSI model and each below layer offers services to the upper layer. It is not mandatory for a service provider whose aim is to build Software as a Service (SaaS) to also build a Platform as a Service (PaaS); however, they may decide to rent or buy from another provider and can later develop their own service. Here the most important cloud platform has been mentioned. There also exist other platforms which are subsets of these, developed to further satisfy business needs.

Based on their characteristics cloud computing can be classified as: Public Cloud, Private cloud, Hybrid cloud and Virtual private cloud. Public cloud is distinguishing set of cloud services that are offered to the public. Users pay for their usage and no first investment is required. However, when using public cloud users can have lack of weekly control over their data, network and security.

Private cloud is clouds designed and developed by private businesses to fulfill their business needs. They can be designed and managed by a single organization or by external contractors. Private cloud resolves most issues related to public cloud providing high performance, data security and high data availability.

Hybrid Cloud is a mixture of private cloud and public cloud, which try to complement each other. Hybrid clouds try to adopt main features from both types of cloud running infrastructure services in private in order to provide high availability and security and run other parts in public cloud.

Virtual private clouds are a way to eliminate the limitations that public and private cloud have by offering a cloud service which is developed over public cloud and providing virtual private network service thus offering service providers the option to design and implement their own security policies.

## 3.7 Cloud Storage Architecture

One of the most important layers of Cloud Computing is Storage as a Service Layer. This is the layer that combines and manipulates physical resources to provide different services which are later used by other layers. One of the services offered by Infrastructure layer is storage services. On account of its importance, developers have separated and developed it as a separate layer. Cloud storage is defined as a service model used to maintain and manage client's data over network. Cloud storage as a service must ensure reliability and availability of client's data and provide high consistency. However, users have to pay the cloud services a monthly fee based on consumption and are not charged further operating systems fee for building other security tools. Most of storage services implement security policies, such as encryption and authentication in order to enhance security of their services and to ensure their costumers the highest security measures to restrict untrusted access to stored data of their clients.

In addition to cloud computing, there are three main cloud-based storage architecture models: public cloud-based storage, private cloud-based storage and hybrid cloud-based storage. Public cloud-based architecture is a storage platform that is offered to public based on a subscription agreement to store their data. Public cloud storage stores data in an unstructured manner and is suitable for users and businesses that cannot afford to purchase dedicated services. In public cloud storages data are stored in big data centers that are spread in different locations. The most important public cloud storage services that dominate the market are Amazon Simple Storage Service (S3)[45], Google Cloud Storage [84]and Microsoft Azure [91].

Private cloud storages, also called cloud storage on premises, are cloud storages developed by different organizations which provide dedicated services. Most times private clouds provide customized application which allows users to have more control over their data. Normally private cloud gives their services to a secure environment and their external access is secured by firewall and other security policies.

Hybrid cloud storage is a composition of private and public clouds which interact with each other using interconnections managed by cloud programming technology. The interconnection is maintained infrastructure layer by defining specific rules and policies. Hybrid cloud storages offer flexibility and more data development options to their customers. For example, organizations can choose to store structures and data used actively in a private cloud and store unstructured and archival data in a public cloud. Based on benefit offers, in recent years many organizations have adopted hybrid cloud services. However, besides the benefits, cloud storage requires high technical implementation and management professional expertise.

Cloud storage architecture is based on infrastructure virtualization to delivery storage in demand and provides high scalability in a multi-tenancy way. Multi-tenancy is a single instance server designed to serve multiple costumers. Cloud storage architecture consists of two main components: front end and back end component. Front end component refers to an API used to access the storage. In the traditional convention API was referred to SCASI protocol. In the recent years new forms of front end components have been developed.  Some main front ends component examples are web service front ends, file-based front ends, and Internet SCSI, or iSCSI protocol. The middleware layer is used to implement and perform various tasks such as replication and data-placement algorithm and to geographically distribute data. Finally, back end component is responsible for implementing physical storage data.

## 3.8 Existing Platforms

- One of the biggest web-based cloud computing service platforms is Amazon Web Services (AWS) [92]. AWS is composed of a collection of online cloud applications to provide cloud storage, different services and functionalities. The offered services are accessible from web using HTTP, REST and SAP protocols and enable customers to deploy different applications and services on-demand basis at reasonable prices. The most well-known centralized service Amazon Elastic Compute Cloud EC2 is used by users to initiate and manage servers instance in data centers using API or different tools and utilities. Amazon EC2 is built on top of Infrastructure layer and is core part of AWS providing the computing ability for costumers. Using Amazon EC2 instances, customers can initiate and build virtual machines which can be used to run different personal applications and after finishing each instance runs as a separate virtual machine. EC2 stores virtual machines instances in multiple locations. Each EC2 location is composed of Regions and Available Zones geographically distributed.

  Another feature used by Amazon Web Services is Amazon Simple Storage Service (Amazon S3) that is used to store virtual machines instance. Data in Amazon S3 are stored in form of objects. Multiple objects are then grouped in buckets. The size of each bucket can vary from 1 byte to 5 Gigabytes of data. Object names are sued as a path name for the virtual machine instance. Each bucket must be created prior to being used and can be stored in one of the Regions. To enhance security and allow customers to integrate their existing services with services deployed in amazon cloud, Amazon has deployed Amazon Virtual Private Cloud as a secure channel between amazon cloud and companies existing IT Infrastructure. This feature enables customers to create virtual private network connections between customers' premises and cloud and integrate their services with the cloud services.

- **Google Cloud Storage** [93] is an online service which allows users to store and access data through Google cloud platform by providing high scalability, security and sharing features. Like Amazon S3, Google Cloud storage is as an Infrastructure as a Service (IaaS) which allows users to use API to access their data. To enhance security, prior to building their infrastructure users are required to authenticate and agree with the terms of reference.

  According to its design data stored in Google storage are objects names that can vary up to 100 Terabytes. Multiple objects are grouped in buckets. Buckets are stored in multiple locations geographically distributed. Each object is identified within a bucket by a unique user-assigned key. The main features provided by Google cloud storage are Interoperability, Consistency, Access Control and Resumable Uploads.

  Interoperability is a feature that provides Google cloud storage to interoperate with other cloud storage tools and libraries. Google cloud platform provides high consistency for every read-after-write operation in uploaded files. Google cloud storage implements access control list to access and manage stored objects and buckets. To eliminate upload failures due to connection interruptions, Google cloud storage provides customers with a resumable data transfer feature to resume upload after connection establishment in the point of the interruption without having to start data transfer from the beginning. Design, enhanced security and simplicity to deploy new services make Google storage nowadays preferable for many costumers.

- **Microsoft Windows Azure Storage** [94] is a scalable cloud storage used exclusively by  Microsoft as a search engine. Due to the number of services and its flexible design, in recent years WAS has become very popular with thousands of users who have started using it to store their data and to migrate their services.

Cloud storages in WAS are provided in three different forms such as file or blobs, table and queuing. To comply with user needs, windows azure storage has deployed features that include high consistency, global and scalable namespace, disaster recovery and multi-tenancy, and Cost of Storage.

Windows azure storage maintains strong consistency providing high availability and partition tolerance that comes due to the implementation of a specific fault tolerant model which divides management in different layers. Stream layer is an append-only data model and is responsible for providing high availability in case of network partitioning or other failures. Consistency is provided by partition layer which is built upon stream layer. Partition layer enables the separation of nodes responsible for consistency from nodes used for storing data. In case of network partitioning partition layer assigns the partitioned server to partitioned servers in available racks. It is this separation and redirection of a specific collection of faults that allow the system to provide strong consistency and high availability in case of failures.

Windows erasure storage ensures a single global and scalable namespace that allows clients to configure their cloud storage in a regular manner and scale its storage space according to its needs. They control storage namespace through DNS. Each namespace contains three parts: an account name, a partition name and an object name.

Disaster Recovery: Windows azure storage stores data in multiple locations and uses the replication technique to protect client's data against disasters failures. The locations are chosen to be far away from each other so as to avoid being affected in case disaster happens in a region. Multi-tenancy and Cost storage is another feature implemented by windows azure storage which permits customers to use the same shared storage infrastructure.

- **OpenStack**[95] is a private storage platform, deployed as a joint project by Rackspace Hosting and NASA to provide users with facilities to design cloud Infrastructure as Service platforms. Design architecture of OpenStack is composed by couple modules such as user management module, file management module, and resource management module.

  User management module maintains user authentication process and login-logout sessions. Prior to using it every user has to fill in all the necessary information to register and after successful activation, users can insert personal information. Each platform has its own manager that is responsible for assigning roles to users and deleting both frozen setups and restore operations for users. This is a module that mostly deals with users` validity to access the platform and their rights assignments in other modules.

  File management module provides options where users can upload and download, as well as delete files and folders. Users have no restriction on the type of the file to upload. They can upload any kind of file such as video, music, images, documents and folders. Download option maintains validity of users to access and download the required data. File deletion and management options are used to allow authorized users to delete certain data as well as classify and organize data inside the module.

## 3.9 Discussions

In chapter 2 and in chapter 3 we have given an overview of the main existing distributing systems and cloud platforms and their characteristics. According to CAP theorem [96], [97], it is impossible for a distributed system to guarantee consistency availability and Partition tolerance at the same time.

From our discussions we have seen that developers need to decide on important parameters to be optimized for a sustainable solution. Some solutions such as BigTable [98], HBase [99] and MongoDB [100] are all CP systems that achieve strong consistency and Partition tolerance (CP) by losing the ability to ensure availability. In such systems requests are likely to fail due to node failure or other forms of failure.

Other models such as Cassandra [15], and Dynamo [21] provide a solution to maintain high availability and Partition Tolerant (AP) by not ensuring strong consistency. These models are solutions that use the replication technique to ensure data reliability and due to network partition outdated replicas that may disrupt file consistency might also occur.

Consistency and availability (CA) are the most difficult to be maintained and can apparently be achieved only if there exist no partitions of the network. Most of the systems fail to maintain high availability with strong consistency.

To overcome these obstacles developers has been deploying two main models: centralized storage model and distributed storage model. Both models have their strengths and weaknesses. Existence of a single node to maintain all transactions between storage nodes and clients raises the risk of being single point of failure and posing bottleneck problems. This is a solution that provides strong consistency at the cost of availability. To improve availability models, [19] provides a set of servers which act as masters and have the same right in storage nodes. In this model one of the nodes is selected as the master and other nodes are replica nodes. In case of master failure, one of the replicas takes up the duty of master. The main drawbacks of these storage models are the difficulty to maintain consistency and the network overload due to the large number of messages exchanged among master and replicas to maintain consistency.

Distributed storage model [8], [37] gives a sustainable solution for high availability in distributed systems; every node acts as master and in case of node failure or network partitioning access to shared resources can continue from other nodes. This model eliminates the need for a permanent master node which will perform transactions between client requests and storage nodes. The main concern with this storage model is related to maintaining consistency. For example, the same resource may be requested by more than one client from different nodes, and nodes need to coordinate among each other to avoid simultaneous access. To solve these issues developers, deploy voting algorithm, where nodes require a majority of votes from other nodes to perform a specific task or access a shared resource.

Another aspect to be optimized is the network partitioning problem which requires implementation of algorithm to coordinate and reestablish priorities and duties among remaining nodes after partitioning. Some models use static voting algorithm to maintain consistency in such cases, while others propose dynamic voting algorithm as a proper solution.

Cloud computing is a new kind of distributed systems development and is oriented toward business; it tries to take and implement the best practices from each model in order to offer the costumers a service which provides high availability, strong consistency and free from network partitioning. From user's side cloud computing is like client-server architecture. There is a gateway which is used by clients to connect to cloud and perform their tasks. In the back-end side is not seen by customers and it is composed of thousands of nodes which collaborate to provide service to the customers. To maintain availability and consistency on the shared resources most of the cloud's solutions integrate the concept of distributed model discussed here.

File consistency, availability and scalability play the biggest role in defining the quality of a cloud computing. The model we are offering to discuss herein covers most of the problematics that existing models had. Acting as a centralized model and inhering the features of distributed models, our algorithm reduces the number of the messages to file consistency in the shared files, which is one of the main concerns, while offering high scalability. Our algorithm is resilient to partitioning and eliminates lock inconsistency when simultaneous access happens, by following a strict procedure for maintaining distributed file consistency.  In our model exist only one lock manager per server and the decisions are taken independently from other lock manger servers. There exists no master node server, and each server has equal rights.

## 3.10 Methodology of Research and Further Steps

This study includes both qualitative and quantitative research methods and techniques. Qualitative methods include observation, analysis, design and implementation of our proposal. Quantitative methods include the analysis of data collected from different tests held with our application.

Observation was conducted in the first phase of the study. The design of existing models, which seems to make a clear separation between the two main categories and to have a uniformity of the way to maintain concurrent access, particularly led to the idea the researcher has proposed and implemented.

The literature review was grouped in two categories: theoretical studies and the ones relevant to the present field of study. The literature reviewed played a big role in denoting the challenges that distributed storage face and in the preparation of the proposal design. This research has been particularly influenced by the work of Kishida, Hajime, Yamazaki, and Haruaki [14]. Even though their study is a pure centralized architecture, it inspired us to use all its conceptual ideas and design a fully distributed model.

Prior to starting with the design of our model, a detailed analysis of all components of the study was performed by drawing diagrams of all possible scenarios and communication ways between servers. We used Unified Modeling Language (UML) tool to prepare our diagrams which will be used in the coming chapters.

Another aspect of the qualitative methods implemented in our study is defining components and designing a full scheme including all components and relations among them. In our design we denote two types of components: global components which communicate with their respective instances located in other servers and local components which have only local duties. The implementation of our project is done by using programing language Java.

The last used is a quantitative method and it consists in the empirical analysis of the data collected from different tests. We have conducted several tests with the algorithm setting configured with different preset resources. We have collected three types of data that differ from each other in the preset resources that we define in our algorithm.

The first type of performed tests is held with the algorithm configured to work from centralized-based architecture to distributed-based architecture with the same fixed preset resources. We use these data to perform a comparative analysis of the performance of both architectures.

The second type of data are collected from tests performed in distributed based architecture and our aim is to analyze and measure how performance of our model is affected based on the amount of resources used.

The third type of data collected is related with one behavior of our model that has to do with resource starvation. We performed tests by pushing one of the master nodes to enter in starvation mode, which refers to the situation when the server has no more resources to serve to further user requests and analyzed master server's behavior.

The evaluation and analysis of the results of the tests were carried out by the use of Microsoft Office Excel program. Since the tests are performed for different natures of the problem the evaluation is done in separate ways. According to its implementation, our algorithm gives detailed information about every movement and procedure executed from the moment a request is received till the end of its execution. Since our focus was the time added during the process of exchanging messages between master nodes to eliminate concurrent access in shared files, we have filtered the data and then manipulated it. The result outcomes are organized in charts, which have been presented in this thesis.

# CHAPTER 4

# ADLMCC – ASYMMETRIC DISTRIBUTED LOCK MANAGEMENT IN CLOUD COMPUTING

## 4.1 Background

With regard to the algorithm that we are going to discuss, we consider a code with MDS property and discuss the basic file operations in practical approaches. We design conditions that will prevent both read-write and write-write from simultaneously executing at the same file [101]. When we apply Maximum Distance Separable (MDS) [13] code to store data files, initially each file is divided into k parts that will be called chunks. Through the application of linear combination encoding, the k chunks will be altered in n code chunks, where $n > k$. Now the n code chunks are stored at N nodes. Based on how MDS operates, to reconstruct the whole file again, it is necessary to contact k nodes. The maximum number of failures that can be tolerated is $n - k$. Each node stores a certain number of chunks that will be referred to as $\alpha$ chunk and considering a cloud with n nodes they make $n\alpha$ chunks in total.

To construct a new node which can replace failed or corrupted ones, it is necessary to contact any d nodes and download $\beta$ out of $\alpha$ packet from each node. The operations that can be executed on the file are three:

1) Read, download or reconstruct the entire file - that is the case when a lot of users would perform these requests. This is the case that requires the generation of the complete file.

The main contribution of this chapter appears in the papers [101], [102], [103].

2) Write, update - Any change in the initial file requires a change of all the corresponding data stored in nodes. This is also a normal behavior that happens frequently when lots of users try to perform simultaneous write requests, which brings the update of the information stored in the nodes.

3) Node Repair, which happens when any node fails or gets corrupted. In this case, it is required to generate a new similar node. To do so, connection to d different nodes and download of β packets from each of them is necessary.

In our proposed model, a request for a file can be received from each of the servers that are part of the cloud Computing. According to the situation of files, the lock manager of the server can directly execute the request without interacting with other lock managers or must require collaborating with other lock managers for maintaining the consistency of the file.

When one of the servers, part of the cloud computing receives any requests from end-users, it immediately initiates the procedure to acquire the right to access the file. The user requests can be categorized into the following scenarios. Everything happens in a smooth way and requests coming from different servers do not interfere with each other. In a real-world, there will continuously be two or more requests from the same server and from different servers that will make simultaneous requests. Most of the time, these simultaneous requests might result in some problems. It will result in file inconsistency when a request is reading from a file and a write operation tries to write it at the same time.

A solution to the inconsistency of the file is to put incoming write operation in queue and to wait until read operation is completed. Following this approach, in case that another read request comes, it will also be placed in the waiting queue after write operation. This will create a lot of polling which is a waste of resources since two reads can occur at the same time. In real world, there will be many requests trying to read or write many nodes and in case these requests are not properly controlled and managed, the entire network might get congested and would take a lot of time to process users' requests.

Based on what was discussed above, the models offered are built upon the master slave model where there is a master to whom all requests are designated, and it  distributes the load among other slaves. Another approach is to have a set of masters who collaborate by choosing one of the servers as the master and the other servers act as replica for that. For any failure that can happen to the master there is one of the replicas that will be elected as master. Based on these models there will be a continuous need to have communication and periodic updates for maintaining the state and consistency of the files in the cloud.

In the next section we describe the proposed algorithm for distributed lock management in cloud computing and its features. Prior to that, we define some expulsion situation that is taken for granted in all our analyses. The main concern is to define an approach which will reduce the number of inconsistencies and give the result of the request within the least possible time. There are six possibilities for simultaneous request cases. Despite the total number of simultaneous requests, they will be composed of two pairs of requests only:
Write-write, read-write, write-repair, read-read, read-repair, repair-repair

According to the type of operation request by the user, requests that can be simultaneously executed in a certain file are:

- read-read - Yes. Two simultaneous reads can happen without a problem as they do not create inconsistency.
- read-write - No. Since write operation updates the file, it will lead to inconsistency in read operation.
- read-repair – Yes. According to the fact that repair is another kind of a read operation.
- write-write- No. Simultaneous writes will create inconsistency. Both will try to simultaneously update the file giving incorrect results in the end.

- write-repair- No. This is same as read-write case.
- repair-repair- Yes. This is same as read-read case

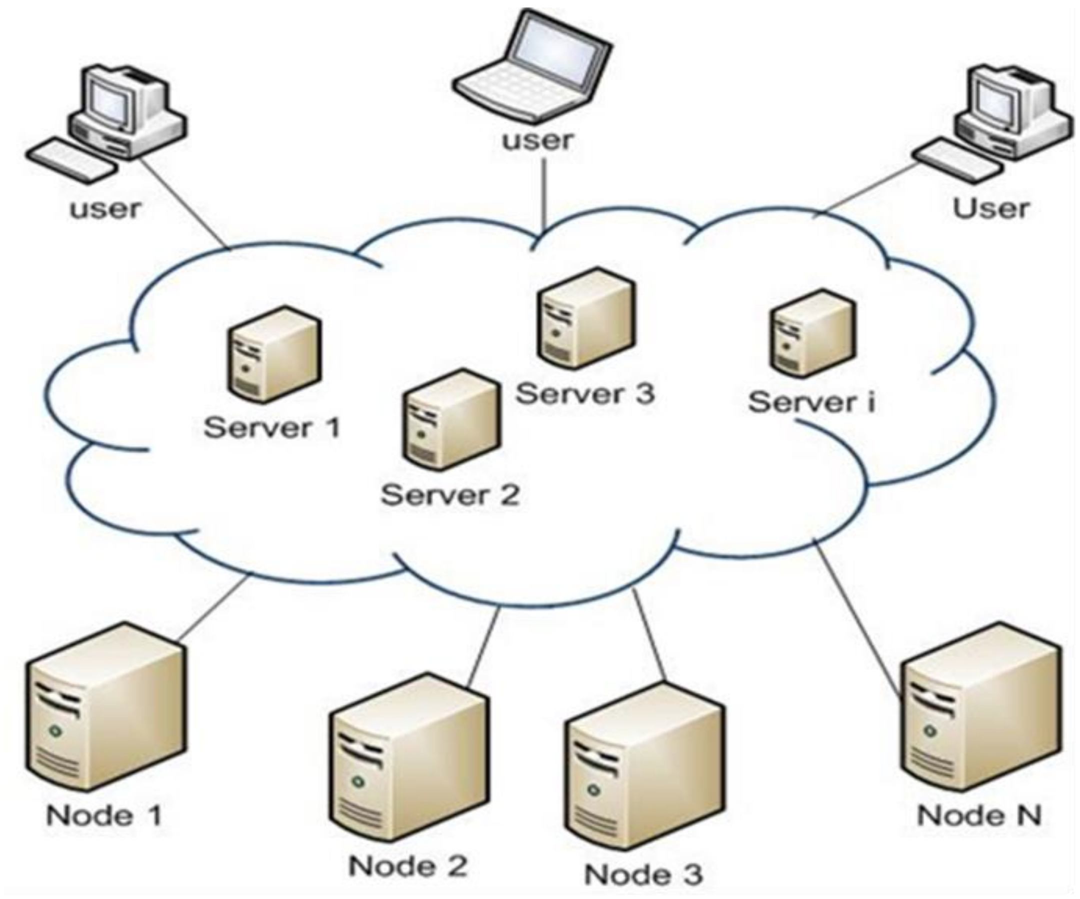


***Figure 1.*** Cloud Computing Architecture

For maintaining the consistency of the file, a unique lock-bit of one bit is denoted in every chunk stored in the distributed nodes. Lock-bit is a one-bit value describing the lock applied to a specific chunk, read lock or write lock. Lock bit 0 is representing the case when no lock is applied to the chunk.

As described in Fig. 1, our diagram has three components: users that use the cloud and deliver read/write requests to servers, servers that implement a solution based on the type of request and nodes that store the chunks that are shared with every server. All servers have the same privileges on nodes. Whenever a server gets any request for accessing any of the files, it firstly follows the procedure described in section 4 to gain the authority to read/write to a proper file and after that, it contacts the nodes for executing that request.

## 4.2 Asymmetric Distributed Lock Management in Cloud Computing

### 4.2.1 ADLMCC Architecture

Asymmetric distributed lock management in cloud computing [102] is accountable for preventing data inconsistency when concurrent access for specific files stored on nodes happens. File concurrency can occur with requests issued to one of the servers as well as between requests issued form different servers.

The lock manager of one server collaborates with lock managers of other servers to control concurrent access on files. This process is done by using inter-process communication among them. For this reason, several lock statuses are kept in their internal lock control tables.

The lock manager runs as an individual process on every server and cooperates with other lock managers only for maintaining the inconsistency of the shared files. The purpose of the lock manager is to coordinate the work of servers in order to maintain concurrent file access when clients issue lock requests for files saved in the end nodes storages. Our aim is to control read and write lock consistency in the shared files. Only one lock manager is accountable for a file at a time across the cloud.

In the structure of the lock manager there is a database composed of six important key factors that maintain concurrency control in the self-management of shared files and in the condition where communication happens between servers in the cloud. The six key factors in the structure of ADLMCC are: Server Node Table (SNT), File Directory (FD), Requesting Lock Table (RLT), Migrate-out Table (Mout-T), Migrate-in Table (Min-T), and Locked File List (LFL).

The structure of ADLMCC is essential for guaranteeing file consistency across the cloud. Each lock manager maintains and updates two different sets of data structures: one is maintained locally and is not propagated to any of the other servers and the other is fully distributed among other servers.

Server Node Table (SNT) is responsible for memorizing the configurations of all servers in the cloud. The information memorized in SNT is servers ID, Servers status and switchover server used in the case of main server failures. According to the sensitivity of the data, switchover server can be one server, or we may decide to have more than one switchover. SNT table is fully synchronized among all other servers (Fig. 2).

```
SERVER_NODE_TABLE is a list of ServerNodeInfo
Class ServerNodeInfo consists of:
        serverId
        serverStatus
        switchOverServer
```

***Figure 2.*** Server Node Table Pseudocode

File Directory (FD) is the directory which determines the path of each stored file in nodes and it is also the server that has created the file and has its ownership. We agree that FD is a data structure with pairs of file names and server numbers and is fully synchronized in the cloud. File creation is related with the server from where the client initially created or uploaded it. When a file is added in the system, the owner server propagates its path to every other server and makes it known by everyone (Fig. 3).

```
FILE_DIRECTORY is a map, mapping all filePaths to their Server
// Insert a record for a new file and its owner server
function insertNewFile(filePath, ownerServer)
    input: filePath is the path of the file searching for
        ownerServer is the owner server of the new file
    FILE_DIRECTORY ← new (filePath, ownerServer)

// Find the owner server of a file
function findFileOwner (filePath)
    input: filePath is the path of the file searching for
    return FILE_DIRECTORY.get(filePath);
```

***Figure 3.*** File Directory Pseudocode

Migrate-in Table (Min-T) and Migrate-out Table (Mout-T) respectively maintain information about achieved permission for accessing specific files from another server and the list of files that their lock has migrated to other servers.

In Migrate-in Table (Min-T) is stored information about every file to which the server has requested access and has successfully taken it. Migrate-out Table (Mout-T) is used to keep track of all files which the server is the owner of and access to which has been granted to another server. Information in Migrate-in Table (Min-T) and Migrate-out Table (Mout-T) is locally maintained and is specific for each server (Fig. 4).

***MIGRATE_IN_TABLE*** is a map, mapping all *file paths* to the *migrate-in Server*

***MIGRATE_OUT_TABLE*** is a map, mapping all *file paths* to the *migrate-out-Server*

```
functionremoveServerFromMiT(filePath)
        input: filePath is the path of the file requested
If (LockManager.MIGRATE_IN_TABLE contains filePath ) then
        LockManager.MIGRATE_IN_TABLE← removefilePath;
functionremoveServerFromMoT(filePath)
        input: filePath is the path of the file requested
If (LockManager.MIGRATE_OUT_TABLE contains filePath ) then
        LockManager.MIGRATE_OUT_TABLE← remove filePath;
```

***Figure 4.*** Migrate Out and Migrate In Table Pseudocode

Request Lock Table (RLT) stores the list of all locks requested by one of the servers. The information kept in RLT consists of user requester ID, file name requested, lock mode and timestamp. The table is maintained locally and works for managing locks locally in the server. Each server has its own RLT table maintained locally, which manages contents and differs from the other servers (Fig. 5).

```
REQUEST_LOCK_TABLE is a list of LOCK_REQUEST of a SERVER
Class LOCK_REQUEST consists of:
        requesterId
        lockMode
        requestedFile
        startTimestamp
        grantedTimestamp
        finishTimestamp
```

***Figure 5.*** Request Lock Table Pseudocode

LFL contains all the necessary information about locks that a specific lock manager is managing and is responsible for. The set of attributes of LFL contains sets of requester's ID, requester server, file name, lock mode and a timestamp, and two lists for queuing lock requests: one for granted and the other for blocked locks (Fig.6).

```
LOCK_FILE_LIST is a list of LOCK_FILE_INFO
Class LOCK_FILE_INFO consists of:
        requesterId
        requestedFile
        lockMode
        timestamp
        Queue of Granted_Locks
        Queue of Blocked_Locks
```

***Figure 6.*** Lock File List Pseudocode

### 4.2.2 Lock Manager Algorithm

Lock managers are independent algorithms running on cloud with one manager per server. They run independently of each other and collaborate with one another for maintaining the consistency of the files in the cloud. In cloud there will be many client requests issued to different servers, asking for granting simultaneous access to the same file. These requests perform concurrent accesses to files and necessitate the intervention of the lock managers of servers to properly maintain the inconsistency of files. The lock manager maintains locally the consistency in files and only in case of necessity do lock managers cooperate with others for the lock acquitting to perform lock request operations. Figure 2 shows the example of three servers that communicate with each other for maintaining the concurrent access to the files stored under distributed storage nodes.

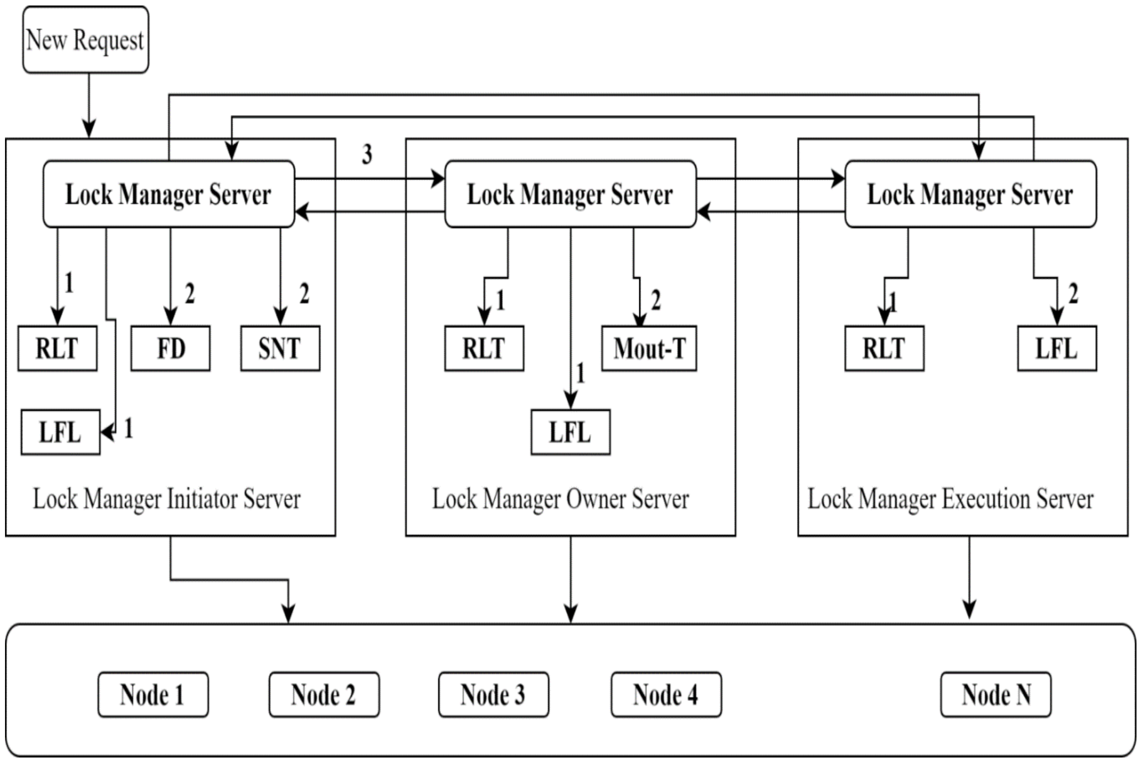


***Figure 7.*** ADLMCC Architecture

The architecture of ADLMCC has three main components: users which deliver requests for accessing different files, servers which serve as interface for users and maintain concurrent access to the shared files, and storage nodes or simply nodes that store data files. Lock manager runs in every server and each server contains all the 6 key factors that are essential for maintaining the file inconsistency. To simplify our design, in every server we have designed only the key factors that are used for this specific example. We refer to the server that receives the request as lock manager initiator server. As lock manager owner server, we refer to the server which created the file for the first time and which will be the owner of that file for all the period of time that this file will exist. The ownership of the file remains unchanged even when the file is being modified by requests delivered to servers, but which are not delivered to the lock manager owner server. The permissions of the specific file, which is requested from lock manager initiator server, might have been granted to another server and now the request has to be migrated to that server, which we will refer to as lock manager execution server. According to the state of the file requested and its attributes, lock management algorithm will follow the sequence from one to four procedures to archive the lock to a file and respond to the client as described in Fig. 2. The sequence from one to four procedures is as following:

- Self-management of shared locks in servers
- Finding a lock manager
- Checking a request migration
- Lock acquisition

### 4.2.3 Self-Management of Shared Locks in Servers

End-users deliver a request to one of the servers for accessing a specific file stored in the distributed storage nodes. After receiving the request, the server lock manager controls in case the file name requested has been requested by any earlier request already existing in RLT table. If any earlier request for that file is found in its RLT table, the lock manager inserts the request in the table of RLT associating it with the requester ID and its related operations for the requested file. The next step is to add the request to the LFL table for further procedures. After receiving the request, LFL starts checking the lock type already applied to the file for acquiring the right for read or write the file in the request. In case the file name is not found in any of the existing requests in RLT, the lock manager does not yet possess the necessary information about the state of the file and checks in FD to find the owner of the file and if he is the owner, he controls in Mout-T whether it has already given permission for executing the file.

After ensuring that the file is not found in Mout-T, the lock manager has sufficient information for the lock status on the file and adds the file for execution; otherwise the file is found in Mout-T. After getting the server, which is executing the file from Mout-T, the lock manager transfers the file for execution to that server and removes the file from its RLT.

Considering this scenario, when the initiator server is the owner of that specific file and the request either is found in RLT or is not inserted in Mout-T, the request is set to LFL table for execution. LFL receives the request and communicates with nodes to know the availability of the chunk stored among them. Each request issued from the server to the nodes has a unique request Id. Nodes will be available if there is no lock on them, and if there is a lock, then the availability will depend on the type of the lock. If nodes are available for a certain request, they will supply their adequate information to the server [53].

The information that they will provide consists of their lock status, node number and request Id. The server collects the information and checks in the lock table if that node can be granted that lock. If it can be granted, it updates its lock table with the corresponding lock as in the end user request id, and then updates lock bit on the node. It would not be necessary to update lock bit of the nodes whose request lock is the same as the recent lock because these requests will be read-read, read-repair or repair-repair and all these have the same vote bit [53].

The request is managed locally by the lock manager server, and no further communication among other lock manager servers is required. The activity diagram is explained in Fig. 9 and pseudo code is given in Fig.8. In case that the file name is not found in any of the earlier requests existing in RLT table and the server is not the owner, it is necessary to initially acquire execution right for the file by cooperating with the lock manager owner and other lock managers as described in section 4.2.4.

```
function manageLock (file, LockManagerServer, request)
input: file is the file which gets a request to be accessed
    LockManagerServer is the server which starts the request to access the file
    request is the started request
If (LockManagerServer is Owner) then
    Check RLT
    If (Owner is accessing the file) then
        LFL.Queue ← request;
        request.execute();
    Else
        Check in M-out T
        If (permission is granted) then
            RemoteServer ← the server found in M-out T;
            Move request to RemoteServer;
            RemoteLockManager.Queue ← request;
            request.execute();
            Return permission right to the Owner Server;
        Else
            request.execute();
```

***Figure 8.*** Self-Management of Shared Locks in Server's Pseudo Code

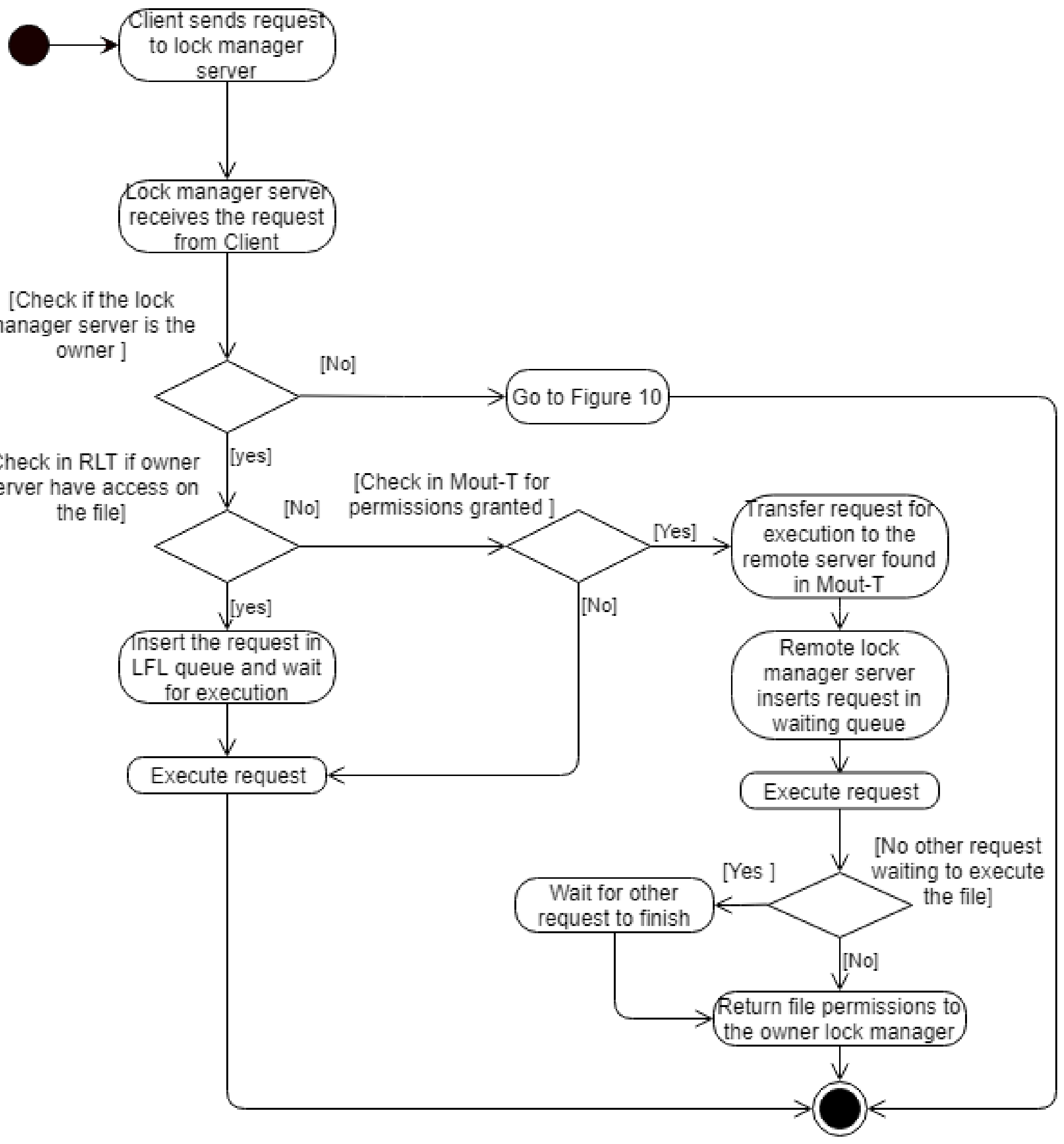

Client sends request to lock manager server
Lock manager server receives the request from Client
[Check if the lock manager server is the owner ]
[No]
Go to Figure 10
[yes]
[Check in RLT if owner server have access on the file]
[No]
[Check in Mout-T for permissions granted ]
[Yes]
Transfer request for execution to the remote server found in Mout-T
[yes]
[No]
Insert the request in LFL queue and wait for execution
Remote lock manager server inserts request in waiting queue
Execute request
Execute request
[No other request waiting to execute the file]
[Yes ]
Wait for other request to finish
[No]
Return file permissions to the owner lock manager

***Figure 9.*** Self-Management of Shared Locks in Servers Activity Diagram

### 4.2.4 Finding a Lock Manager

Following the explanation in section 4.2.1 lock manager is not able to decide itself for executing the file as requested in client request and requires additional information from other servers. The first process after controlling in RLT and Mout-T table is to find in FD table the lock manager server responsible for the file. After finding lock manager responsible for the file, lock manager server checks on its SNT table to discover the owner server and ensures it is alive and still in the cloud. If the Server exists alive in the Cloud, the lock manager on that server will be taken as responsible for the file. If not, the switchover Server in the table of SNT is engaged as the alternative node for the lock management. The lock manager at the lock initiator server sends a lock request message to the lock manager on the node found in the process above [53].

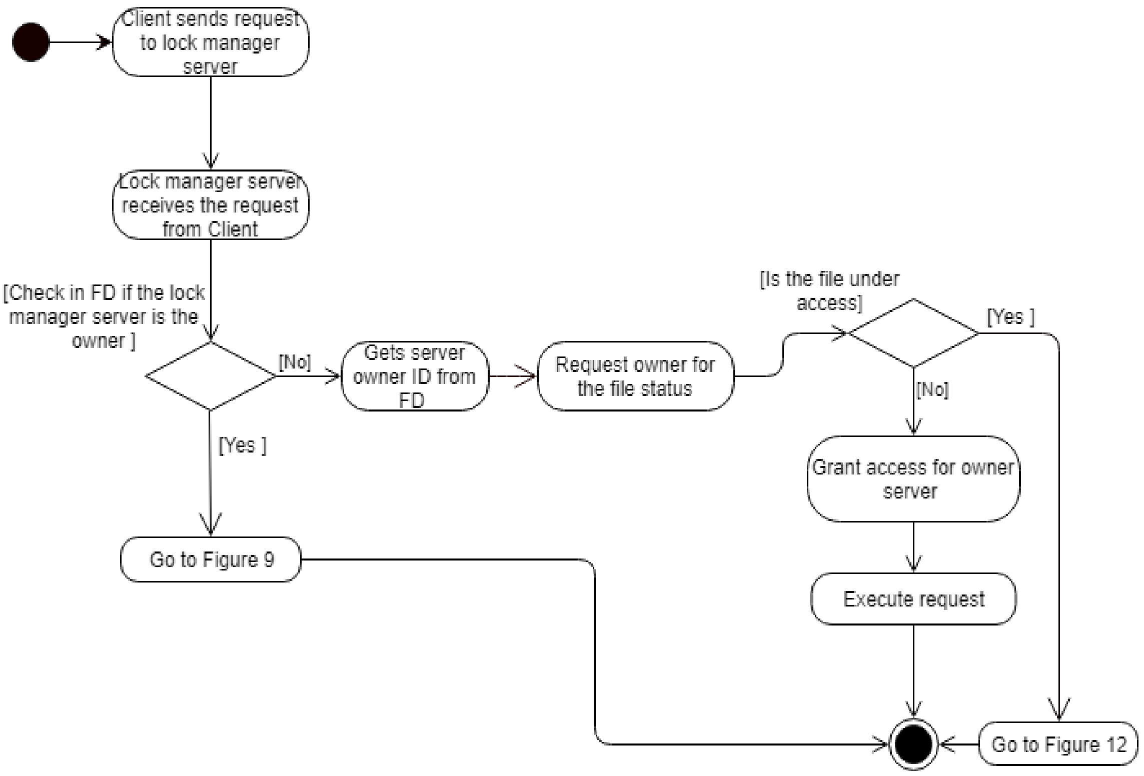

Client sends request to lock manager server
Lock manager server receives the request from Client
[Check in FD if the lock manager server is the owner ]
[No]
Gets server owner ID from FD
Request owner for the file status
[Is the file under access]
[Yes ]
[No]
[Yes ]
Grant access for owner server
Execute request
Go to Figure 9
Go to Figure 12

***Figure 10.*** Initiator Server Gets Execution Permissions from Owner Server Activity Diagram

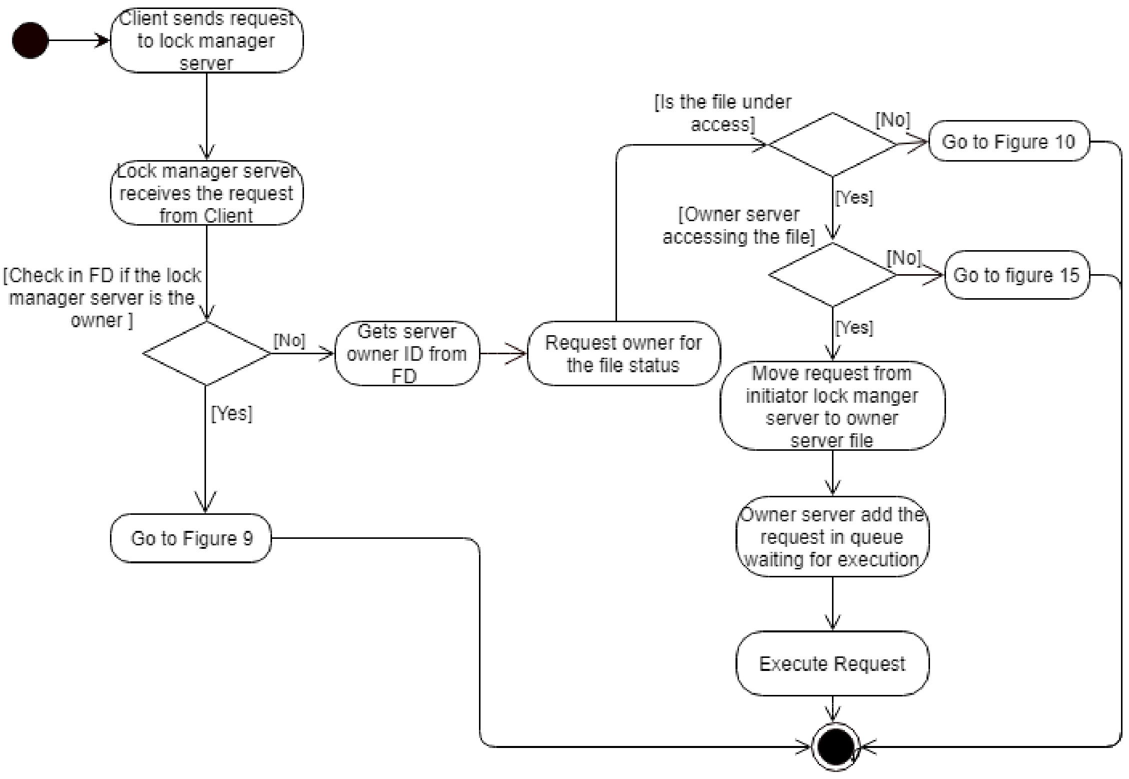
```
function manageLock (file, LockManagerServer, request)
input: file is the file which gets a request to be accessed
        LockManagerServer is the server which starts the request to access the file
        request is the started request
If (LockManagerServer is Owner) then
        …
Else
        Check if file is being accessed.
        If (file is accessed) then
                …
        Else
                request.execute();
```

***Figure 11.*** Initiator Server Gets Execution Permissions from Owner Server Pseudo Code

### 4.2.5 Checking Request Migration

The lock manager which receives a lock request message from the lock request initiator server is responsible for managing the lock on the file [53]. The manager checks in the table of RLT and on Mout-T if any lock is applied to the file or the lock management is migrated to a lock manager on another Server. In case the name of the file is not found in RLT, or in M-out T, the responsible server places the file in Mout-T and the lock management is migrated to the server initiator. Server initiator grants lock management for the file, as described in the relation diagram of Fig. 10, while the pseudo code is given in Fig 11.

In case the file name in the request is found in the RLT of the owner server, it means that the server has locked the file. The responsible server informs the server initiator and the request is migrated from the server initiator to the owner server and is processed from there. This activity is explained in detail in Fig. 12 and pseudo code is described in Fig 13.



Client sends request to lock manager server
Lock manager server receives the request from Client
[Check in FD if the lock manager server is the owner ]
[No]
[Yes]
Gets server owner ID from FD
Request owner for the file status
Go to Figure 9
[Is the file under access]
[No]
Go to Figure 10
[Yes]
[Owner server accessing the file]
[No]
Go to figure 15
[Yes]
Move request from initiator lock manger server to owner server file
Owner server add the request in queue waiting for execution
Execute Request

***Figure 12.*** Initiator Server Migrates Execution Permissions to Owner Server Activity Diagram

In case the name of the file in the request is found on Mout-T, the owner server replies to the initiator with the ID of the server that has granted access to that file and the initiator server with the information taken from the owner migrates the request to that server and the request is progressed from that specific server. By doing so now it is the remote server who will reply to the request and the server initiator removes the request from its queue and is ready to process another request. The initiator server does not keep any track of that request and does not get any feedback regarding its execution. Fig. 14 and Fig. 15 illustrate the pseudo code and the activity diagram that graphically explains this specific situation.

```
function manageLock (file, LockManagerServer, request)
input: file is the file which gets a request to be accessed
          LockManagerServer is the server which starts the request to access the file
          request is the started request
If (LockManagerServer is Owner) then
Else
          Check if file is being accessed.
          If (file is accessed) then
                    If (OwnerServer is accessing the file) then
                              Move request to owner server;
                              OwnerLockManager.Queue ← request;
                              request.execute();
          Else
```

***Figure 13.*** Initiator Server Migrates Execution Permissions to Owner Server Pseudocode

### 4.2.6 Lock Acquisition

The lock manager receives the request and updates its RLT. It also updates the LFL Table with the filename and its attributes as well as informs the server initiator that the request is added to the list for execution. Now it is the remote server who will reply to the request and the server initiator removes the request from its queue and is ready for processing another request. The initiator server does not keep any track of that request and does not receive any feedback whether the execution has been carried out or not. The initiator server removes the request from its RLT table. After the migration of the request from the server initiator, the server starts sending votes to nodes for r/w/rep the file in the request. There are two lists in LFL, which are the timeout time t_0 and a request queue. The request queue distributes the incoming requests in two slots, where all the requests of one slot are handled together and every request has its own priority.

```
function manageLock (file, LockManagerServer, request)
input: file is the file which gets a request to be accessed
        LockManagerServer is the server which starts the request to access the file
        request is the started request
If (LockManagerServer is Owner) then
        …
Else
        Check ownership of file
        Check if file is being accessed
        If (file is accessed) then
                If (OwnerServer is accessing the file) then
                        …
                Else
                        Get RemoteServer which is accessing the file, from
OwnerServer;
                        Move request to RemoteServer;
                        RemoteServerLockManager.Queue $\leftarrow$ request;
                        request.execute();
                        return persmissions to OwnerServer.
        Else
```

***Figure 14.*** Initiator Server Migrates Execution Permissions to Remote Server Pseudo Code

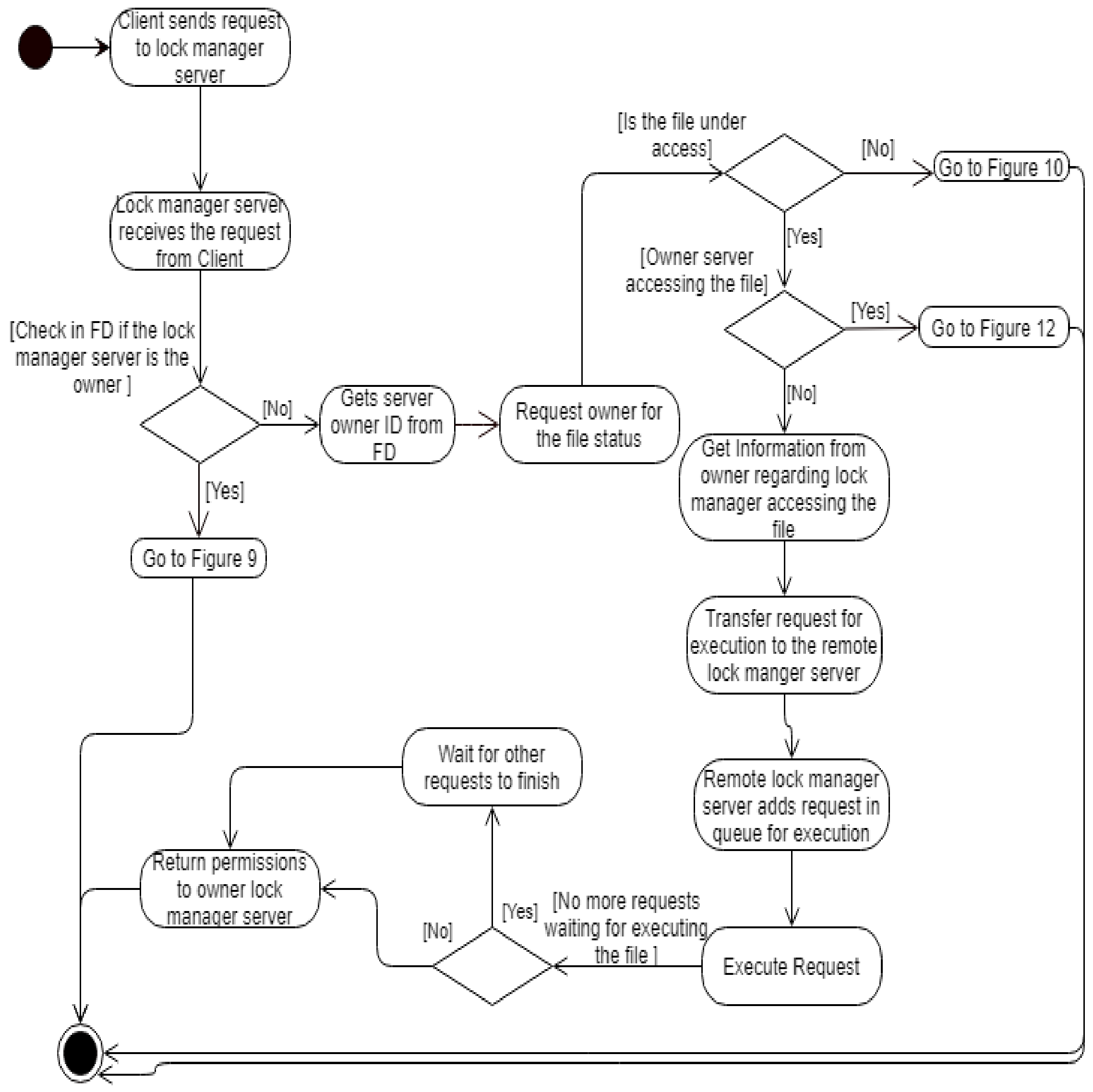

Client sends request to lock manager server
Lock manager server receives the request from Client
[Check in FD if the lock manager server is the owner ]
[No]
[Yes]
Go to Figure 9
Gets server owner ID from FD
Request owner for the file status
[Is the file under access]
[No]
Go to Figure 10
[Yes]
[Owner server accessing the file]
[Yes]
Go to Figure 12
[No]
Get Information from owner regarding lock manager accessing the file
Transfer request for execution to the remote lock manger server
Remote lock manager server adds request in queue for execution
Execute Request
[No more requests waiting for executing the file ]
[Yes]
[No]
Wait for other requests to finish
Return permissions to owner lock manager server

***Figure 15.*** Initiator Server Migrates Execution Permissions to A Remote Server Activity Diagram

## 4.3 Resource Starvation in Asymmetric Distributed Lock Management in Cloud Computing

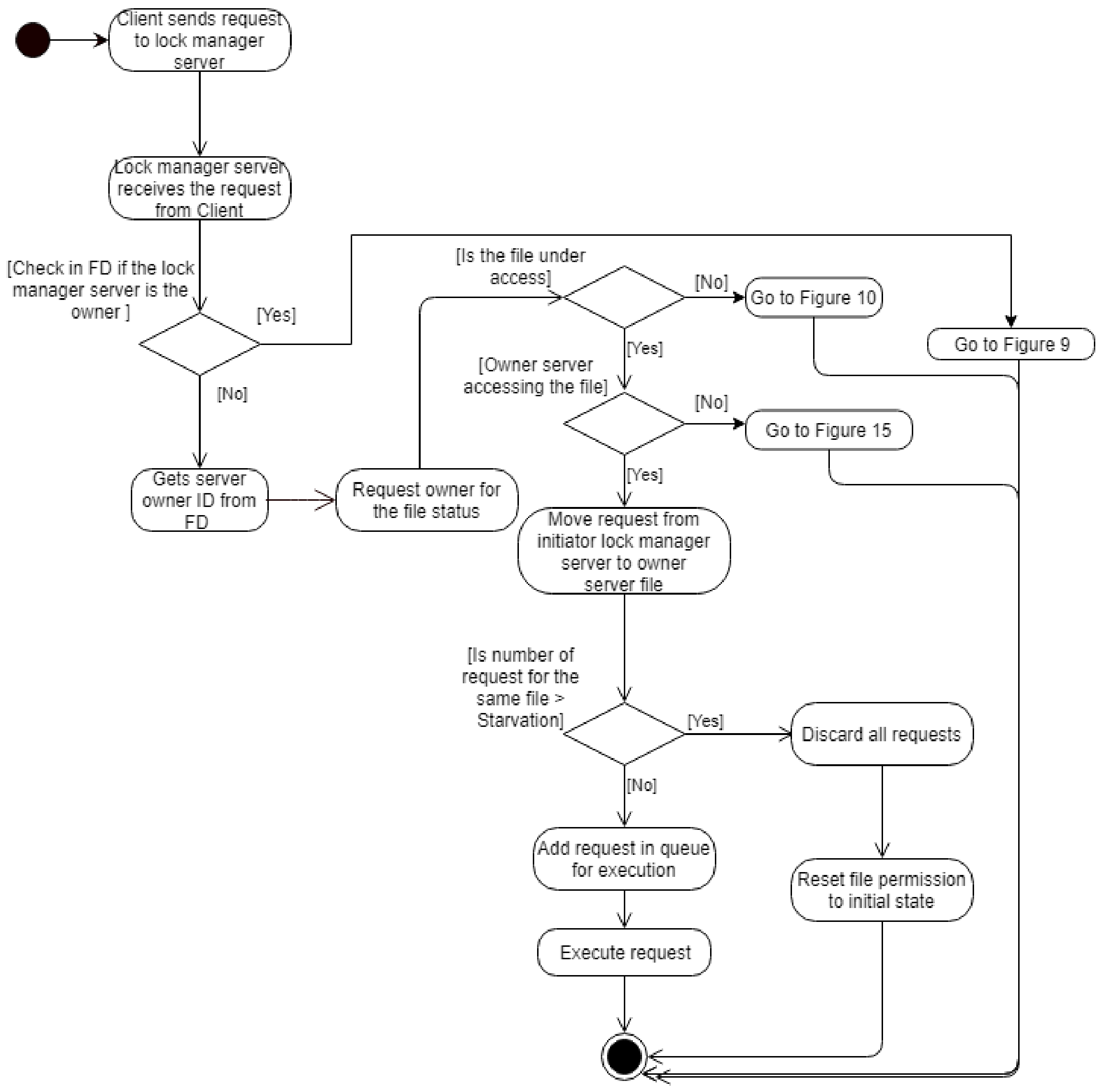
### 4.3.1 Checking a Request Migration

As explained in Section 4.2.5, in case the owner lock manager realizes that the request is found either in its RLT or in Mout-T, initiator server can no longer grant the right for accessing the file. In this point it has to collaborate for migrating the request execution to the owner or a third lock manager.

There exist two possibilities for the file: either the file is under execution from the owner or its right permissions have been migrated to a third server. Referring to the diagram in Fig. 16 and pseudo code in Fig. 17, N lock managers have the right to request the execution permission of a file but only the one who requested first will be granted it in a moment of time.

Assuming that the file is under owner execution, any time that one of the lock managers will request to execute the file needs to migrate the request for execution to the owner. This can happen perpetually, and the owner lock manager will get exhausted without any resource for serving later coming requests. In the literature of cloud computing this is called resource starvation and requires developer's attention for finding an appropriate equilibrium to supply users with the agreed service performance.

Coming back to our algorithm, to provide the required availability we define a parameter called Resources Starvation [103], which is responsible for obtaining cloud services running. When the requests in queue reach a certain number that is equal to starvation number, owner lock manager discards all requests and resets file permissions. In this mode all the requests need to be reinitiated for getting the permission of the file.



Client sends request to lock manager server
Lock manager server receives the request from Client
[Check in FD if the lock manager server is the owner ]
[Yes]
[No]
Gets server owner ID from FD
Request owner for the file status
[Is the file under access]
[No]
Go to Figure 10
[Yes]
[Owner server accessing the file]
[No]
Go to Figure 15
[Yes]
Go to Figure 9
Move request from initiator lock manager server to owner server file
[Is number of request for the same file > Starvation]
[Yes]
Discard all requests
[No]
Add request in queue for execution
Reset file permission to initial state
Execute request

***Figure 16.*** Initiator Server Migrates Execution Permissions to Owner Server Activity Diagram

```
function manageLock (file, LM1, request)
input: file is the file which gets a request to be accessed
    LM1 lock manager which receives a request to execute a file
    request is the started request
If (LockManagerServer is Owner) then
    …
Else
    Check ownership of file
    Check if file is being accessed
    If (file is accessed) then
        If (OwnerServer is accessing the file) then
            Move request from LM1 to OwnerServer;
            If (Nr of Requests > Starvation) then
                Discard all requests;
            Else
                Queue ← request;
                request.execute();

            Reset file permissions to initial state;
        Else
            …
    Else
```

***Figure 17.*** Initiator Server Migrates Execution Permissions to Owner Server Pseudo Code

### 4.3.2 Lock acquisition

For the same explanations provided in Section 4.3.1, when the file in the request is found in Mout-T of the owner and, when a new request has been delivered from any of the other servers, it should be migrated to the remote server which has already granted permission from the owner.

As shown in Fig. 18 and Fig. 19, there will be consistent requests from N-2 lock managers plus owner requesting to execute file. Within this state, we will have two main aspects that will affect the cloud service: one is the delay for one request to be migrated from the initiator to the executor server and the other is resource starvation happing by the same logic as described before. The new state of the system is illustrated in Fig. 18 and the pseudo code is given in Fig. 19.

Following the same assumption in every lock manager, we denote the starvation parameter and any time that the number of requests is equal to the starvation parameter all the requests in queue, beside the one that is under execution, will be dropped.

After finishing the execution of the last requests, the file permissions are reset to default, and the ownership of the file is given back to the owner. The rest of the functionalities of the lock manager remain unchanged.

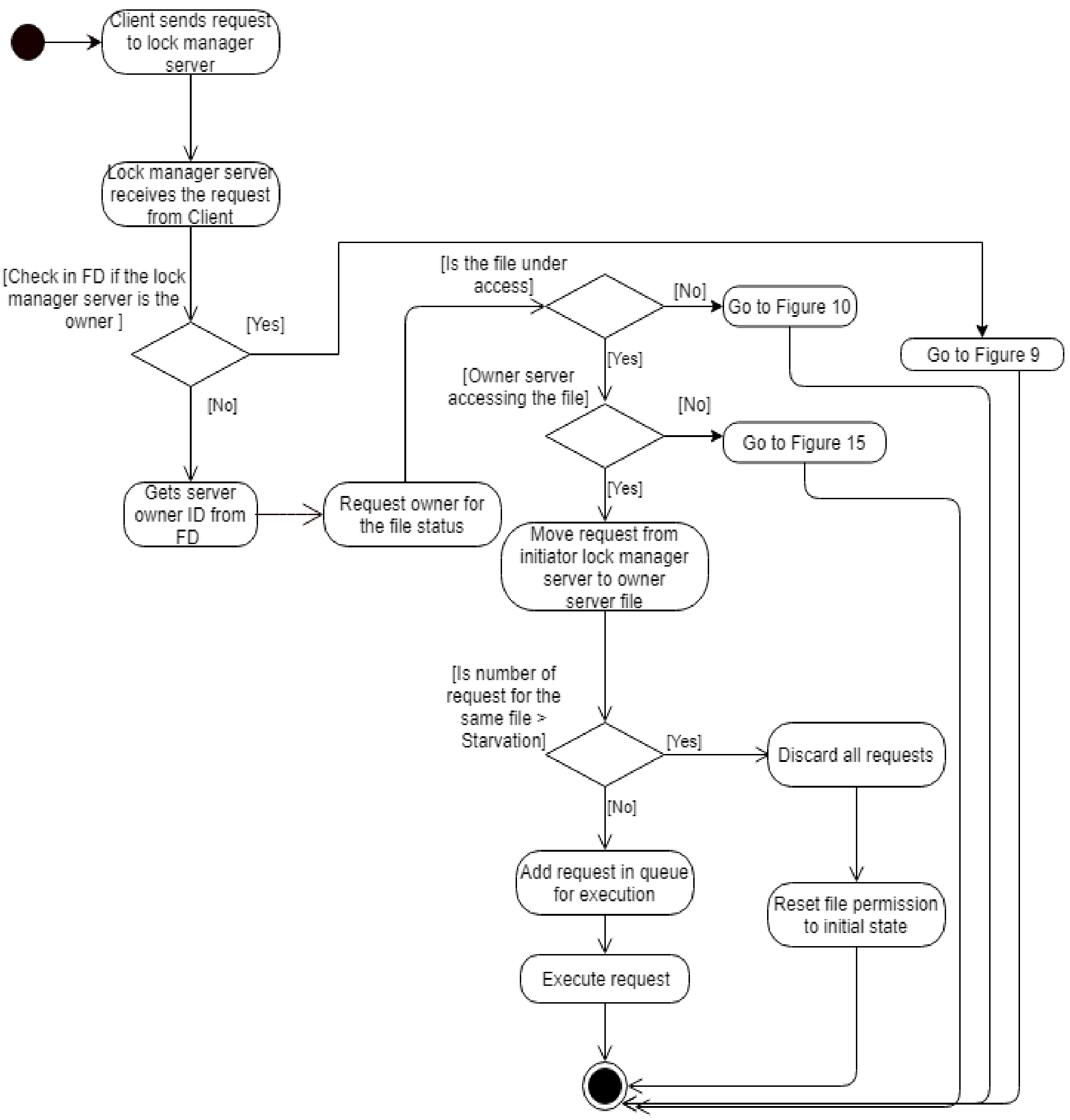

Client sends request to lock manager server
Lock manager server receives the request from Client
[Check in FD if the lock manager server is the owner ]
[Yes]
[No]
Gets server owner ID from FD
Request owner for the file status
[Is the file under access]
[No]
Go to Figure 10
[Yes]
[Owner server accessing the file]
[No]
Go to Figure 15
[Yes]
Go to Figure 9
Move request from initiator lock manager server to owner server file
[Is number of request for the same file > Starvation]
[Yes]
Discard all requests
[No]
Add request in queue for execution
Reset file permission to initial state
Execute request

***Figure 18.*** Initiator Server Migrates Execution Permissions to Remote Server Activity Diagram

```
function manageLock (file, LM1, request)
input: file is the file which gets a request to be accessed
    LM1 lock manager which receives a request to execute a file
    request is the started request
If (LockManagerServer is Owner) then
    …
Else
    Check ownership of file
    Check if file is being accessed
    If (file is accessed) then
        If (OwnerServer is accessing the file) then
            Move request from LM1 to OwnerServer;
            If (Nr of Requests > Starvation) then
                Discard all requests;
            Else
                Queue ← request;
                request.execute();

            Reset file permissions to initial state;
        Else
            …
    Else
```

***Figure 19.*** Initiator server migrates execution permissions to remote server pseudo code

# CHAPTER 5

# IMPLEMENTATION OF LOCK MANAGER ALGORITHM

## 5.1 Introduction

In Chapter 4 we explained in detail the design and functionality of the lock manager algorithm that we have proposed as a solution to maintain concurrent access in cloud storages. In this chapter we are going to explain its implementation in real environment and the way we collected data through different tests we have performed. The simulator is implemented in programming language Java with a simple graphical user interface which provides functions and facilities to build different cluster infrastructure and populate cluster nodes with shared files.

The algorithm code is composed by several entities. Every entity plays an important role for creating and maintaining the functionalities of our algorithm. From Fig. 20 we can see the main entities divided in different classes and, their attributes and operations. One of the entities of our algorithm is Lock Manager class which is responsible for maintaining the components of our cluster, such as file directory, server nodes, locked file list, server status, find owner server of a file etc. In the class Server are configured the main attributes related with server list maintenance such as server Id, Alive status and the mode for issuing requests to nodes. In factory class are included all the attributes and operations for creating the clients, servers, files, storage nodes, and clients. The file node class is responsible for maintaining information about storage nodes. Request Lock class is important for our algorithm because it maintains the information about the mode of lock in files.

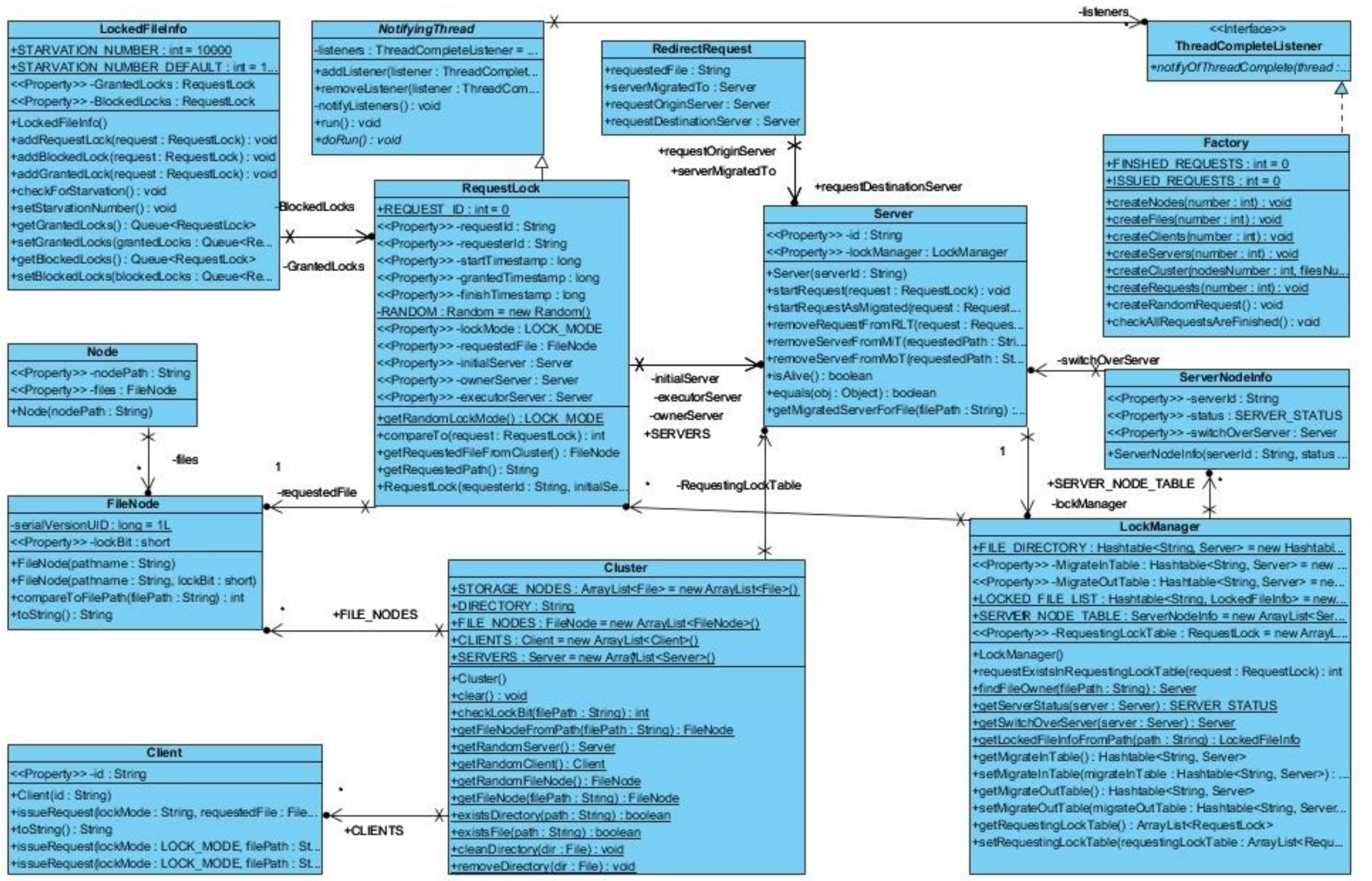
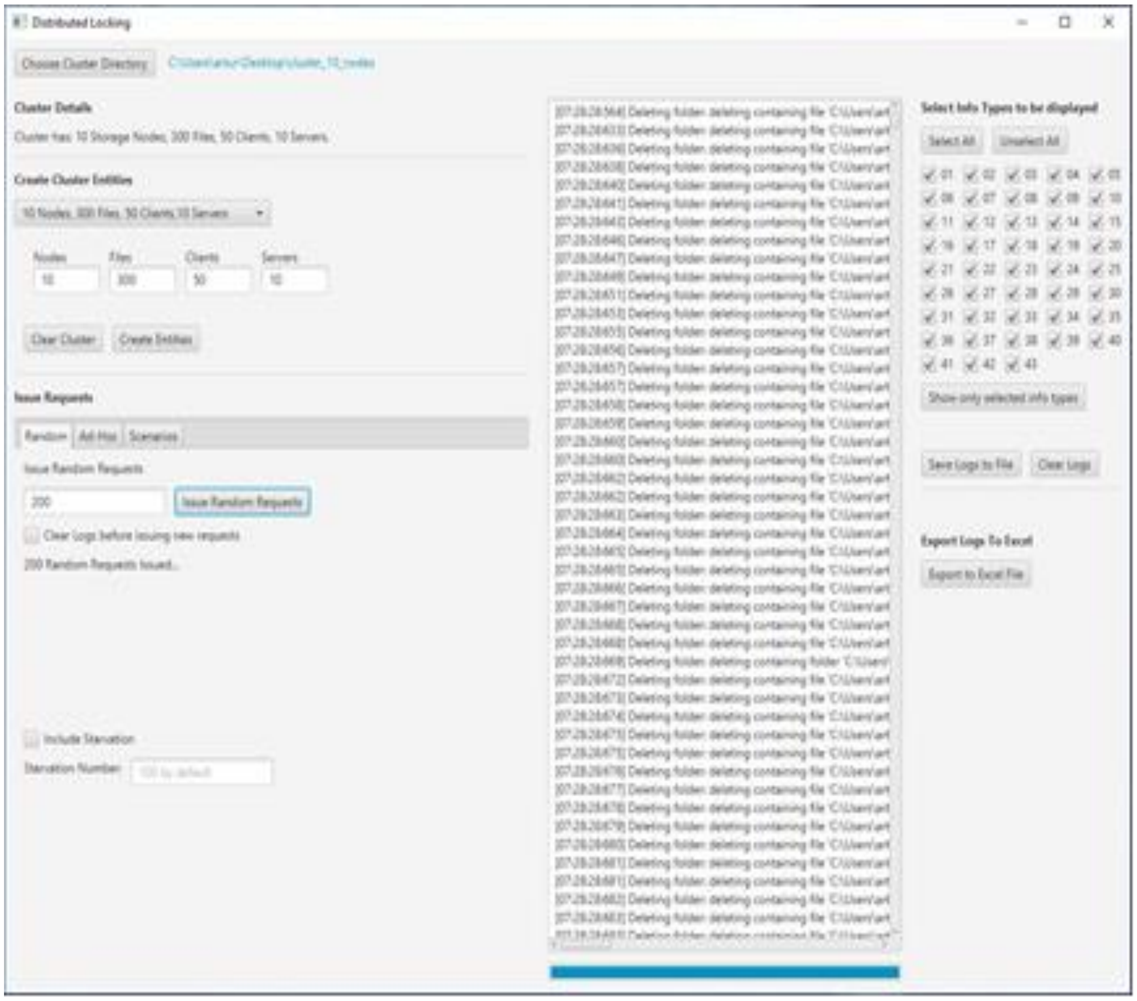

LockedFileInfo
+STARVATION_NUMBER : int = 10000
+STARVATION_NUMBER_DEFAULT : int = 1...
<<Property>> -GrantedLocks : RequestLock
<<Property>> -BlockedLocks : RequestLock
+LockedFileInfo()
+addRequestLock(request : RequestLock) : void
+addBlockedLock(request : RequestLock) : void
+addGrantedLock(request : RequestLock) : void
+checkForStarvation() : void
+setStarvationNumber() : void
+getGrantedLocks() : Queue<RequestLock>
+setGrantedLocks(grantedLocks : Queue<Re...
+getBlockedLocks() : Queue<RequestLock>
+setBlockedLocks(blockedLocks : Queue<Re...
NotifyingThread
-listeners : ThreadCompleteListener = ...
+addListener(listener : ThreadComplet...
+removeListener(listener : ThreadCom...
-notifyListeners() : void
+run() : void
+doRun() : void
RedirectRequest
+requestedFile : String
+serverMigratedTo : Server
+requestOriginServer : Server
+requestDestinationServer : Server
<<Interface>>
ThreadCompleteListener
+notifyOfThreadComplete(thread :...
-listeners
Factory
+FINSHED_REQUESTS : int = 0
+ISSUED_REQUESTS : int = 0
+createNodes(number : int) : void
+createFiles(number : int) : void
+createClients(number : int) : void
+createServers(number : int) : void
+createCluster(nodesNumber : int, filesNu...
+createRequests(number : int) : void
+createRandomRequest() : void
+checkAllRequestsAreFinished() : void
-BlockedLocks
-GrantedLocks
RequestLock
+REQUEST_ID : int = 0
<<Property>> -requestId : String
<<Property>> -requesterId : String
<<Property>> -startTimestamp : long
<<Property>> -grantedTimestamp : long
<<Property>> -finishTimestamp : long
-RANDOM : Random = new Random()
<<Property>> -lockMode : LOCK_MODE
<<Property>> -requestedFile : FileNode
<<Property>> -initialServer : Server
<<Property>> -ownerServer : Server
<<Property>> -executorServer : Server
+getRandomLockMode() : LOCK_MODE
+compareTo(request : RequestLock) : int
+getRequestedFileFromCluster() : FileNode
+getRequestedPath() : String
+RequestLock(requesterId : String, initialSe...
+requestOriginServer
+serverMigratedTo
+requestDestinationServer
Server
<<Property>> -id : String
<<Property>> -lockManager : LockManager
+Server(serverId : String)
+startRequest(request : RequestLock) : void
+startRequestAsMigrated(request : Request...
+removeRequestFromRLT(request : Reques...
+removeServerFromMIT(requestedPath : Stri...
+removeServerFromMoT(requestedPath : St...
+isAlive() : boolean
+equals(obj : Object) : boolean
+getMigratedServerForFile(filePath : String) :...
-switchOverServer
ServerNodeInfo
<<Property>> -serverId : String
<<Property>> -status : SERVER_STATUS
<<Property>> -switchOverServer : Server
+ServerNodeInfo(serverId : String, status ...
-initialServer
-executorServer
-ownerServer
+SERVERS
Node
<<Property>> -nodePath : String
<<Property>> -files : FileNode
+Node(nodePath : String)
-files
1
-requestedFile
-RequestingLockTable
1
+SERVER_NODE_TABLE
-lockManager
FileNode
-serialVersionUID : long = 1L
<<Property>> -lockBit : short
+FileNode(pathname : String)
+FileNode(pathname : String, lockBit : short)
+compareToFilePath(filePath : String) : int
+toString() : String
+FILE_NODES
Cluster
+STORAGE_NODES : ArrayList<File> = new ArrayList<File>()
+DIRECTORY : String
+FILE_NODES : FileNode = new ArrayList<FileNode>()
+CLIENTS : Client = new ArrayList<Client>()
+SERVERS : Server = new ArrayList<Server>()
+Cluster()
+clear() : void
+checkLockBit(filePath : String) : int
+getFileNodeFromPath(filePath : String) : FileNode
+getRandomServer() : Server
+getRandomClient() : Client
+getRandomFileNode() : FileNode
+getFileNode(filePath : String) : FileNode
+existsDirectory(path : String) : boolean
+existsFile(path : String) : boolean
+cleanDirectory(dir : File) : void
+removeDirectory(dir : File) : void
LockManager
+FILE_DIRECTORY : Hashtable<String, Server> = new Hashtabl...
<<Property>> -MigrateInTable : Hashtable<String, Server> = new ...
<<Property>> -MigrateOutTable : Hashtable<String, Server> = ne...
+LOCKED_FILE_LIST : Hashtable<String, LockedFileInfo> = new...
+SERVER_NODE_TABLE : ServerNodeInfo = new ArrayList<Ser...
<<Property>> -RequestingLockTable : RequestLock = new ArrayL...
+LockManager()
+requestExistsInRequestingLockTable(request : RequestLock) : int
+findFileOwner(filePath : String) : Server
+getServerStatus(server : Server) : SERVER_STATUS
+getSwitchOverServer(server : Server) : Server
+getLockedFileInfoFromPath(path : String) : LockedFileInfo
+getMigrateInTable() : Hashtable<String, Server>
+setMigrateInTable(migrateInTable : Hashtable<String, Server>) : ...
+getMigrateOutTable() : Hashtable<String, Server>
+setMigrateOutTable(migrateOutTable : Hashtable<String, Server...
+getRequestingLockTable() : ArrayList<RequestLock>
+setRequestingLockTable(requestingLockTable : ArrayList<Requ...
Client
<<Property>> -id : String
+Client(id : String)
+issueRequest(lockMode : String, requestedFile : File...
+toString() : String
+issueRequest(lockMode : LOCK_MODE, filePath : St...
+issueRequest(lockMode : LOCK_MODE, filePath : St...
+CLIENTS

***Figure 20.*** Class Diagram

The front-end section is composed of different modules and each module provides a different function. From the Fig. 21 we can denote the module of cluster directory. Cluster directory module provides functions to create cluster entities such as storage nodes, shared files, client's variables, and master servers. Cluster directory module implements features to empty directory and builds the other entities with a different composition. Values of each entity can be selected according to developer needs and there exist no boundaries to limit cluster size.



C:\Users\artur\Desktop\cluster_10_nodes

Cluster has: 10 Storage Nodes, 300 Files, 50 Clients, 10 Servers.

10 Nodes, 300 Files, 50 Clients,10 Servers

Nodes: 10 Files: 300 Clients: 50 Servers: 10

Issue Random Requests: 200

☐ Clear Logs before issuing new requests

200 Random Requests Issued...

☐ Include Starvation

Starvation Number: 100 by default

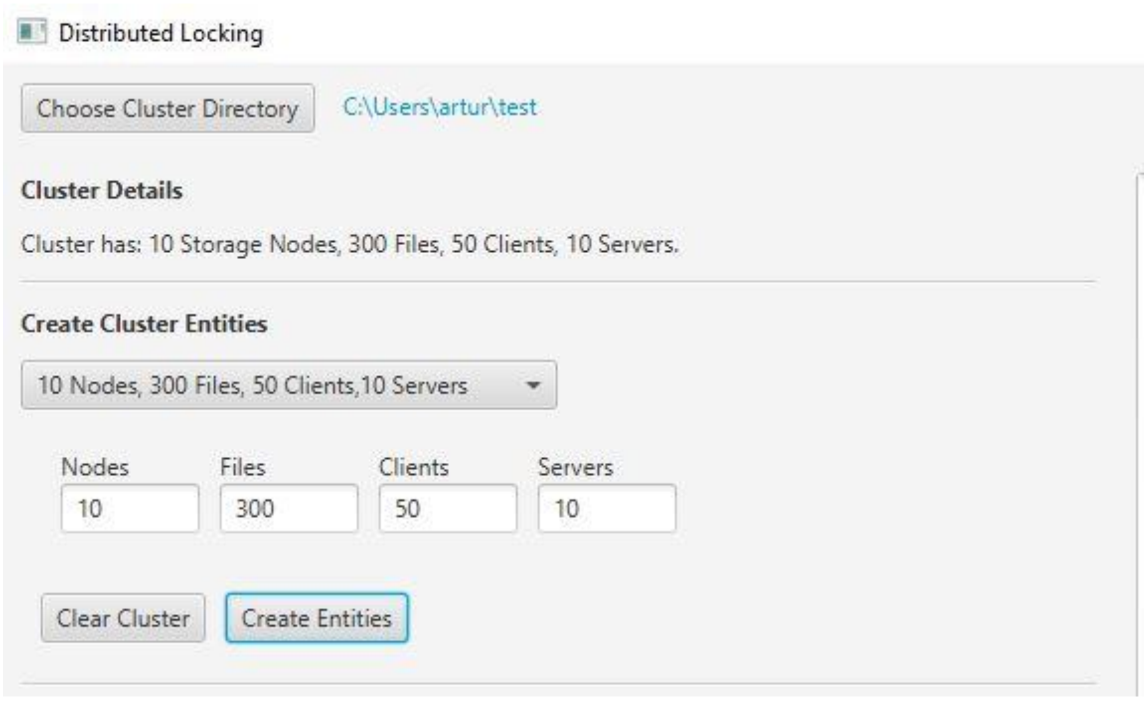
✓ 01 ✓ 02 ✓ 03 ✓ 04 ✓ 05
✓ 06 ✓ 07 ✓ 08 ✓ 09 ✓ 10
✓ 11 ✓ 12 ✓ 13 ✓ 14 ✓ 15
✓ 16 ✓ 17 ✓ 18 ✓ 19 ✓ 20
✓ 21 ✓ 22 ✓ 23 ✓ 24 ✓ 25
✓ 26 ✓ 27 ✓ 28 ✓ 29 ✓ 30
✓ 31 ✓ 32 ✓ 33 ✓ 34 ✓ 35
✓ 36 ✓ 37 ✓ 38 ✓ 39 ✓ 40
✓ 41 ✓ 42 ✓ 43

```
[07:28:28.564] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.633] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.636] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.638] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.640] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.641] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.643] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.646] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.647] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.649] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.651] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.653] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.655] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.656] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.657] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.657] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.658] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.659] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.660] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.660] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.662] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.662] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.663] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.664] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.665] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.665] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.666] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.667] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.668] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.668] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.669] Deleting folder: deleting containing folder 'C:\Users\
[07:28:28.672] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.673] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.674] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.675] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.675] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.676] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.677] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.678] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.679] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.680] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.681] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.681] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.682] Deleting folder: deleting containing file 'C:\Users\art
[07:28:28.683] Deleting folder: deleting containing file 'C:\Users\art
```

*Figure 21.* Lock Manager Graphical Interface

Issue request module provides functions and facilities used by clients to interact with master servers and issues request to access shared files stored under storage nodes. The main function provided by this module is related with the way to interact with master nodes that can happen in random mode, predefined scenarios and ad-hoc mode that gives the option to predefine the way followed by requests and which file to access.

Select info types to be displayed is another module that provides function and facilities to filter logs that the server will collect while running the process. This is an essential module because it keeps track of every process running and provides detailed data which we are going to analyze in order to measure the performance of our algorithm.



C:\Users\artur\test

**Cluster Details**

Cluster has: 10 Storage Nodes, 300 Files, 50 Clients, 10 Servers.

**Create Cluster Entities**

10 Nodes, 300 Files, 50 Clients,10 Servers

| Nodes | Files | Clients | Servers |
|---|---|---|---|
| 10 | 300 | 50 | 10 |

***Figure 22.*** Cluster Directory Module

The first procedure that needs to be done is to select the path where the nodes are created and to define the number of files per each node. The algorithm is designed to give us the possibility to use predefined values; or, we can also choose the number of servers, nodes, files per node and the number of users that deliver requests for accessing files stored in nodes.

### 5.1.1 Building Lock Manager Algorithm Infrastructure

The example shown in Fig. 22 refers to Cluster directory module. We have chosen to build an infrastructure with 10 servers, 10 nodes and 300 files stored in all nodes. There are 50 virtual users that we have named as clients, who will deliver requests for accessing the files that are stored in shared nodes.

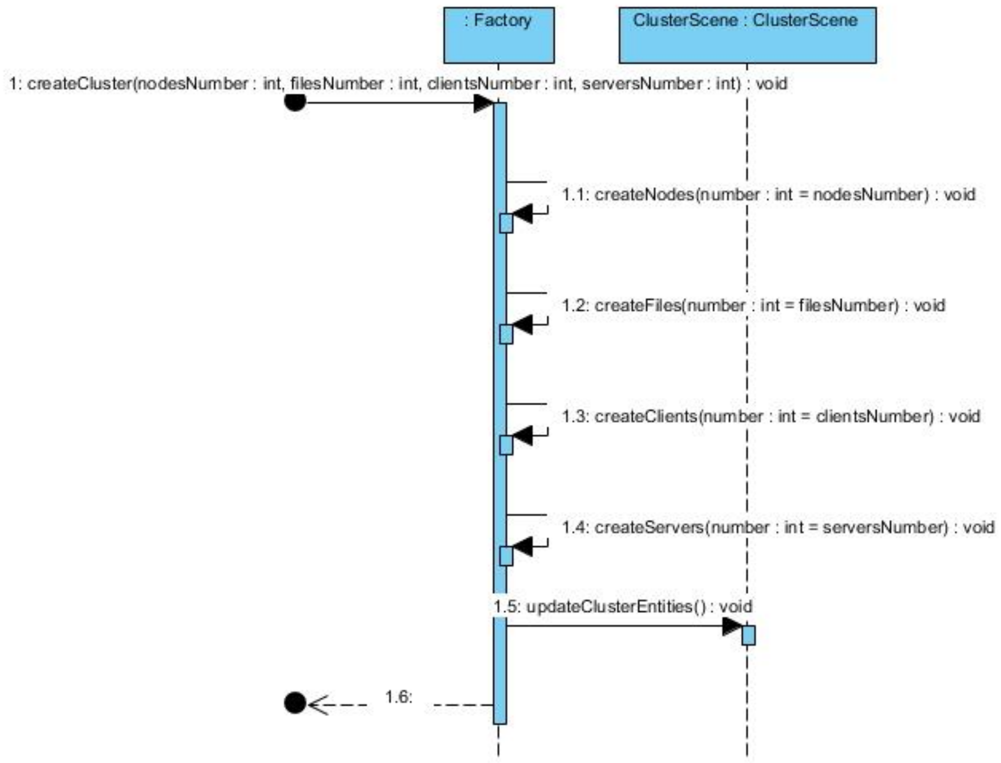
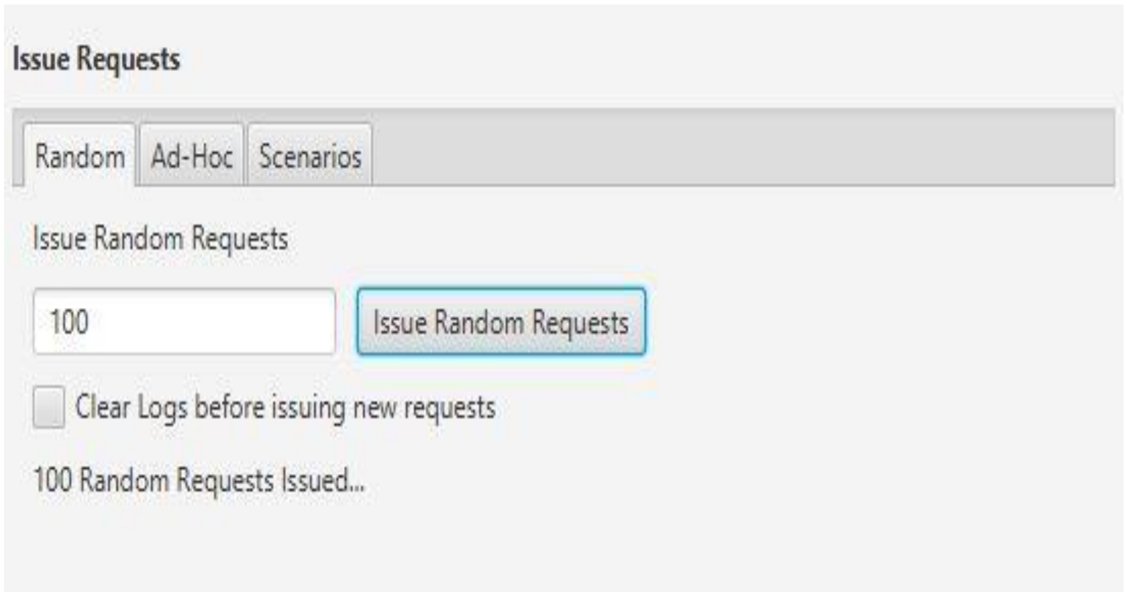

: Factory
ClusterScene : ClusterScene
1: createCluster(nodesNumber : int, filesNumber : int, clientsNumber : int, serversNumber : int) : void
1.1: createNodes(number : int = nodesNumber) : void
1.2: createFiles(number : int = filesNumber) : void
1.3: createClients(number : int = clientsNumber) : void
1.4: createServers(number : int = serversNumber) : void
1.5: updateClusterEntities() : void
1.6:

***Figure 23.*** Cluster Directory Module Sequence Diagram

The algorithm design gives the option to define a parameter which represents the number of requests sent by virtual users that will request to write/read one of the shared files. In this example we have chosen to run one of the predefined values. For easy convention we have named the cluster as test. The sequence of steps performed by cluster directory module for creating the cluster is described in Fig. 23.

### 5.1.2 Request Delivery Mode

Once the infrastructure is built we can deliver requests to perform read/write operations in the stored files. When one of the files is read/ write, we insert a stick that describes the type of the operation performed in it, the time when the operation happened and how much time it took to accomplish the operation. The request can be delivered in different numbers and different modes. Through modes we understand the way how the request is delivered and whether these requests are delivered in a chosen order or all requests are delivered randomly. For the sake of analysis, we have implemented both modes, random and ad-hoc mode.



100

100 Random Requests Issued...

***Figure 24.*** Random Issuing Request Module

In the ad-hoc mode (Fig. 25), it is us who decided the number of requests, which server was to act as initiator, which file was to be requested and what the operation performed would be. We decided that it was a write or read operation, or we left operations to be randomly decided by algorithm. Meanwhile, in random mode (Fig. 22) we have no control over the type of requests delivered. The type of requests and files are chosen automatically by the algorithm itself.

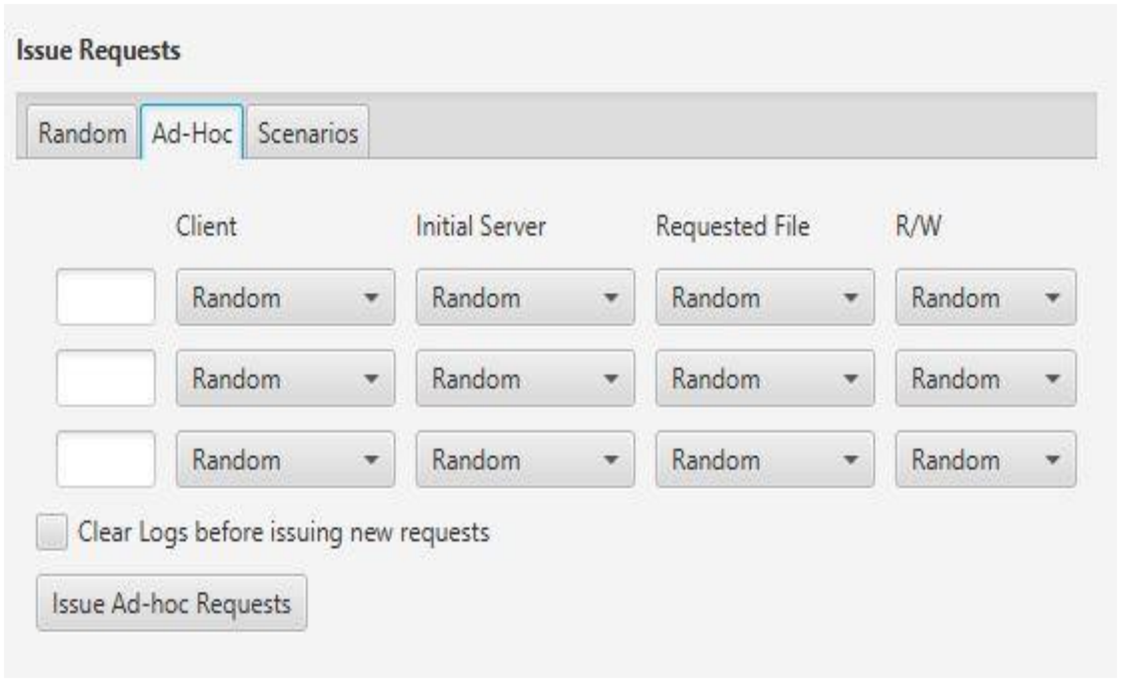


***Figure 25.*** Ad-Hoc Request Module

Providing different modes to clients for issuing requests to servers has also different approaching of coding. From the moment the request has been issued until its fulfilment, the algorithm follows a strict sequence of steps. The first step implies the sequence of request creation described in Fig. 26. The total number of requests is controlled by a loop which continues to execute for all the time the condition is true.

The way of delivering request to servers changes the sequence of steps followed by the program to perform, for the mode random the sequence of steps is presented in Fig. 27.

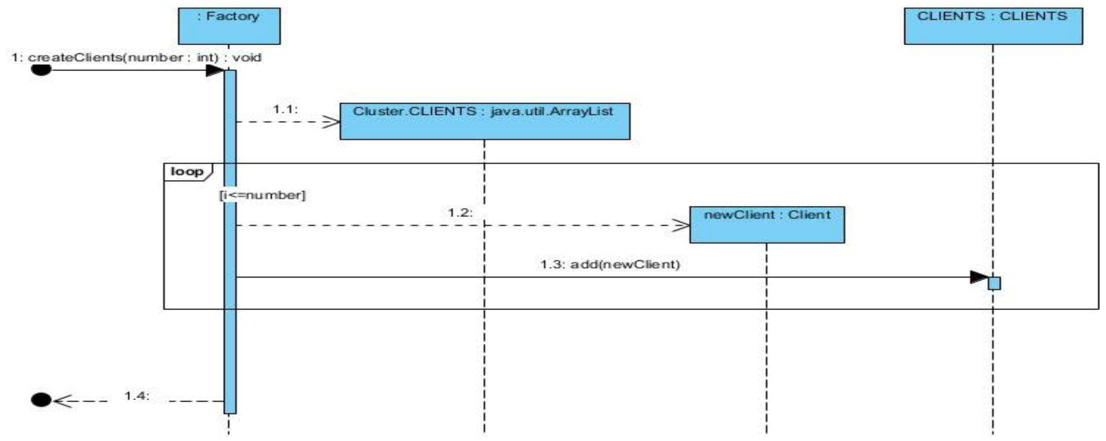


***Figure 26.*** Request Creation Sequence Diagram

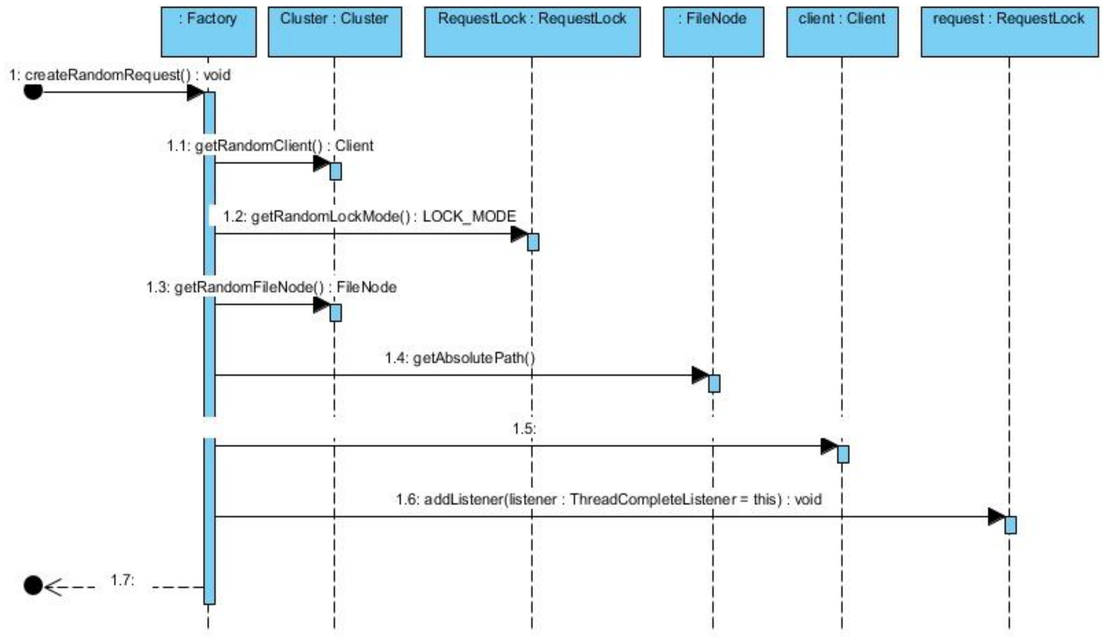

: Factory
Cluster : Cluster
RequestLock : RequestLock
: FileNode
client : Client
request : RequestLock
1: createRandomRequest() : void
1.1: getRandomClient() : Client
1.2: getRandomLockMode() : LOCK_MODE
1.3: getRandomFileNode() : FileNode
1.4: getAbsolutePath()
1.5:
1.6: addListener(listener : ThreadCompleteListener = this) : void
1.7:

***Figure 27.*** Random Mode Delivery Request Sequence Diagram

For the mode, where the user would like to specify the type of the client who will deliver the request, the operation type, the server to where the request will be delivered, and which file is required are embodied in Ad-Hoc mode, which is described in Fig. 28.

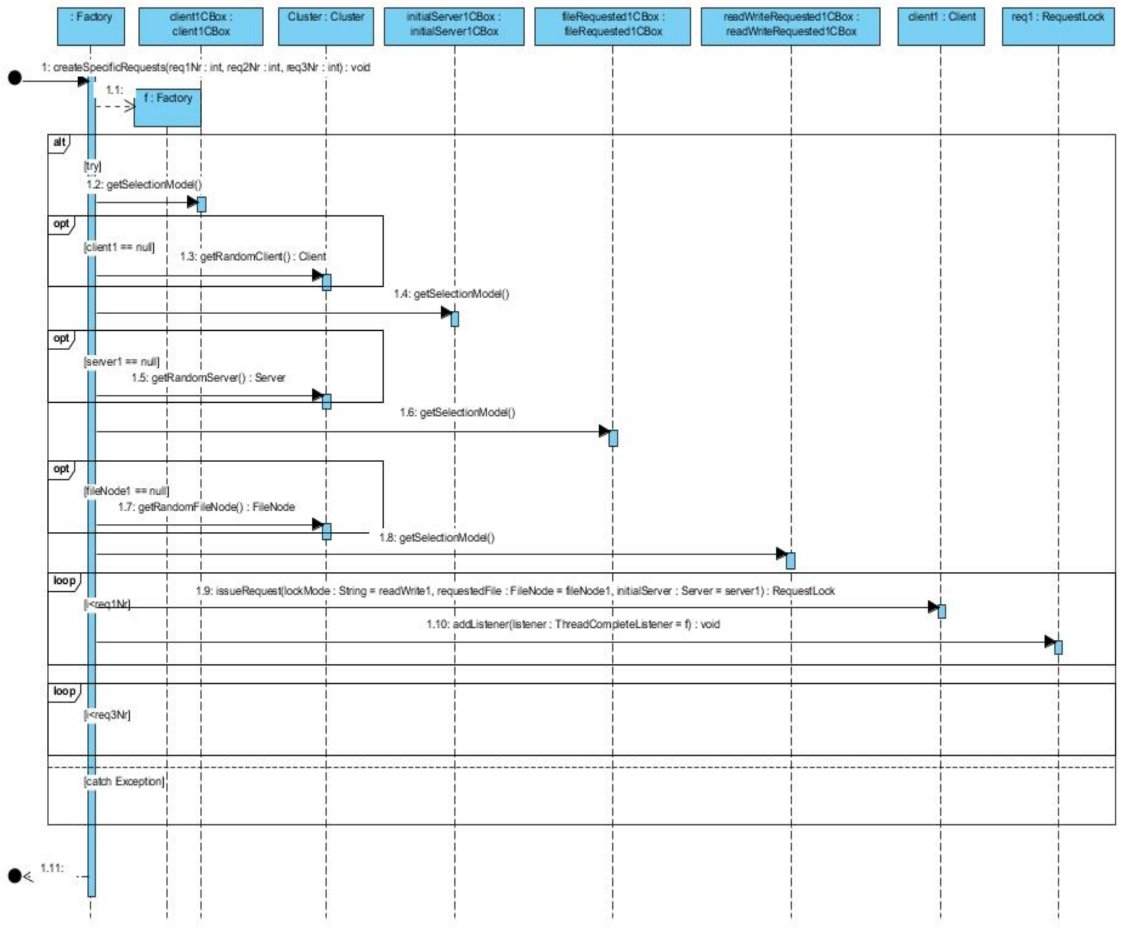
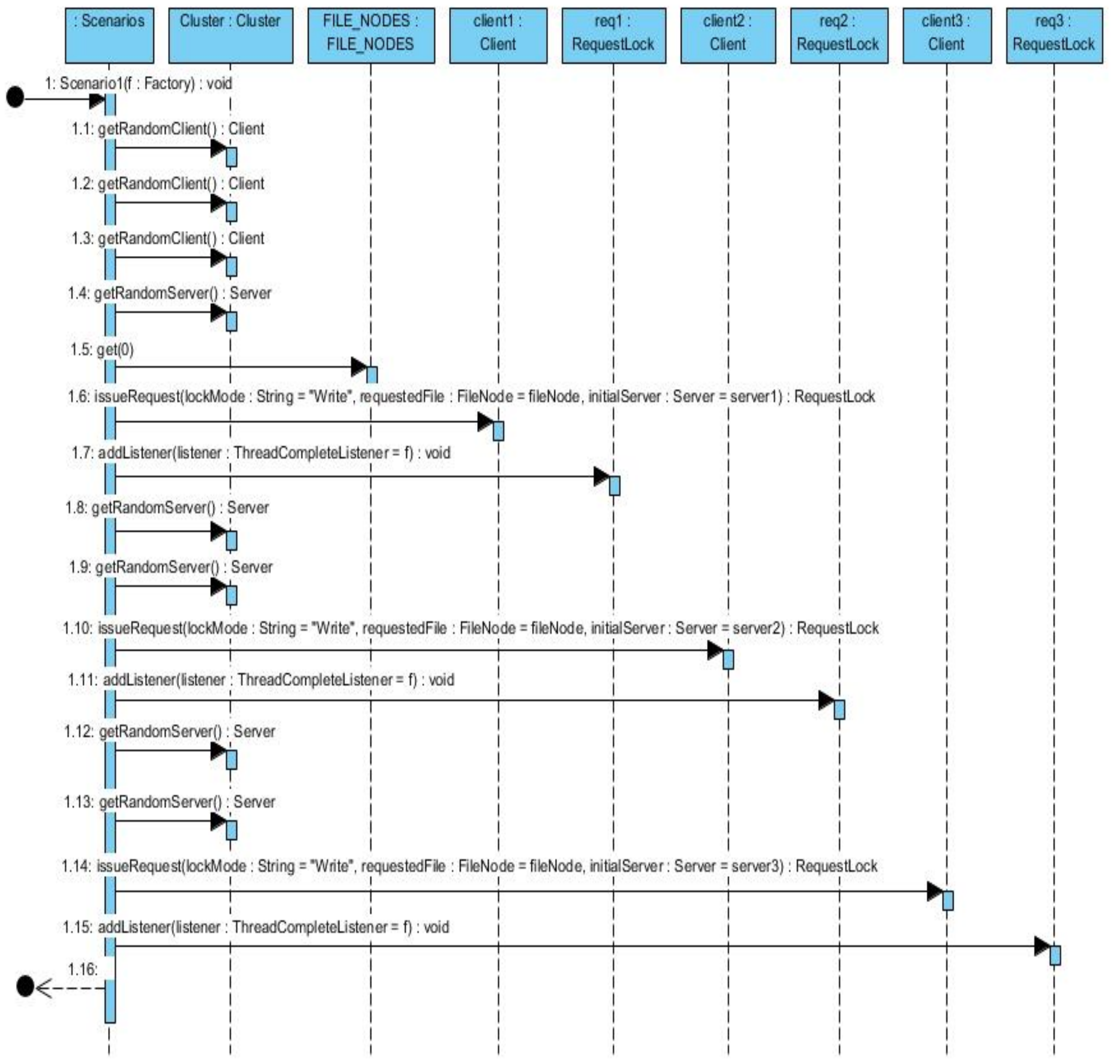

: Factory
client1CBox : client1CBox
Cluster : Cluster
initialServer1CBox : initialServer1CBox
fileRequested1CBox : fileRequested1CBox
readWriteRequested1CBox : readWriteRequested1CBox
client1 : Client
req1 : RequestLock
1: createSpecificRequests(req1Nr : int, req2Nr : int, req3Nr : int) : void
1.1:
f : Factory
alt
[try]
1.2: getSelectionModel()
opt
[client1 == null]
1.3: getRandomClient() : Client
1.4: getSelectionModel()
opt
[server1 == null]
1.5: getRandomServer() : Server
1.6: getSelectionModel()
opt
[fileNode1 == null]
1.7: getRandomFileNode() : FileNode
1.8: getSelectionModel()
loop
[i<req1Nr]
1.9: issueRequest(lockMode : String = readWrite1, requestedFile : FileNode = fileNode1, initialServer : Server = server1) : RequestLock
1.10: addListener(listener : ThreadCompleteListener = f) : void
loop
[i<req3Nr]
[catch Exception]
1.11:

***Figure 28.*** Specific Request Deliver Sequence Diagram

Finally, the algorithm provides some basic scenarios which can be used to perform specific tasks according to some predefined tests. The sequence of steps followed to perform this process is given in Fig. 29. To simplify the design, in the sequence diagram we have listed all steps followed by one single request. The sequence for the other request is a repetition of the same steps.



: Scenarios
Cluster : Cluster
FILE_NODES : FILE_NODES
client1 : Client
req1 : RequestLock
client2 : Client
req2 : RequestLock
client3 : Client
req3 : RequestLock
1: Scenario1(f : Factory) : void
1.1: getRandomClient() : Client
1.2: getRandomClient() : Client
1.3: getRandomClient() : Client
1.4: getRandomServer() : Server
1.5: get(0)
1.6: issueRequest(lockMode : String = "Write", requestedFile : FileNode = fileNode, initialServer : Server = server1) : RequestLock
1.7: addListener(listener : ThreadCompleteListener = f) : void
1.8: getRandomServer() : Server
1.9: getRandomServer() : Server
1.10: issueRequest(lockMode : String = "Write", requestedFile : FileNode = fileNode, initialServer : Server = server2) : RequestLock
1.11: addListener(listener : ThreadCompleteListener = f) : void
1.12: getRandomServer() : Server
1.13: getRandomServer() : Server
1.14: issueRequest(lockMode : String = "Write", requestedFile : FileNode = fileNode, initialServer : Server = server3) : RequestLock
1.15: addListener(listener : ThreadCompleteListener = f) : void
1.16:

***Figure 29.*** Scenario Mode Delivery Request Sequence Diagram

Once delivered, every request has its own specifics that can be similar to some other request. There exist cases where multiple clients request the same file and wish to perform the same operation. In some other cases multiple clients request the same file and wish to perform different operations. These are cases which our algorithm addresses and it eliminates the implication of different operations in the same file when concurrent access occurs. The steps followed to maintain file consistency are described in Fig. 30.

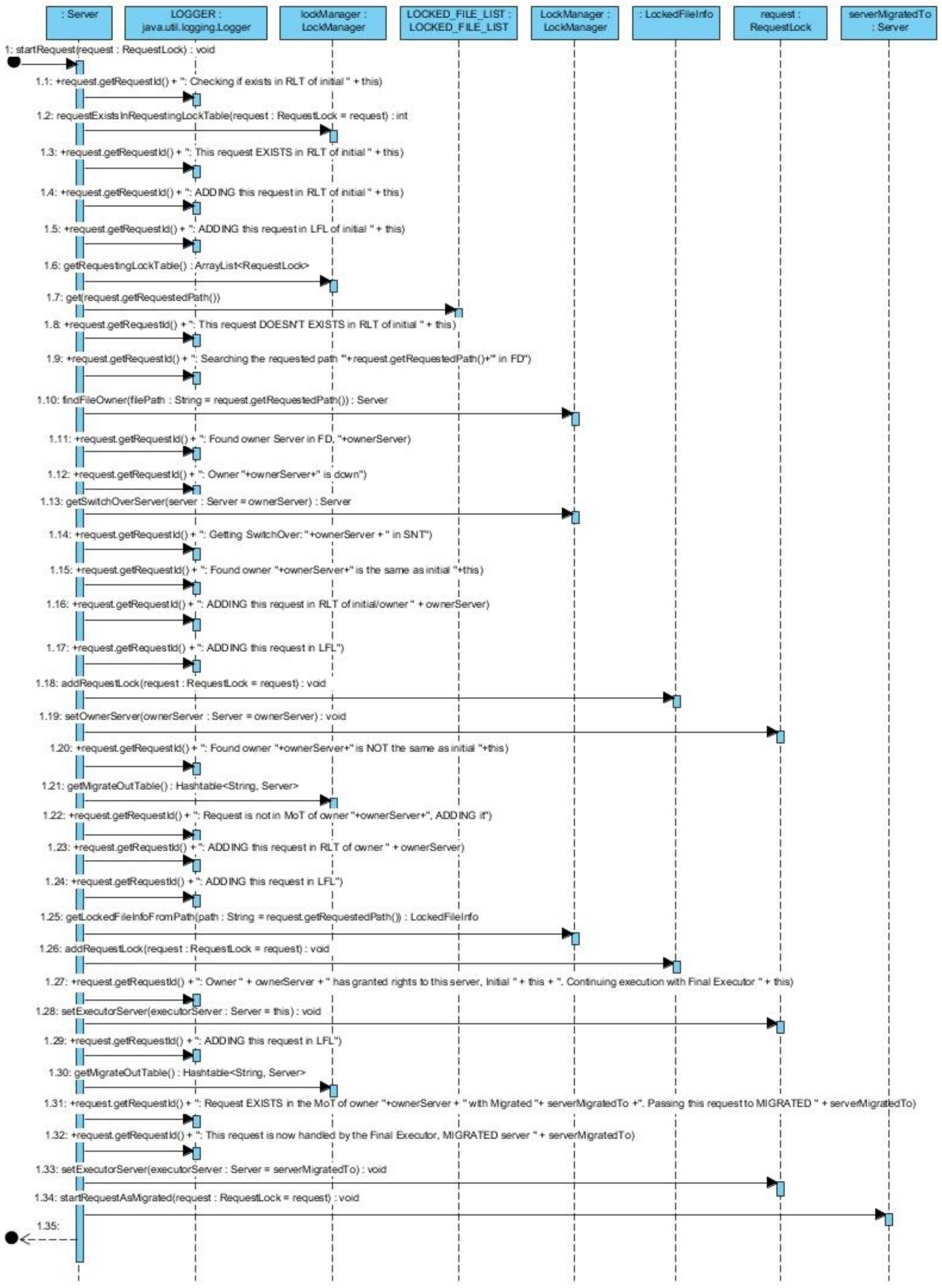

: Server
LOGGER : java.util.logging.Logger
lockManager : LockManager
LOCKED_FILE_LIST : LOCKED_FILE_LIST
LockManager : LockManager
: LockedFileInfo
request : RequestLock
serverMigratedTo : Server
1: startRequest(request : RequestLock) : void
1.1: +request.getRequestId() + ": Checking if exists in RLT of initial " + this)
1.2: requestExistsInRequestingLockTable(request : RequestLock = request) : int
1.3: +request.getRequestId() + ": This request EXISTS in RLT of initial " + this)
1.4: +request.getRequestId() + ": ADDING this request in RLT of initial " + this)
1.5: +request.getRequestId() + ": ADDING this request in LFL of initial " + this)
1.6: getRequestingLockTable() : ArrayList<RequestLock>
1.7: get(request.getRequestedPath())
1.8: +request.getRequestId() + ": This request DOESN'T EXISTS in RLT of initial " + this)
1.9: +request.getRequestId() + ": Searching the requested path "+request.getRequestedPath()+" in FD")
1.10: findFileOwner(filePath : String = request.getRequestedPath()) : Server
1.11: +request.getRequestId() + ": Found owner Server in FD, "+ownerServer)
1.12: +request.getRequestId() + ": Owner "+ownerServer+" is down")
1.13: getSwitchOverServer(server : Server = ownerServer) : Server
1.14: +request.getRequestId() + ": Getting SwitchOver: "+ownerServer + " in SNT")
1.15: +request.getRequestId() + ": Found owner "+ownerServer+" is the same as initial "+this)
1.16: +request.getRequestId() + ": ADDING this request in RLT of initial/owner " + ownerServer)
1.17: +request.getRequestId() + ": ADDING this request in LFL")
1.18: addRequestLock(request : RequestLock = request) : void
1.19: setOwnerServer(ownerServer : Server = ownerServer) : void
1.20: +request.getRequestId() + ": Found owner "+ownerServer+" is NOT the same as initial "+this)
1.21: getMigrateOutTable() : Hashtable<String, Server>
1.22: +request.getRequestId() + ": Request is not in MoT of owner "+ownerServer+", ADDING it")
1.23: +request.getRequestId() + ": ADDING this request in RLT of owner " + ownerServer)
1.24: +request.getRequestId() + ": ADDING this request in LFL")
1.25: getLockedFileInfoFromPath(path : String = request.getRequestedPath()) : LockedFileInfo
1.26: addRequestLock(request : RequestLock = request) : void
1.27: +request.getRequestId() + ": Owner " + ownerServer + " has granted rights to this server, Initial " + this + ". Continuing execution with Final Executor " + this)
1.28: setExecutorServer(executorServer : Server = this) : void
1.29: +request.getRequestId() + ": ADDING this request in LFL")
1.30: getMigrateOutTable() : Hashtable<String, Server>
1.31: +request.getRequestId() + ": Request EXISTS in the MoT of owner "+ownerServer + " with Migrated "+ serverMigratedTo +". Passing this request to MIGRATED " + serverMigratedTo)
1.32: +request.getRequestId() + ": This request is now handled by the Final Executor, MIGRATED server " + serverMigratedTo)
1.33: setExecutorServer(executorServer : Server = serverMigratedTo) : void
1.34: startRequestAsMigrated(request : RequestLock = request) : void
1.35:

***Figure 30.*** Lock Management Sequence Diagram

### 5.1.3 Data collected

Another aspect of the design is the information collected and its manipulation. The algorithm gives information for all internal procedures running to achieve the action of the client request. Once the execution of all requests is finished, we are able to export the logs in Excel format or in text file and from there we can analyze and prepare reports on the entire execution processes.

In table 1, we introduce a detailed data description collected in lock manager algorithm, describing each procedure starting from the moment the request was created until it was successfully completed. For each step, algorithm inserts a timestamp referring to the moment the procedure was initiated.

The logs exported in Excel format contain a detailed report of a number of parameters such as Request Id, Operation mode, whether permission is already granted for that file, Initiator Server same as Owner, Client Requesting, Requested File, Initiator Server, Owner Server, Executer Server, Started Timestamp, Granted Timestamp, Finished Timestamp and Execution Time.

| ID | Info Type \ Example |
|---|---|
| 01 | **Request created** |
| | [01:10:27:887] [01] Req1: Request Created = RequestLock [Request ID=Req1, Client Requesting ID=2, Lock Mode=Write, Requested File=file3.txt, Start Timestamp=01:10:27:887, Initial Server=Server[3]] |
| 02 | **Request thread started** |
| | [20:48:30:365] [02] Req2: Request started in thread 19 |
| 03 | **Request passed to initial server** |
| | [20:48:30:365] [03] Req2: Passed to initial Server[3] |
| 04 | **Checking if request exists in RLT of initial server** |
| | [20:48:30:365] [04] Req2: Checking if exists in RLT of initial Server[3] |
| 05 | **Request exists in RLT of initial server** |
| | |

| | |
|---|---|
| **06** | **Adding Request in RLT of initial server** |
| | |
| **07** | **Adding Request in LFL** |
| | [01:10:27:945] [07] Req2: ADDING this request in LFL |
| **08** | **Request doesn’t exist in RLT of initial server** |
| | [20:48:30:365] [08] Req2: This request DOESN'T EXISTS in RLT of initial Server[3] |
| **09** | **Searching the requested path in FD** |
| | [20:48:30:365] [09] Req1: Searching the requested path 'C:\Users\U000000\Google Drive\DistributedLocking\Cluster\Node2\file2.txt' in FD |
| **10** | **Found owner server in FD** |
| | [20:48:30:365] [10] Req2: Found owner Server in FD, Server[2] |
| **11** | **Owner server is down** |
| | |
| **12** | **Getting switchover server in SNT** |
| | |
| **13** | **Owner server is the same as initial server** |
| | [20:48:30:365] [13] Req1: Found owner Server[2] is the same as initial Server[2] |
| **14** | **Adding Request in RLT of initial server which is the same as the owner server** |
| | [20:48:30:365] [14] Req1: ADDING this request in RLT of initial/owner Server[2] |
| **15** | **Owner server is not the same as the initial server** |
| | [20:48:30:365] [15] Req2: Found owner Server[2] is NOT the same as initial Server[3] |
| **16** | **Request is not in the MiT of initial server, adding it** |
| | [20:48:30:365] [16] Req2: Request is not in MiT of initial Server[3], ADDING it |
| **17** | **Request is already in the MiT of initial server** |
| | [20:48:30:365] [17] Req2: Request is not in MoT of owner Server[2], ADDING it |
| **18** | **Request is not in the MoT of owner server, adding it** |
| | |
| **19** | **Request is already in the MoT of owner server** |
| | |

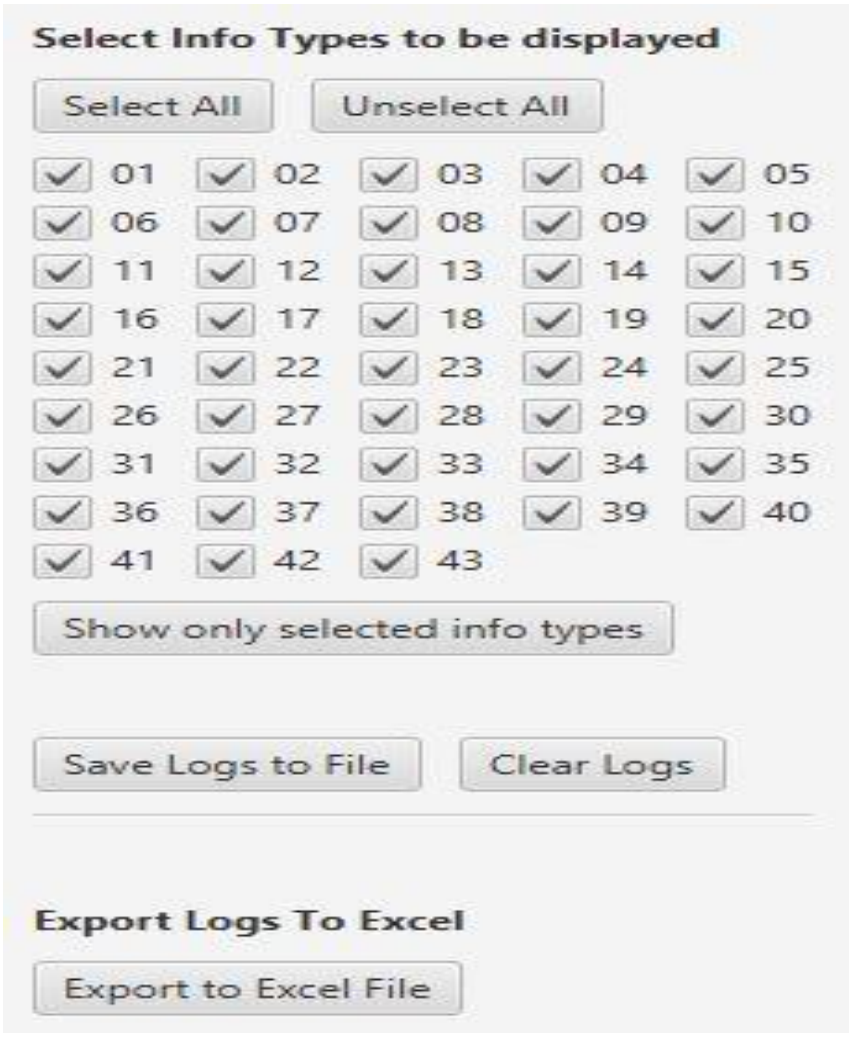
| | |
|---|---|
| **20** | **Adding Request in the RLT of owner server** |
| | |
| **21** | **Checking lock bit of request file** |
| | [20:48:30:365] [21] Req1: Checking BitLock if permission for requested file can be granted... |
| **22** | **Printing lock bit: Requested File has lock bit 0** |
| | [20:48:30:365] [22] Req1: Requested file has BitLock = 0 |
| **23** | **Printing Lock Mode: Lock Mode is Read** |
| | [20:48:30:365] [23] Req2: Requested LockMode is READ |
| **24** | **Request added in Blocked List** |
| | [23:13:23:725] [24] Req1: Added in Blocked List |
| **25** | **Request added in Granted Queue** |
| | [20:48:30:365] [25] Req1: Added in Granted Queue |
| **26** | **File read successfully** |
| | [20:48:30:365] [26] Req1: File READ successfully! |
| **27** | **File written successfully** |
| | [20:48:30:365] [27] Req2: File WRITTEN successfully! |
| **28** | **Request finished** |
| | [20:49:46:224] [28] Req3: FINISHED! - RequestLock [Request ID=Req3, Client Requesting ID=2, Lock Mode=Write, Requested File=file1.txt, Start Timestamp=20:49:46:117, Granted Timestamp=20:49:46:195, Finish Timestamp=20:49:46:224, Initial Server=Server[1], Owner Server=Server[3]] |
| **29** | **Printing all three timestamps of a finished Request** |
| | [20:48:30:365] [29] Req1: TIMESTAMPS: Start=20:48:30:337, Granted=20:48:30:346, Finished=20:48:30:364 |
| **30** | **Printing the execution time in milliseconds of a finished Request** |
| | [20:48:30:365] [30] Req1: Execution time = 18 milliseconds |
| **31** | **Printing lock bit: Requested File has lock bit 1** |
| | [20:48:30:365] [22] Req1: Requested file has BitLock = 1 |
| **32** | **Printing lock bit: Requested File has lock bit 2** |
| | [20:48:30:365] [22] Req1: Requested file has BitLock = 2 |
| **33** | **Printing Lock Mode: Lock Mode is Write** |
| | [20:48:30:365] [23] Req2: Requested LockMode is WRITE |
| **34** | **Request removed from Granted List** |
| | [23:13:23:766] [34] Req2: Request removed from Granted List |

| | |
|---|---|
| **35** | **Granted Lock List is empty, releasing Blocked Requests from Blocked Locks Queue** |
| | [22:54:48:315] [35] Req3: Granted Lock List for File file1.txt is empty, releasing Blocked Requests |
| **36** | **No Blocked Request to release** |
| | [22:54:48:315] [36] Req3: Releasing Blocked Requests - there are no Blocked requests for File file1.txt |
| **37** | **Releasing all blocked READ requests up to a WRITE request** |
| | [22:54:48:290] [37] Req1: Releasing Blocked Requests - the first Blocked request is READ mode. Getting all Blocked READ requests up to a WRITE request |
| **38** | **Releasing a Blocked Request** |
| | [22:54:48:290] [38] Req1: Releasing Blocked Request - releasing Req2, Read mode |
| **39** | **Releasing the first Blocked WRITE Request** |
| | [23:04:49:440] [39] Req1: Releasing Blocked Request - releasing the first WRITE request: Req2 |
| **40** | **Migrating request** |
| | [14:44:58:796] [40] Req2: Owner Server[5] has granted rights to this server, Initial Server[1]. Continuing execution with Final Executor Server[1] |
| **41** | **Request is dropped due to starvation** |
| | [23:08:53:615] [41] Req102: Request is dropped due to starvation; 100 requests are waiting for permissions for file file1.txt |
| **42** | **Lock Bit Reset due to starvation** |
| | [23:08:53:619] [42] Lock Bit of file file1.txt is set to 0 due to starvation |
| **43** | **Request is granted access after starvation, as the first blocked request** |
| | [23:08:53:621] [43] Req7: Request is started after starvation of file file1.txt |

***Table 1.*** List of Processes Performed for Fulfilling Execution of A Single Request

### 5.1.4 Filtering information

Select Info Types to be displayed section (Fig. 31) is used for filtering the kind of the information to be collected during the execution process. The asymmetry of the algorithm relies on the fact that requests are executed into two different ways. In the first way requests are executed by the local server without collaborating with other servers, part of the cloud; the second way refers to the approach when it is obligatory for the server to receive information about the status of a specific file. Besides servers involved in request execution process, no other servers have clues about the status of a specific request or of a lock applied to a specific file. In this approach the requisite for a master to maintain the overall consistency and load balancing is eliminated.



.

***Figure 31.*** Information Selection Module

In Table 2 we provide an example of data exported in Excel format. This example refers to the case where lock manager 7 requests lock manager server 10 to grant permission for a file named file34. After verifying status of the requested file lock manager server 10 has concluded that the file named file34 is free from lock and has granted execution permission to lock manager server 7.

| Client | Req. File | Initial Server | Owner Server | Executor Server | Started Timestamp | Granted Timestamp | Finished Timestamp | Exec Time | Time until Granted |
|---|---|---|---|---|---|---|---|---|---|
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:269 | 09:43:33:194 | 09:43:33:519 | 1250 | 925 |
| Client[1] | file34.txt | Server[7] | Server[10] | Not migrated | 09:43:32:270 | 09:43:32:279 | 09:43:32:438 | 168 | 9 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:270 | 09:43:33:052 | 09:43:33:551 | 1281 | 782 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:272 | 09:43:33:072 | 09:43:33:547 | 1275 | 800 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:273 | 09:43:33:099 | 09:43:33:540 | 1267 | 826 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:275 | 09:43:32:409 | 09:43:32:431 | 156 | 134 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:277 | 09:43:33:079 | 09:43:33:546 | 1269 | 802 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:277 | 09:43:33:169 | 09:43:33:524 | 1247 | 892 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:278 | 09:43:33:181 | 09:43:33:521 | 1243 | 903 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:281 | 09:43:32:433 | 09:43:32:745 | 464 | 152 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:281 | 09:43:33:220 | 09:43:33:512 | 1231 | 939 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:282 | 09:43:32:393 | 09:43:32:775 | 493 | 111 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:282 | 09:43:32:391 | 09:43:32:827 | 545 | 109 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:282 | 09:43:33:059 | 09:43:33:550 | 1268 | 777 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:283 | 09:43:33:112 | 09:43:33:537 | 1254 | 829 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:283 | 09:43:33:188 | 09:43:33:520 | 1237 | 905 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:302 | 09:43:33:144 | 09:43:33:530 | 1228 | 842 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:302 | 09:43:33:138 | 09:43:33:531 | 1229 | 836 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:302 | 09:43:33:131 | 09:43:33:532 | 1230 | 829 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:302 | 09:43:33:125 | 09:43:33:534 | 1232 | 823 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:303 | 09:43:33:093 | 09:43:33:542 | 1239 | 790 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:303 | 09:43:33:233 | 09:43:33:509 | 1206 | 930 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:303 | 09:43:32:468 | 09:43:32:527 | 224 | 165 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:303 | 09:43:33:227 | 09:43:33:511 | 1208 | 924 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:304 | 09:43:32:409 | 09:43:32:762 | 458 | 105 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:305 | 09:43:33:086 | 09:43:33:544 | 1239 | 781 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:306 | 09:43:33:065 | 09:43:33:548 | 1242 | 759 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:306 | 09:43:32:489 | 09:43:33:554 | 1248 | 183 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:318 | 09:43:33:119 | 09:43:33:535 | 1217 | 801 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:318 | 09:43:33:163 | 09:43:33:525 | 1207 | 845 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:330 | 09:43:33:156 | 09:43:33:527 | 1197 | 826 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:330 | 09:43:32:515 | 09:43:32:988 | 658 | 185 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:334 | 09:43:32:515 | 09:43:32:982 | 648 | 181 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:334 | 09:43:33:106 | 09:43:33:539 | 1205 | 772 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:334 | 09:43:32:515 | 09:43:32:976 | 642 | 181 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:335 | 09:43:33:272 | 09:43:33:501 | 1166 | 937 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:335 | 09:43:32:535 | 09:43:32:963 | 628 | 200 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:335 | 09:43:32:513 | 09:43:32:969 | 634 | 178 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:336 | 09:43:32:403 | 09:43:32:753 | 417 | 67 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:359 | 09:43:33:331 | 09:43:33:488 | 1129 | 972 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:360 | 09:43:33:298 | 09:43:33:495 | 1135 | 938 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:360 | 09:43:32:524 | 09:43:33:020 | 660 | 164 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:360 | 09:43:32:505 | 09:43:33:026 | 666 | 145 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:360 | 09:43:33:207 | 09:43:33:515 | 1155 | 847 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:361 | 09:43:33:240 | 09:43:33:508 | 1147 | 879 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:361 | 09:43:32:523 | 09:43:33:013 | 652 | 162 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:361 | 09:43:33:253 | 09:43:33:505 | 1144 | 892 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:361 | 09:43:33:258 | 09:43:33:504 | 1143 | 897 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:362 | 09:43:33:151 | 09:43:33:528 | 1166 | 789 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:362 | 09:43:33:201 | 09:43:33:517 | 1155 | 839 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:362 | 09:43:33:265 | 09:43:33:502 | 1140 | 903 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:362 | 09:43:32:520 | 09:43:33:007 | 645 | 158 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:363 | 09:43:32:520 | 09:43:33:000 | 637 | 157 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:363 | 09:43:33:214 | 09:43:33:513 | 1150 | 851 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:363 | 09:43:32:499 | 09:43:33:038 | 675 | 136 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:363 | 09:43:32:501 | 09:43:33:045 | 682 | 138 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:364 | 09:43:33:246 | 09:43:33:507 | 1143 | 882 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:364 | 09:43:33:291 | 09:43:33:496 | 1132 | 927 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:364 | 09:43:32:516 | 09:43:32:994 | 630 | 152 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:365 | 09:43:33:278 | 09:43:33:499 | 1134 | 913 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:365 | 09:43:32:495 | 09:43:33:033 | 668 | 130 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:365 | 09:43:33:176 | 09:43:33:523 | 1158 | 811 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:369 | 09:43:33:285 | 09:43:33:498 | 1129 | 916 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:388 | 09:43:32:456 | 09:43:32:661 | 273 | 68 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:400 | 09:43:33:345 | 09:43:33:485 | 1085 | 945 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:407 | 09:43:33:304 | 09:43:33:493 | 1086 | 897 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:407 | 09:43:32:460 | 09:43:32:611 | 204 | 53 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:418 | 09:43:32:464 | 09:43:32:562 | 144 | 46 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:418 | 09:43:32:570 | 09:43:32:957 | 539 | 152 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:428 | 09:43:32:567 | 09:43:32:951 | 523 | 139 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:442 | 09:43:32:571 | 09:43:32:944 | 502 | 129 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:465 | 09:43:33:337 | 09:43:33:486 | 1021 | 872 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:465 | 09:43:32:583 | 09:43:32:938 | 473 | 118 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:476 | 09:43:33:317 | 09:43:33:490 | 1014 | 841 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:477 | 09:43:33:351 | 09:43:33:484 | 1007 | 874 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:485 | 09:43:33:311 | 09:43:33:492 | 1007 | 826 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:499 | 09:43:32:597 | 09:43:32:931 | 432 | 98 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:508 | 09:43:33:324 | 09:43:33:489 | 981 | 816 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:519 | 09:43:33:358 | 09:43:33:482 | 963 | 839 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:528 | 09:43:32:733 | 09:43:32:848 | 320 | 205 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:538 | 09:43:33:364 | 09:43:33:481 | 943 | 826 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:555 | 09:43:32:627 | 09:43:32:925 | 370 | 72 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:555 | 09:43:33:378 | 09:43:33:478 | 923 | 823 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:566 | 09:43:33:371 | 09:43:33:480 | 914 | 805 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:567 | 09:43:33:407 | 09:43:33:473 | 906 | 840 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:574 | 09:43:33:415 | 09:43:33:472 | 898 | 841 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:577 | 09:43:33:385 | 09:43:33:477 | 900 | 808 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:577 | 09:43:32:666 | 09:43:32:919 | 342 | 89 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:583 | 09:43:33:392 | 09:43:33:476 | 893 | 809 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:586 | 09:43:33:399 | 09:43:33:474 | 888 | 813 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:593 | 09:43:32:724 | 09:43:32:861 | 268 | 131 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:597 | 09:43:32:731 | 09:43:32:836 | 239 | 134 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:604 | 09:43:32:674 | 09:43:32:913 | 309 | 70 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:607 | 09:43:33:449 | 09:43:33:465 | 858 | 842 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:612 | 09:43:33:457 | 09:43:33:463 | 851 | 845 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:618 | 09:43:33:443 | 09:43:33:466 | 848 | 825 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:624 | 09:43:32:688 | 09:43:32:898 | 274 | 64 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:630 | 09:43:32:714 | 09:43:32:875 | 245 | 84 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:639 | 09:43:33:422 | 09:43:33:470 | 831 | 783 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:650 | 09:43:33:435 | 09:43:33:467 | 817 | 785 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:659 | 09:43:32:680 | 09:43:32:907 | 248 | 21 |
| Client[1] | file34.txt | Server[7] | Server[10] | Server[7] | 09:43:32:664 | 09:43:33:429 | 09:43:33:469 | 805 | 765 |

***Table 2.*** Type of the Data Collected during Execution Process

According to our presets, there is one client who has requested the same file 100 times and all requests are issued from lock manager server 7. Since we adopt the approach of calculating communication cost among servers, Started Timestamp refers to initiating process time from lock manager server 7 and Granted Timestamp refers to the received success reply from lock manager server 10. Time until granted is given as a difference between Granted Timestamp and Started Timestamp. Finished Timestamp refers to file execution time according to system time. Exec Time represents the sum of Time until Granted and additional time consumed to read/write the file from the client.

# CHAPTER 6

# EXPERIMENTAL RESULTS AND DISCUSSION

## 6.1 A Comparative Analysis between Centralized and Distributed Models

In chapter 5 we explained the implementation of the algorithm in the real setting and the ways of collecting data that we are going to analyze herein. The lock manager algorithm maintains the concurrent access in the shared files through three different approaches. These approaches presented here are: a. Self-management of shared locks in server, b. owner server management, c. third server lock management. According to [2] once written, files are only read, and it is certain that a high probability exists that read/write requests are issued from the server who created the file. Considering this fact, our approach provides a sustainable solution and avoids all other communication with other servers in cloud. Nevertheless, there will be many other requests delivered in different servers requesting the right attribution of a certain file. This approach leads to another state where the server which receives the request has to collaborate with other servers to maintain file consistency. For the simplicity of our work we call it initiator server.

According to our algorithm definition , at any time that the initiator server is not the owner server or does not provide the necessary information about a certain file requested by a specific client, it has to collaborate with other servers to get further information about the state of a certain file. This extra communication will add a delay that varies mostly in the

The main contribution of this chapter appears in the papers [70], [104].

speed of the network and in the amount of required communication. We have taken in consideration these delays and herein we have analyzed two states of the algorithm. Through all this study we will refer to this delay as time until execution permissions granted. In the first state we implement the first approach of the self-management of the shared lock and in the second state the request is randomly received by all servers. In addition to that, we are going to examine and compare the performance of both states. For both scenarios we keep the same cluster configuration and define that clients deliver 100 requests for accessing the same file.

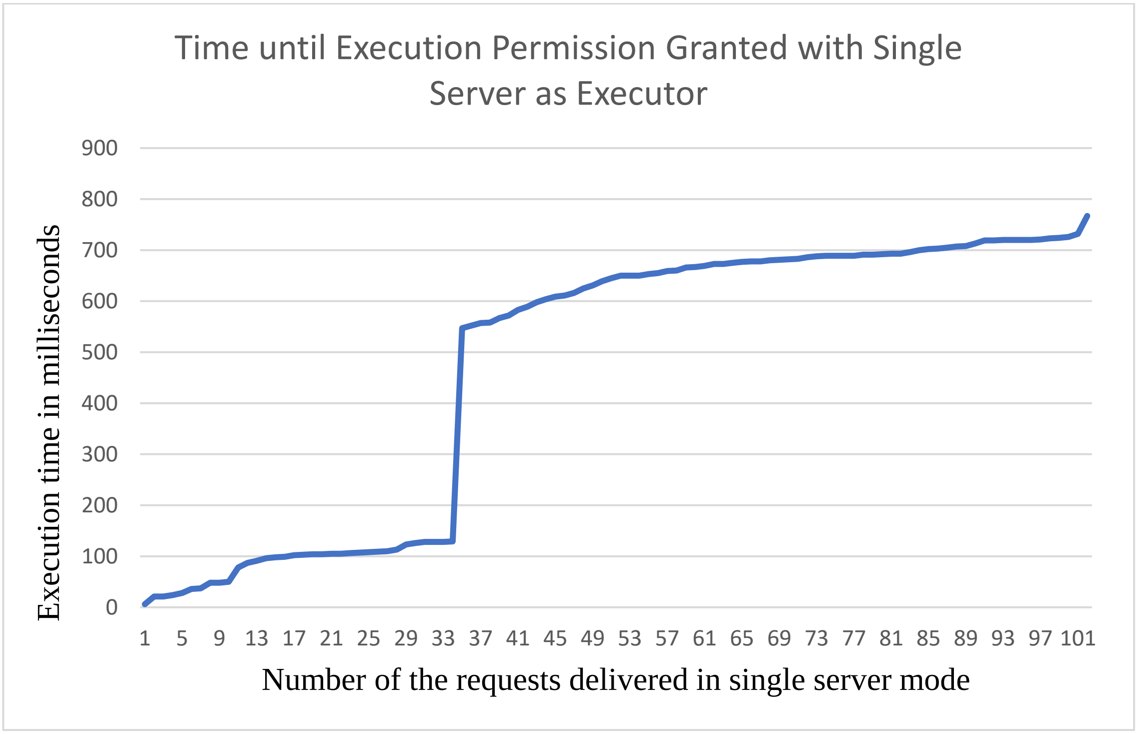


***Figure 32.*** Time until execution permissions granted in milliseconds for 100 requests in single server mode

In Fig. 32 we get the performance of execution of 100 requests by a single server[104]. The ratio between read/write requests and the order is arbitrarily decided by the algorithm. According to our presets, all the requests are delivered to a single file by the same server.

In this example we have simulated the scenario when the initiator server is the owner server for that specific file and no further communication with other servers is required. As seen in the graph in Fig. 32, we can denote three parts that require further analysis. For the first 30 requests the time until execution permission granted increases almost linearly but remains in lower values.

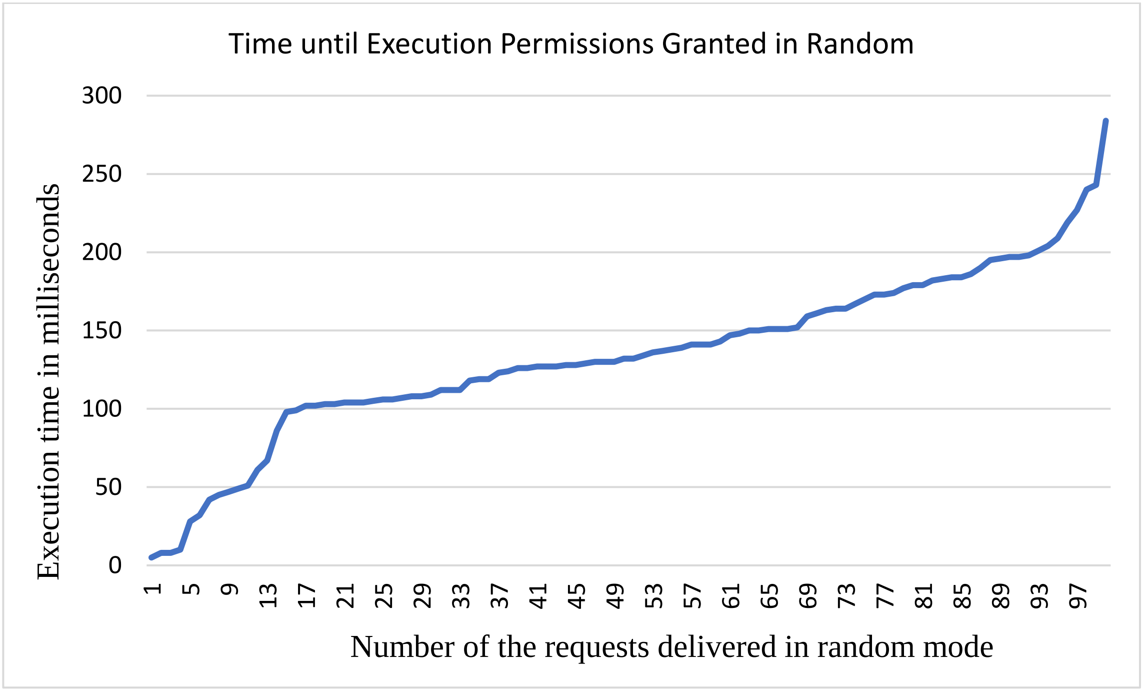


***Figure 33.*** Time until execution permissions granted in milliseconds for 100 requests in random mode

The maximum time reaches the value of 130 milliseconds. At the same time, this is the part where all requests delivered are read operation and they can be executed in parallel. At request 34, we denote a vertical growth which is explained by the fact that this is the point where it needs to wait for all read operations in queue to finish and after that to apply write operations. This is the point that represents the major delay that our algorithm adds. The rest of requests are write requests and again time increases almost linearly but in contrast with the read operation, now the time needed is higher since write operations cannot be executed in parallel. In the second approach, again clients delivered 100 requests but now in contrast with the first mode, all requests are delivered in random mode to different servers. As shown in Fig. 33, in random mode, the overall time until execution permissions granted is less than half of the time until execution permissions granted from single server, and even though we do not have a proper analysis of the percentage of read/write requests, we still get a better performance compared to single server mode. One point to be taken into account is that we have analyzed the situation when the number of requests is equal for both cases and compared to the real situation in a real cloud computing, there is a low number of requests. In the next sections we are going to analyze the situation when the number of requests changes from 100 to 500 requests and issue a comparison for each of the cases to check the behavior of our algorithm when applied in different clouds dimensions.

## 6.2 Performance Evaluation of the Asymmetric Distributed Lock Management in Cloud Computing

In chapter 5 we explained the main concepts that ADLMCC uses to maintain consistency in distributed systems that are realized in three different approaches which consist of: a. Self-management of shared locks in server, b. owner server management, c. remote server lock managements. This approach implies a sustainable solution and avoids all other communication with other servers in the cloud.

Nevertheless, there will be many other requests delivered to different servers requesting the attribution rights of a certain file. This new condition leads to another state where the server receiving the request has to collaborate with other servers to maintain file consistency. For the simplicity of our work we call this imitator server. The amount of communication between servers plays the biggest role in the system performance, adding a delay that will differ based on the communication media and the number of the requests at a certain time. Many cloud solutions are developed according to the type of the service they provide. It is essential for developers to find the equilibrium between resources and purpose of usage. In this chapter we measured the sustainability, delay and performance of the lock manager algorithm and analyzed its behavior while implementing it in big data clouds, mid-size and small size clouds.

### 6.2.1 Cloud Resources Effect in Asymmetric Distributed Lock Management in Cloud Computing

According to the architecture of the lock manager, the delay added in self-management is zero, there is no communication with other servers and the execution time depends only on the number of the requests that the server itself receives.

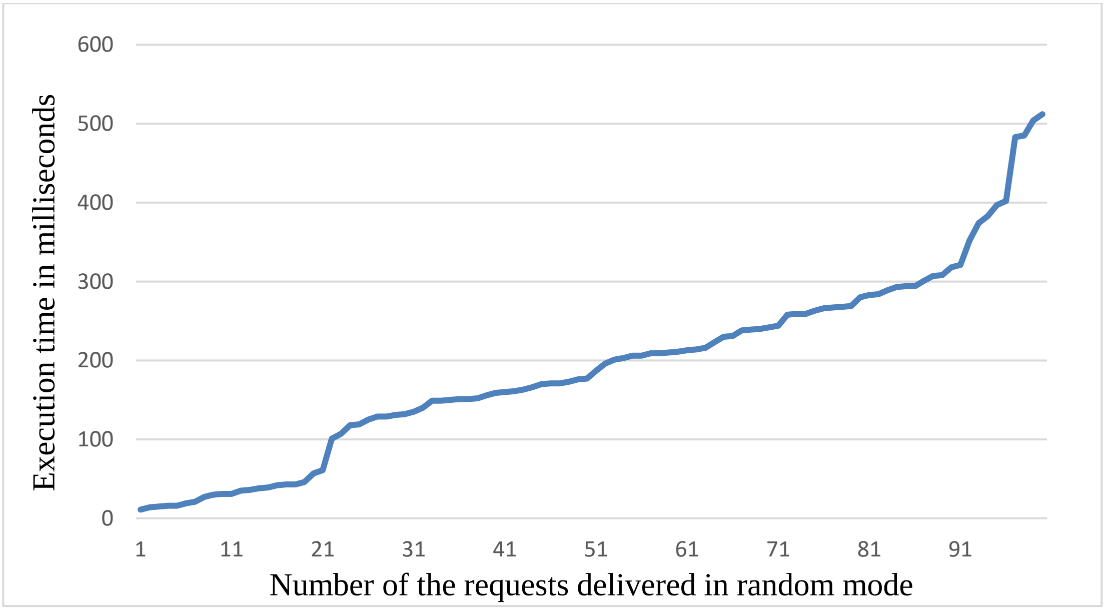


***Figure 34.*** Time until Granted for 100 requests in random with 5 servers

The biggest concern is the additional delay which occurred during the communication to find the state of file and to decide which servers are responsible for executing that request. We refer to this delay as Time until Granted and it is the focus of our analysis.  In the first test we defined the number of requests as a benchmark and we kept it unchanged throughout the process, alternating the number of servers, nodes and files stored on these nodes. Throughout our discussion, the requests were delivered asymmetrically to the servers and were delivered at the same time and randomly. Fig. 34 shows the Time until Granted for 100 requests delivered in random mode and the cloud is composed of 5 servers, 3 nodes, 30 files and 5 clients.

The cloud resources composition clearly demonstrates that this is an example of a small sized cloud which is composed of a few servers and nodes. Bearing in mind the number of requests, every client executes more than one request and a file is requested from more than one client with a different operation mode. Because of the small number of requests, we realize that the Time until Granted is very insignificant. The times differ from almost zero to the highest delay of 600 milliseconds. For most of the requests Time until Grated is less than 300 milliseconds. As the number of requests increases, we note that the graph encompasses a nonlinear change of the Time until Granted.

Following the same explanation, in Fig. 35 we illustrate the example with 100 requests delivered randomly and the cloud composition now is made up of 10 servers, 10 nodes, 300 files and 50 clients.

With the increase in the number of servers and files, the probability for requests to contact the same file is reduced. Due to that, we can observe from the graph that the overall Time until Granted for executing 100 requests is reduced by a half. In line with the first example, with the increase in the number of requests, the time that a request has to wait starts growing. In contrast to the first example, the increase is much more linear and now the total Time until Granted is only 300 milliseconds

We continued to increase the number of resources for the same number of requests. Now the number of servers is 30 and the number of other resources is 20 nodes, 1000 files and 200 clients. With the new changes, the cloud resources extend so that the number of clients and the files stored in nodes is much bigger than the total number of requests. The probability that a client delivers more than one request and that two requests access the same file is almost zero.

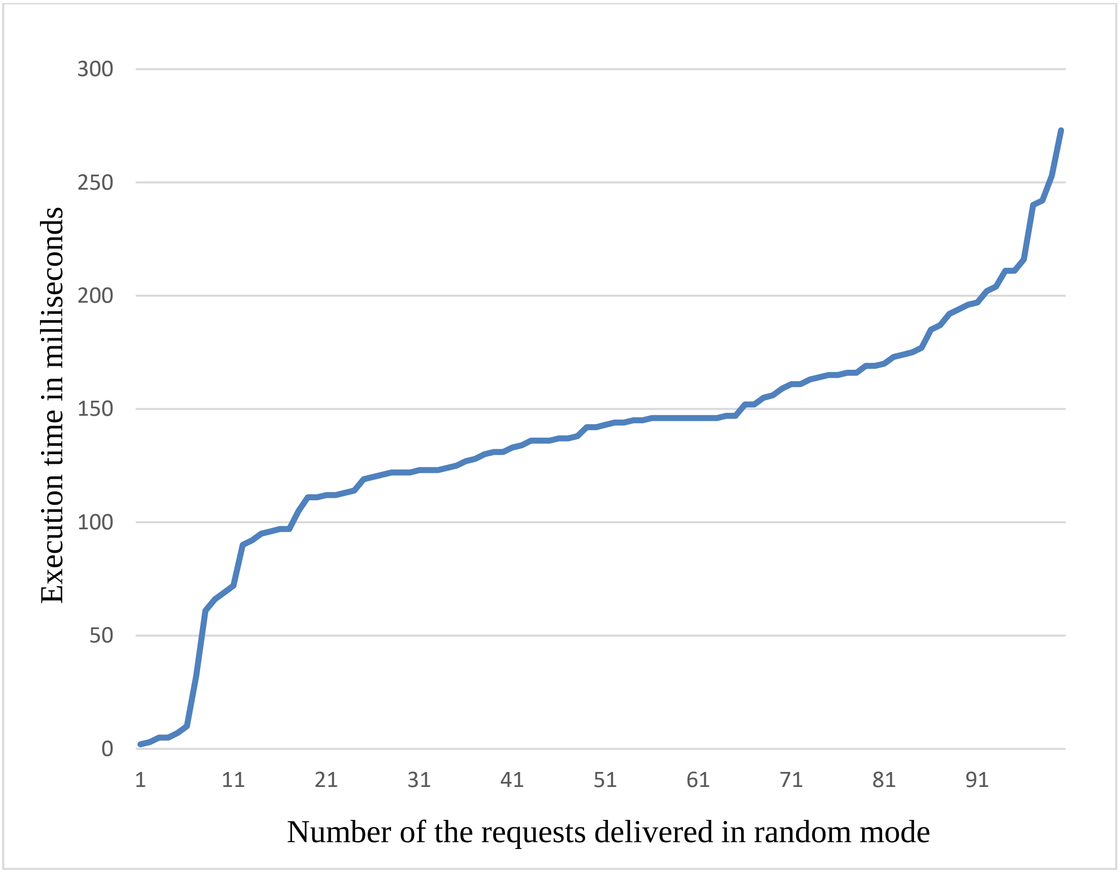


***Figure 35.*** Time until Granted for 100 requests in random with 10 servers

The obtained results, as illustrated in Fig. 36 indicate that the overall Time until Granted continues to decrease, and the graph curve is almost linear. The Time until Granted is less than 250 milliseconds and there is no moment in which the time rises very quickly; everything is smooth with no specific moment to be investigated.

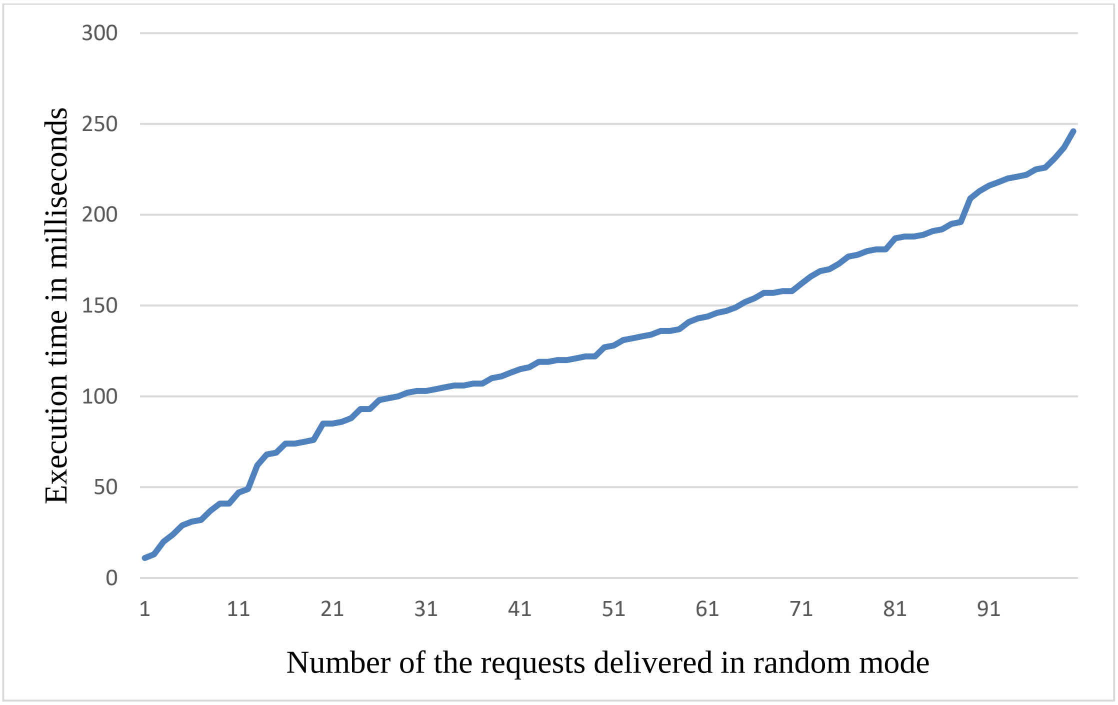


***Figure 36.*** Time until Granted for 100 requests in random with 30 servers

To better understand the relationship between physical resources and the number of requests to be executed, in Fig. 37 we present the conjunction of three graphs. The blue, green and purple curves represent Time until Granted for executing 100 requests delivered in random mode to respectively 10, 5 and 30 servers.

Following our discussion, we observe from graph that the explanation given for Fig. 37 becomes more explicit. Since blue and purple curves conjunctions are almost similar with minor changes, the increase in the number of servers from 10 to 30 has not played a significant role on the performance of systems. However, the low number of servers has a major impact on performance and referring to the same graph, we can perceive reducing the number of servers from 10 to 5 servers per cluster has enlarged the Time until Granted from less than 300 milliseconds to above 500 milliseconds, adding an additional 200 milliseconds.

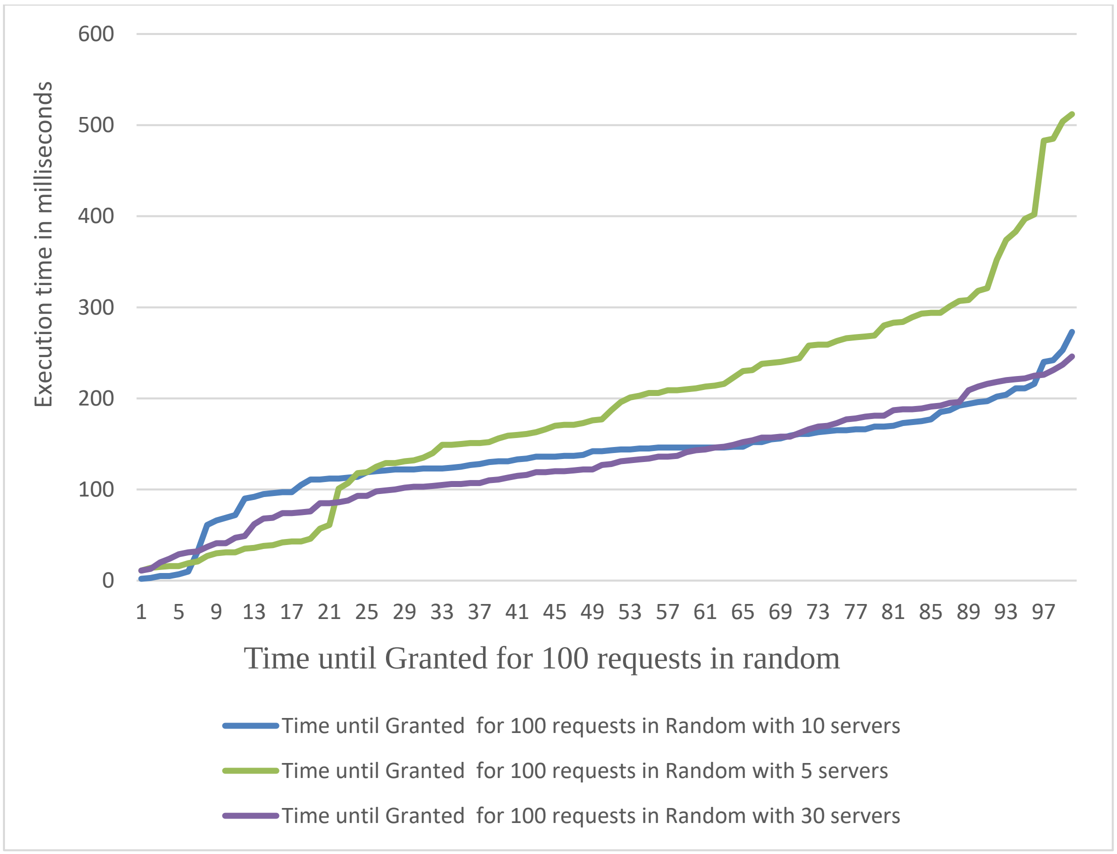


***Figure 37.*** Time until Granted for 100 requests in random

### 6.2.2 Number of Requests Effect in Asymmetric Distributed Lock Management in Cloud Computing

In comparison with the first implementation, to better understand how Time until Granted differs according to the cloud resources and its usage, we conducted the same study only by modifying the parameter of our benchmark. Now the number of the requests has changed from 100 to 500 requests. The cloud is composed of 5 servers, 3 nodes and 30 files. All the requests for file are delivered to only 5 servers. As per the first analysis, which took into account the cloud composition, there was a high probability that a high number of requests were delivered by the same client and a high probability that different requests ask to read/write the same file.

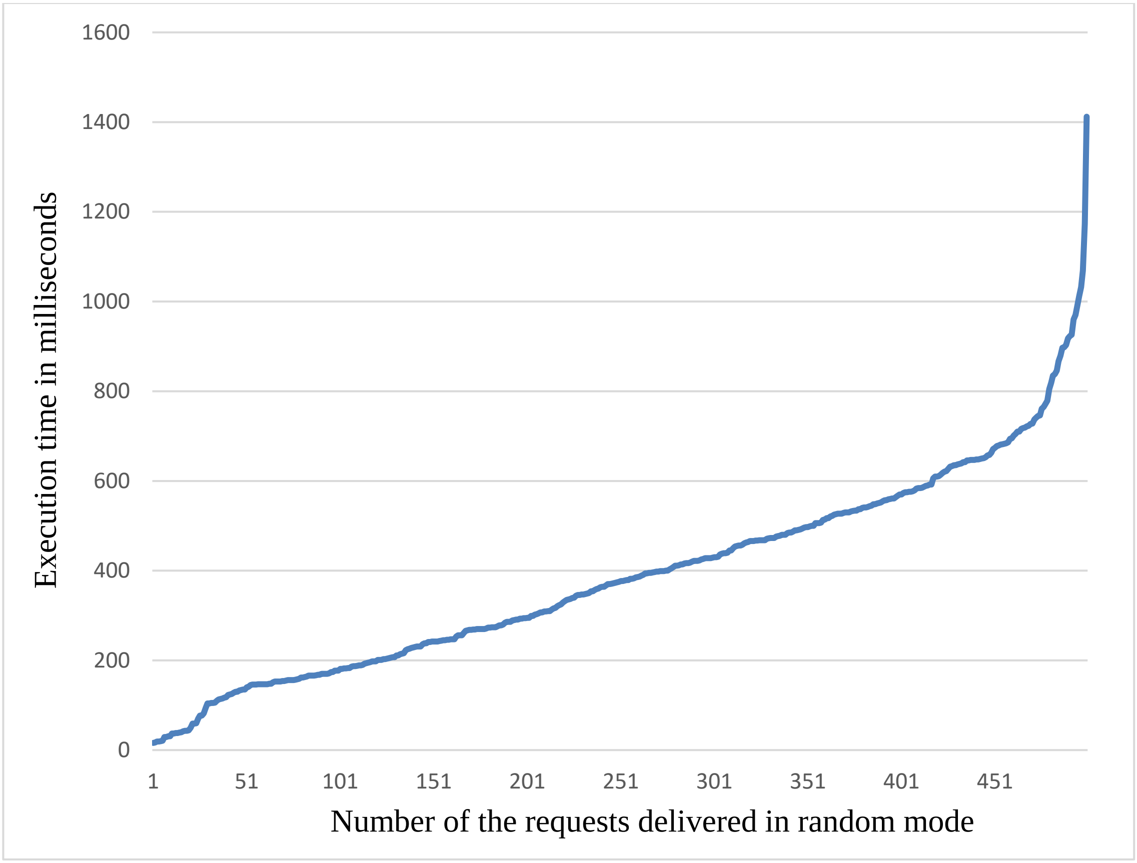


***Figure 38.*** Time until Granted for 500 requests in random with 5 servers

In Fig. 38 we have given the variance of Time until Granted in milliseconds for 500 requests. It is clearly depicted that with the growth of the total number of requests, the overall Time until Granted increases. The graph is almost linear for most of the time and after a certain request it changes exponentially. It is notable from the graph that until 450 requests the time increases linearly and the maximum time for a request to wait is nearly

650 milliseconds, after this point time it goes up very quickly and at request number 500 the Time until Grated goes to 1400 milliseconds.

At this point, from a change of 50 requests we denote a change of time that varies from 650 to 1400 milliseconds. As for the first corresponding tests, the curves changes are almost similar to each other's. Another aspect that will affect the curve linearity is the type of request delivered as the more write requests are delivered first the more Time until granted is needed.

One remark to be mentioned from Fig. 38 is that for the given composition of cloud resources, after a certain number of requests the time increases very rapidly. Following the same explanation, we keep the number of requests at 500 and we change the number of servers to 10, while the number of other resources becomes 10 nodes, 300 files and 50 clients. Still, there will be a high probability that every client delivers more than one request and a high probability that many requests ask to read/write the same file.

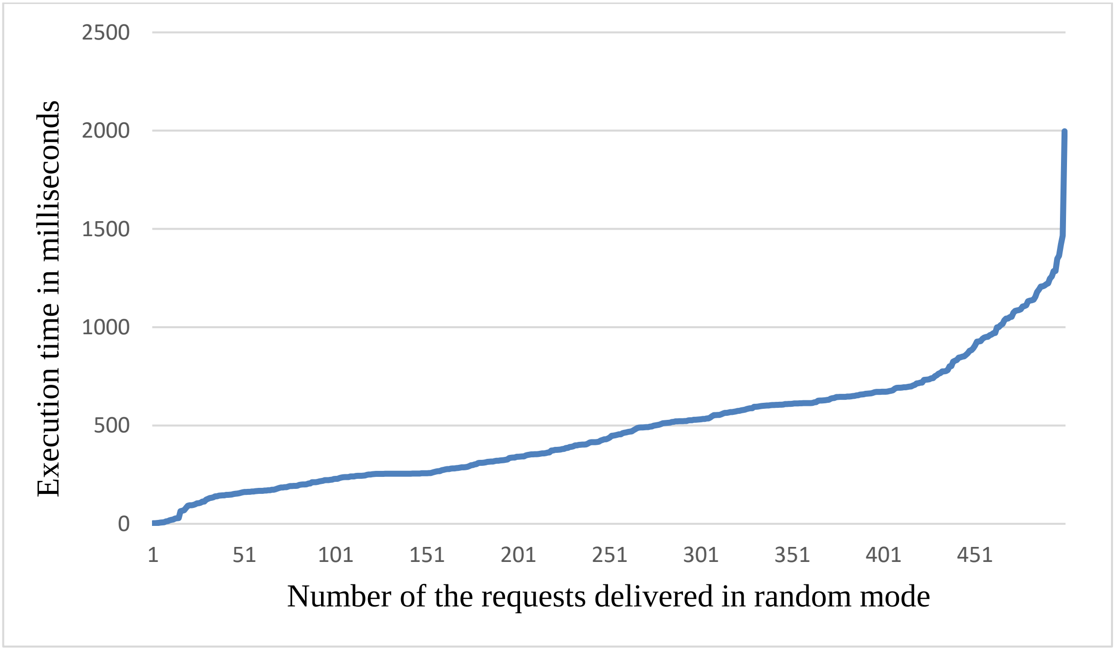


***Figure 39.*** Time until Granted for 500 requests in random with 10 servers

The graph in Fig. 39 illustrates how the Time until Granted changes for the execution of 500 requests in the new settings.

As notable from the graphs above, there is a similarity between Fig. 38 and Fig. 39. The increase in the resources did not provide any improvement in the overall performance. After a certain number of requests, the linearity of the graph breaks and the Time until Grated rises exponentially. For the same number of requests, we increased the number of servers to 30 and the number of the other resources becomes 20 nodes, 1000 files and 200 clients. Within the new configuration, the probability that a client delivers more than one request and that the same server receives more than one request from clients remains significant.

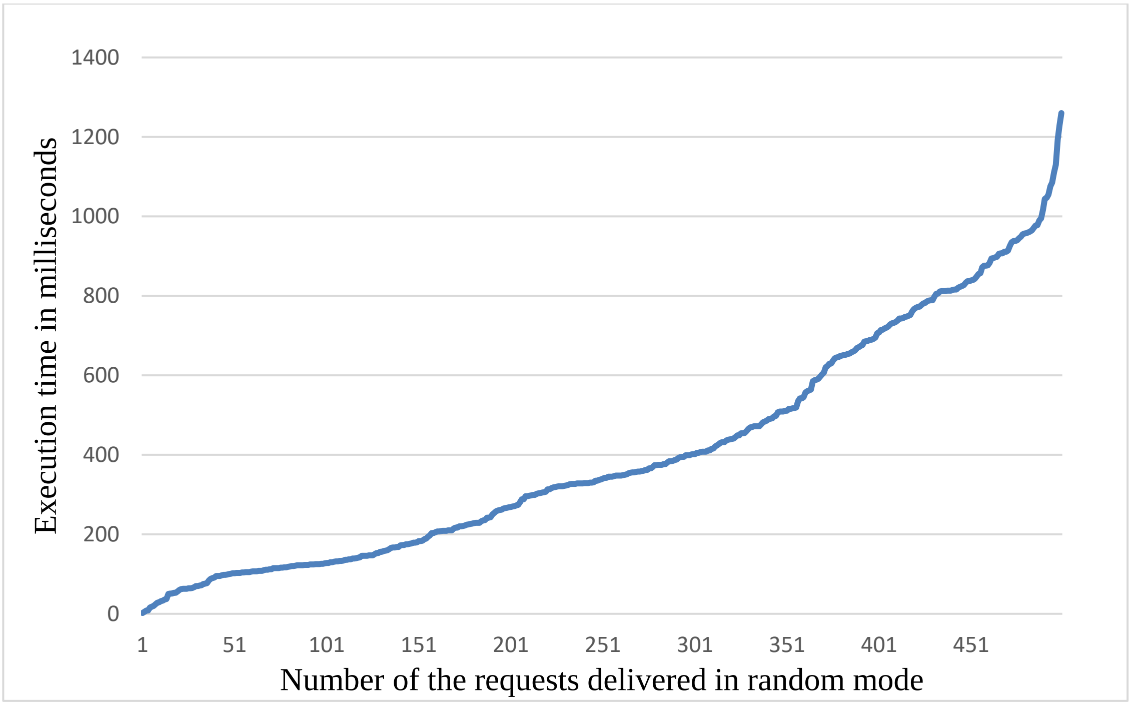


***Figure 40.***Time until Granted for 500 requests in random with 30 servers

Considering the number of files, the probability that the same file is requested from more than one request decreases. The new results are described in Fig. 40, which explains how the Time until Granted varies for executing 500 requests with 30 servers.

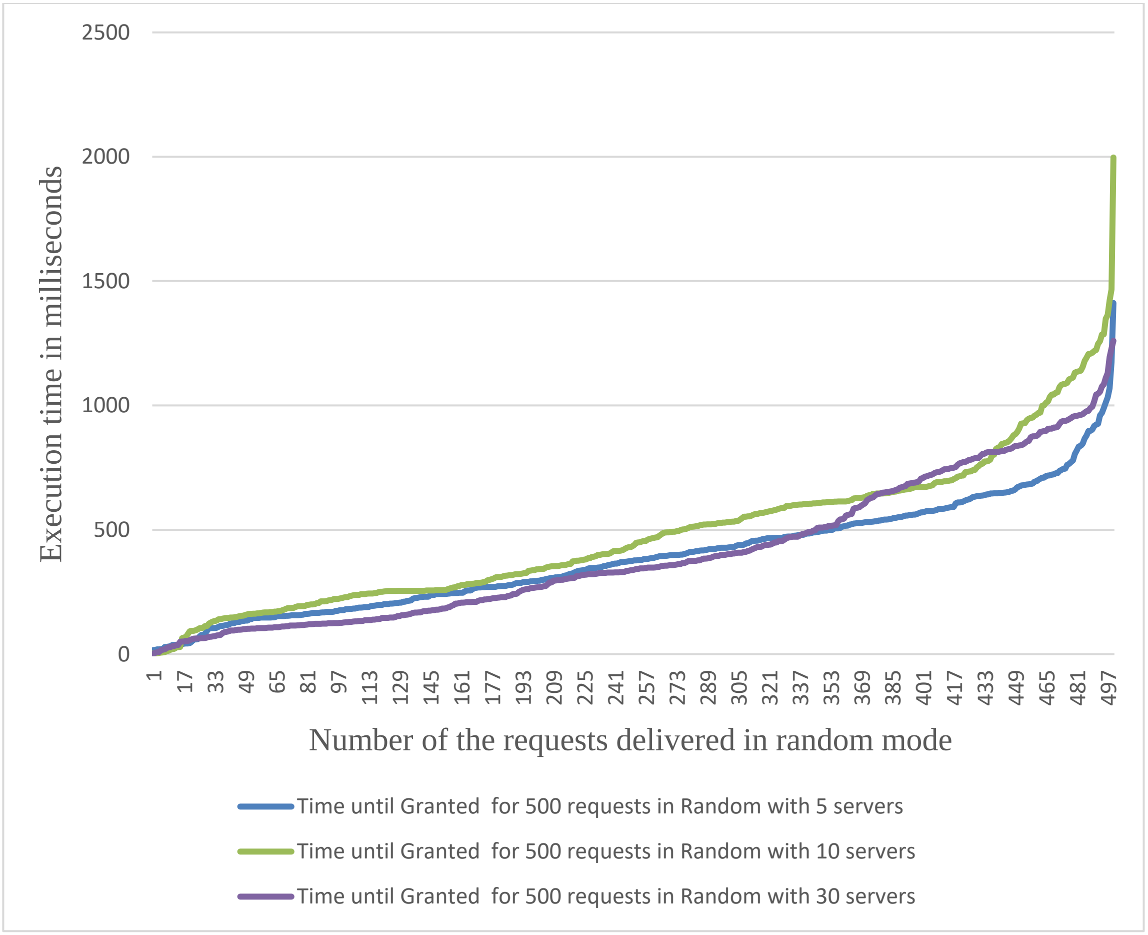


***Figure 41.*** Time until Granted for 500 requests in random

With the increase in resources, even though there still is a high possibility for each client to deliver more than one request and for each server to execute more than one request, we can denote that the overall Time until Granted has decreased. The linearity remains almost unchanged for the entire process. We can note that in this new state equilibrium has been achieved between the amount of resources and the number of the requests. If more resources are added, the more Time until Granted will decrease and if more numbers of requests are delivered, more time is required for the overall execution.

The graph curve connections presented in Fig. 41 illustrate our discussion and support our observation that increasing cluster resources affects the overall cloud performance even though effect moves toward diminishing. Performance evaluation and resource optimization are very important tasks and require the attention of developers.

## 6.3 Effect of Resource Starvation in Asymmetric Distributed Lock Management in Cloud Computing

In sections 4.3.1 and 4.3.2, we have redesigned our lock manager algorithm functionalities to explain and introduce the new concept of resource starvation. With the new design, the execution behavior will change, and a new equilibrium is required for maintaining the overall cloud performance.

In this section we are going to analyze the data collected from test held with our algorithm simulating the situation where the request is executed from remote lock manager. This simulation refers to the algorithm status that is required for the initiator servers to migrate the request to a remote server which has been already granted the execution permission right from owner lock manager. The simulations will be held on the same environment conditions when we apply resource starvation parameter to the lock manager and for the condition that no resource starvation parameter is applied.

The platform is composed of 10 servers, 10 node storages, 300 files, 50 clients and for both simulations the number of requests remains unchanged at 200 requests. For performing the tests, we have defined in presets that all the requests are delivered to server 10 and they ask the same file that is under ownership of server 4. Server 4 has already migrated file permission right to server 9. The same test with the same presets is issued two times; the first time without defining resource starvation parameter and the second time by adjusting the algorithms settings in accordance with the new improvements adding the starvation parameter.

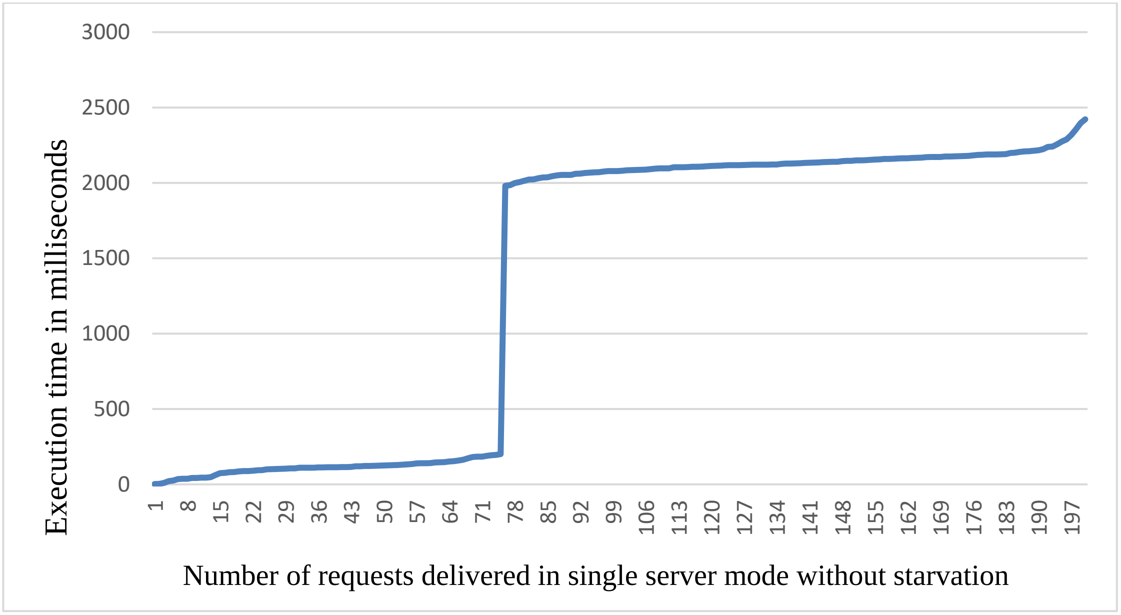


***Figure 42.*** Execution time in milliseconds for 200 requests in single server mode without Starvation

The graph in Fig. 42 shows time until Execution Permission granted with single server as executor without starvation. According to the curve of the graph, we can denote three parts that require special attention. In the first part of the graph we see that the execution time is very low. In the second part, we have a vertical increase in time, and in the third part again we have a stability of the execution process. This is related to the type of requests that have been executed. In the first part we have to deal with read requests that are executed parallel to each other. Considering the fact that read-write, write-write, write-read cannot be executed simultaneously, the vertical increase represents the moment when write request is waiting for read requests to finish and then start their execution. Again, we have a stability of the graph that represents the moment when the same type of requests is executed one after another. The maximum time until Granted for 200 requests reached the value of 2500 milliseconds which is quite a significant value that needs to be taken in consideration. Following the same assumption, if the number of requests continues to be increased, then server 9 will undergo in exhausted mode and no more requests will be processed.

In the second simulation we performed, as shown in Fig. 43, we denoted a starvation parameter that has prohibited server 9 from entering exhausted mode.

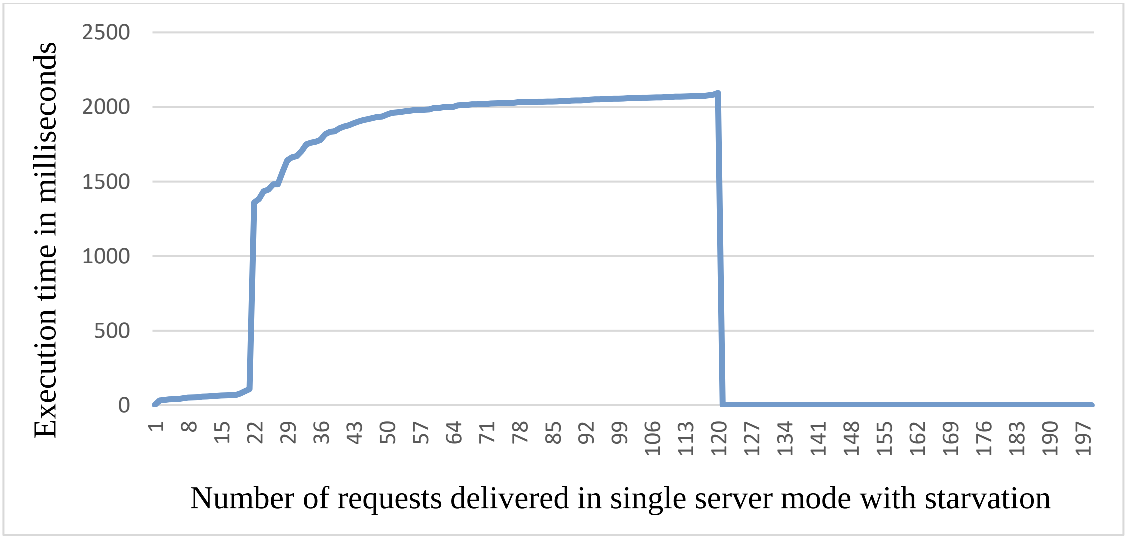


***Figure 43.*** Execution time in milliseconds for 200 requests in single server mode with Starvation

As seen from Fig. 43, when the server receives a certain number of requests, it automatically discards them. For the simplicity of our work, the discarded request is presented in the graph with value 0. The other part of the graph remains unchanged as in Fig. 42, with only a slight difference that happens because of the moment change when write requests are delivered. The resource starvation parameter can be adjusted from provider to provider and will be mostly dependent on the hardware parameters of the servers in use and the sensitivity of the running services.

# CHAPTER 7

## 7.1 CONCLUSIONS

In this thesis, we proposed an algorithm that provides a mechanism to maintain simultaneous read, write requests for accessing distributed data that are stored using erasure codes technique. The algorithm provides a solution to manage a large number of requests in a distributed system in such a way that they can be completed within the least possible time.

Lock manager algorithm introduced in this thesis gives a sustainable solution to the main aspects for a reliable cloud, by offering high availability and scalability while keeping the stored data free from errors. The algorithm prevents concurrent access in shared files by excluding simultaneous read-write and write-write operations to the same file.

Another characteristic of our approach is that the shared files are kept in a proper way that they are free from the errors and fault tolerant from the server failures. For each file we design a responsible server and a switchover server to be used in case of responsible server failure.

Based on algorithm design and functionalities, lock manager gives solution to the main concerns for a reliable cloud such as availability and scalability. Considering that each server acts as a master server for the received requests, the availability of the cloud is maintained in a distributed manner and we avoid the existence of a master server. The number of servers can be modified in accordance with cloud usage requirements without affecting the overall cloud performance.

Another aspect of our thesis has been to analyze the performance of ADLMCC when it is implemented in different cloud environments. The lock manager is the solution for the implementation of different cloud storages.

Based on its implementation characteristics, asymmetric distributed lock management in cloud computing algorithm is a fully distributed solution and eliminates the need for a master node which is going to control and maintain the consistency of shared files. Our approach implements most of the key factors for a cloud to be a reliable and fault tolerant cloud. On the other hand, it keeps the file consistency with the least possible communication among other colleagues. Communication with other lock managers happens only when extra information is needed.

One of the factors that define the quality of the service for our solution is the time to wait for a client until its request is processed. This time is determined from the communication between lock managers for the consistency of the file that has been named as Time until Granted, and from the availability of the requested files. Time until Granted was the focus of this thesis and was analyzed in detail.

According to our test results, we can conclude that Time until Granted is low for a normal load implemented in small and mid-size clouds and becomes significant when the number of requests is increased. Another conclusion achieved from our analysis is that, with the increase in physical resources, there is also an increase in the performance of the cloud and the Time until Granted decreases significantly. Time until Granted depends more on the type of the request. For any increase in the delivered write requests more time is required for the execution process, while for read request less time is required.

The lock manager algorithm offered is a sustainable solution and it is suitable to be implemented in cloud storages platforms that require strong consistency and high precision of the data modification.

Following the outcome from our results it can be concluded that when many requests are directed to the same server, this might lead to server resource exhaustion and service interruption. To maintain server availability, we have defined a parameter called resource starvation which is responsible for maintaining the availability of the resources. According to the test results and analysis, we conclude that the definition of such a parameter is essential and in case that the proper attention is not paid, the server can be exhausted and can commit server failure.

The algorithms proposed can be applied to any cloud storage, but we must keep in mind and analyze client service requirements. If starvation point is reached some requests will get lost and need to be reissued; this can decrease availability of the systems when the same file is subject to be changed from many users.

This study recommends that further research focus on the use of switchover as load balancer for mitigating the resource starvation effect.

# REFERENCES

1 i Juárez, L.P.: 'On the Design and Optimization of Heterogeneous Distributed Storage Systems', University Rovira in Virgili, Department of Engineering Information in Mathematic, PHD thesis, 2011

2 Ghemawat, S., Gobioff, H., and Leung, S.-T.: 'The Google file system', SIGOPS Oper. Syst. Rev., 2003, 37, (5), pp. 29-43

3 Maurya, M., and Mahajan, S.: 'Performance analysis of MapReduce programs on Hadoop cluster', in Editor (Ed.)^(Eds.): 'Book Performance analysis of MapReduce programs on Hadoop cluster' (IEEE, 2012, edn.), pp. 505-510

4 Borthakur, D.: 'The hadoop distributed file system: Architecture and design', Hadoop Project Website, 2007, 11, (2007), pp. 21

5 Koçi, A., and Çiço, B.: 'Storage Based Cloud Computing and Disaster Recovery', 'Storage Based Cloud Computing and Disaster Recovery' (edn.), pp. 395

6 Lamport, L.: 'Paxos made simple', ACM Sigact News, 2001, 32, (4), pp. 18-25

7 Lamport, L.: 'The part-time parliament', ACM Transactions on Computer Systems (TOCS), 1998, 16, (2), pp. 133-169

8 Adya, A., Bolosky, W.J., Castro, M., Cermak, G., Chaiken, R., Douceur, J.R., Howell, J., Lorch, J.R., Theimer, M., and Wattenhofer, R.P.: 'FARSITE: Federated, available, and reliable storage for an incompletely trusted environment', ACM SIGOPS Operating Systems Review, 2002, 36, (SI), pp. 1-14

9 Tewari, S., and Kleinrock, L.: 'Analysis of search and replication in unstructured peer-to-peer networks', in Editor (Ed.)^(Eds.): 'Book Analysis of search and replication in unstructured peer-to-peer networks' (ACM, 2005, edn.), pp. 404-405

10 Ko, A.C., and Zaw, W.T.: 'Fault Tolerant Erasure Coded Replication for HDFS Based Cloud Storage', in Editor (Ed.)^(Eds.): 'Book Fault Tolerant Erasure Coded Replication for HDFS Based Cloud Storage' (IEEE, 2014, edn.), pp. 104-109

11 Sathiamoorthy, M., Asteris, M., Papailiopoulos, D., Dimakis, A.G., Vadali, R., Chen, S., and Borthakur, D.: 'Xoring elephants: Novel erasure codes for big data', in

Editor (Ed.)^(Eds.): ‘Book Xoring elephants: Novel erasure codes for big data’ (VLDB Endowment, 2013, edn.), pp. 325-336

12 Dimakis, A.G., Ramchandran, K., Wu, Y., and Suh, C.: ‘A survey on network codes for distributed storage’, Proceedings of the IEEE, 2011, 99, (3), pp. 476-489

13 Dimakis, A.G., Godfrey, P.B., Wu, Y., Wainwright, M.J., and Ramchandran, K.: ‘Network coding for distributed storage systems’, IEEE transactions on information theory, 2010, 56, (9), pp. 4539-4551

14 Kishida, H., and Yamazaki, H.: ‘SSDLM: architecture of a distributed lock manager with high degree of locality for clustered file systems’, in Editor (Ed.)^(Eds.): ‘Book SSDLM: architecture of a distributed lock manager with high degree of locality for clustered file systems’ (IEEE, 2003, edn.), pp. 9-12

15 Lakshman, A., and Malik, P.: ‘Cassandra: a decentralized structured storage system’, ACM SIGOPS Operating Systems Review, 2010, 44, (2), pp. 35-40

16 Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., and Steere, D.C.: ‘Coda: A highly available file system for a distributed workstation environment’, IEEE Transactions on computers, 1990, 39, (4), pp. 447-459

17 Satyanarayanan, M.: ‘Coda: A resilient distributed file system’, in Editor (Ed.)^(Eds.): ‘Book Coda: A resilient distributed file system’ (1987, edn.), pp.

18 Koci, A.C., Betim: ‘DDCMCC - Distributed Data Consistency Management in Cloud Computing ’, International Scientific Conference Computer Science`2015, 2015, 1, pp. 200 - 206

19 Burrows, M.: ‘The Chubby lock service for loosely-coupled distributed systems’, in Editor (Ed.)^(Eds.): ‘Book The Chubby lock service for loosely-coupled distributed systems’ (USENIX Association, 2006, edn.), pp. 335-350

20 Reiher, P.L., Heidemann, J.S., Ratner, D., Skinner, G., and Popek, G.J.: ‘Resolving File Conflicts in the Ficus File System’, in Editor (Ed.)^(Eds.): ‘Book Resolving File Conflicts in the Ficus File System’ (1994, edn.), pp. 183-195

21 DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W.: ‘Dynamo: amazon's highly

available key-value store', in Editor (Ed.)^(Eds.): 'Book Dynamo: amazon's highly available key-value store' (ACM, 2007, edn.), pp. 205-220

22 Schmuck, F.B., and Haskin, R.L.: 'GPFS: A Shared-Disk File System for Large Computing Clusters', in Editor (Ed.)^(Eds.): 'Book GPFS: A Shared-Disk File System for Large Computing Clusters' (2002, edn.), pp.

23 Schollmeier, R.: 'A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications', in Editor (Ed.)^(Eds.): 'Book A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications' (IEEE, 2001, edn.), pp. 101-102

24 Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B.: 'Design and implementation of the Sun network filesystem', in Editor (Ed.)^(Eds.): 'Book Design and implementation of the Sun network filesystem' (1985, edn.), pp. 119-130

25 Anderson, T.E., Dahlin, M.D., Neefe, J.M., Patterson, D.A., Roselli, D.S., and Wang, R.Y.: 'Serverless network file systems', in Editor (Ed.)^(Eds.): 'Book Serverless network file systems' (ACM, 1995, edn.), pp. 109-126

26 Vazhkudai, S.S., Ma, X., Freeh, V.W., Strickland, J.W., Tammineedi, N., and Scott, S.L.: 'Freeloader: Scavenging desktop storage resources for scientific data', in Editor (Ed.)^(Eds.): 'Book Freeloader: Scavenging desktop storage resources for scientific data' (IEEE Computer Society, 2005, edn.), pp. 56

27 Placek, M., and Buyya, R.: 'A taxonomy of distributed storage systems', Reporte técnico, Universidad de Melbourne, Laboratorio de sistemas distribuidos y cómputo grid, 2006

28 Berretti, S., Thampi, S.M., and Dasgupta, S.: 'Intelligent systems technologies and applications' (Springer, 2016. 2016)

29 Oram, A.: 'Peer-to-Peer: Harnessing the power of disruptive technologies' (" O'Reilly Media, Inc.", 2001. 2001)

30 Subramanian, R., and Goodman, B.D.: 'Peer-to-peer computing: the evolution of a disruptive technology' (Igi Global, 2005. 2005)

31 Hasan, R., Anwar, Z., Yurcik, W., Brumbaugh, L., and Campbell, R.: 'A survey of peer-to-peer storage techniques for distributed file systems', in Editor (Ed.)^(Eds.): 'Book A survey of peer-to-peer storage techniques for distributed file systems' (IEEE, 2005, edn.), pp. 205-213

32 Albrecht, K., Arnold, R., and Wattenhofer, R.: 'Clippee: A large-scale client/peer system', Technical report/ETH, Department of Computer Science, 2003, 410

33 Tutschku, K.: 'A measurement-based traffic profile of the eDonkey filesharing service', in Editor (Ed.)^(Eds.): 'Book A measurement-based traffic profile of the eDonkey filesharing service' (Springer, 2004, edn.), pp. 12-21

34 Clarke, I., Sandberg, O., Wiley, B., and Hong, T.W.: 'Freenet: A distributed anonymous information storage and retrieval system', in Editor (Ed.)^(Eds.): 'Book Freenet: A distributed anonymous information storage and retrieval system' (Springer, 2001, edn.), pp. 46-66

35 Dingledine, R.: 'The free haven project: Design and deployment of an anonymous secure data haven', Massachusetts Institute of Technology, 2000

36 Muthitacharoen, A., Morris, R., Gil, T.M., and Chen, B.: 'Ivy: A read/write peer-to-peer file system', ACM SIGOPS Operating Systems Review, 2002, 36, (SI), pp. 31-44

37 Batten, C., Barr, K., Saraf, A., and Trepetin, S.: 'pStore: A secure peer-to-peer backup system', Unpublished report, MIT Laboratory for Computer Science, 2001, pp. 130-139

38 Milojičić, D., Llorente, I.M., and Montero, R.S.: 'Opennebula: A cloud management tool', IEEE Internet Computing, 2011, 15, (2), pp. 11-14

39 Haeberlen, A., Mislove, A., and Druschel, P.: 'Glacier: Highly durable, decentralized storage despite massive correlated failures', in Editor (Ed.)^(Eds.): 'Book Glacier: Highly durable, decentralized storage despite massive correlated failures' (USENIX Association, 2005, edn.), pp. 143-158

40 Druschel, P., and Rowstron, A.: 'PAST: A large-scale, persistent peer-to-peer storage utility', in Editor (Ed.)^(Eds.): 'Book PAST: A large-scale, persistent peer-to-peer storage utility' (IEEE, 2001, edn.), pp. 75-80

41 Demers, A., Petersen, K., Spreitzer, M., Terry, D., Theimer, M., and Welch, B.: ‘The Bayou architecture: Support for data sharing among mobile users’, in Editor (Ed.)^(Eds.): ‘Book The Bayou architecture: Support for data sharing among mobile users’ (IEEE, 1994, edn.), pp. 2-7

42 Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., and Hauser, C.H.: ‘Managing update conflicts in Bayou, a weakly connected replicated storage system’, in Editor (Ed.)^(Eds.): ‘Book Managing update conflicts in Bayou, a weakly connected replicated storage system’ (ACM, 1995, edn.), pp. 172-182

43 Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D.: ‘Epidemic algorithms for replicated database maintenance’, in Editor (Ed.)^(Eds.): ‘Book Epidemic algorithms for replicated database maintenance’ (ACM, 1987, edn.), pp. 1-12

44 Palankar, M.R., Iamnitchi, A., Ripeanu, M., and Garfinkel, S.: ‘Amazon S3 for science grids: a viable solution?’, in Editor (Ed.)^(Eds.): ‘Book Amazon S3 for science grids: a viable solution?’ (ACM, 2008, edn.), pp. 55-64

45 Ripeanu, M., Garfinkel, S., Palankar, M., and Iamnitchi, A.: ‘Amazon s3 for science grids: a viable solution?’, in Editor (Ed.)^(Eds.): ‘Book Amazon s3 for science grids: a viable solution?’ (edn.), pp.

46 Balmford, A., Crane, P., Dobson, A., Green, R.E., and Mace, G.M.: ‘The 2010 challenge: data availability, information needs and extraterrestrial insights’, Philosophical Transactions of the Royal Society of London B: Biological Sciences, 2005, 360, (1454), pp. 221-228

47 Schwarz, T.J.: ‘Availability in global peer-to-peer storage systems’, in Editor (Ed.)^(Eds.): ‘Book Availability in global peer-to-peer storage systems’ (Citeseer, 2004, edn.), pp.

48 Ramabhadran, S., and Pasquale, J.: ‘Analysis of durability in replicated distributed storage systems’, in Editor (Ed.)^(Eds.): ‘Book Analysis of durability in replicated distributed storage systems’ (IEEE, 2010, edn.), pp. 1-12

49 Utard, G., and Vernois, A.: 'Data durability in peer to peer storage systems', in Editor (Ed.)^(Eds.): 'Book Data durability in peer to peer storage systems' (IEEE, 2004, edn.), pp. 90-97

50 Tang, B., and Fedak, G.: 'Analysis of data reliability tradeoffs in hybrid distributed storage systems', in Editor (Ed.)^(Eds.): 'Book Analysis of data reliability tradeoffs in hybrid distributed storage systems' (IEEE, 2012, edn.), pp. 1546-1555

51 Horn, R.L.: 'System and method for maintaining a data redundancy scheme in a solid state memory in the event of a power loss', in Editor (Ed.)^(Eds.): 'Book System and method for maintaining a data redundancy scheme in a solid state memory in the event of a power loss' (Google Patents, 2013, edn.), pp.

52 Belhadj, M., Daniels, R.D., and Umberger, D.K.: 'Raid rebuild using most vulnerable data redundancy scheme first', in Editor (Ed.)^(Eds.): 'Book Raid rebuild using most vulnerable data redundancy scheme first' (Google Patents, 2003, edn.), pp.

53 Sheth, M., Benerjee, K.G., and Gupta, M.K.: 'Quorum sensing for regenerating codes in distributed storage', in Editor (Ed.)^(Eds.): 'Book Quorum sensing for regenerating codes in distributed storage' (IEEE, 2014, edn.), pp. 1-4

54 Gifford, D.K.: 'Weighted voting for replicated data', in Editor (Ed.)^(Eds.): 'Book Weighted voting for replicated data' (ACM, 1979, edn.), pp. 150-162

55 Gammie, P.: 'Roy Peter Van and Haridi Seif. Concepts, Techniques, and Models of Computer Programming. The MIT Press, 2004. ISBN: 0262220695 Price $70. 930pp', Journal of Functional Programming, 2009, 19, (2), pp. 254-256

56 Erb, B.: 'Concurrent programming for scalable web architectures', 2012

57 Al-Aaridhi, R., Yüksektepe, A., Amft, T., and Graffi, K.: 'Distributed data structures improvement for collective retrieval time', in Editor (Ed.)^(Eds.): 'Book Distributed data structures improvement for collective retrieval time' (IEEE, 2016, edn.), pp. 85-90

58 Amazon, E.: 'Amazon elastic compute cloud', Retrieved Feb, 2009, 10

59 Nachankar, V.: 'Distributed Lock Manager', Indiana University, 2011

60 Hansen, J.G., and Jul, E.: ‘Lithium: virtual machine storage for the cloud’, in Editor (Ed.)^(Eds.): ‘Book Lithium: virtual machine storage for the cloud’ (ACM, 2010, edn.), pp. 15-26

61 Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., and Hauser, C.H.: ‘Managing update conflicts in Bayou, a weakly connected replicated storage system’ (ACM, 1995. 1995)

62 Choi, S., Choi, M., Lee, C., and Youn, H.Y.: ‘Distributed lock manager for distributed file system in shared-disk environment’, in Editor (Ed.)^(Eds.): ‘Book Distributed lock manager for distributed file system in shared-disk environment’ (IEEE, 2010, edn.), pp. 2706-2713

63 Jenkins, G.O.: ‘Distributed lock management in a cloud computing environment’, in Editor (Ed.)^(Eds.): ‘Book Distributed lock management in a cloud computing environment’ (Google Patents, 2017, edn.), pp.

64 Wu, S., and Wu, K.-L.: ‘An Indexing Framework for Efficient Retrieval on the Cloud’, IEEE Data Eng. Bull., 2009, 32, (1), pp. 75-82

65 Chandra, T.D., Griesemer, R., and Redstone, J.: ‘Paxos made live: an engineering perspective’, in Editor (Ed.)^(Eds.): ‘Book Paxos made live: an engineering perspective’ (ACM, 2007, edn.), pp. 398-407

66 SELLSTRÖM, G.A., and TÕNISSON, M.R.: ‘Analysis of Voting Algorithms: a comparative study of the Single Transferable Vote’

67 Hardekopf, B., Kwiat, K., and Upadhyaya, S.: ‘A decentralized voting algorithm for increasing dependability in distributed systems’, in Editor (Ed.)^(Eds.): ‘Book A decentralized voting algorithm for increasing dependability in distributed systems’ (2001, edn.), pp.

68 Castro, M., and Liskov, B.: ‘Practical Byzantine fault tolerance’, in Editor (Ed.)^(Eds.): ‘Book Practical Byzantine fault tolerance’ (1999, edn.), pp. 173-186

69 Kanrar, S., Chattopadhyay, S., and Chaki, N.: ‘A new voting-based mutual exclusion algorithm for distributed systems’, in Editor (Ed.)^(Eds.): ‘Book A new voting-based mutual exclusion algorithm for distributed systems’ (IEEE, 2013, edn.), pp. 1-5

70 Koçi, A., and Çiço, B.: 'Distributed Lock Management in Cloud Computing: Performance and Challenges' International Scientific Conference Computer Science`2018, 2018, 1, pp. 67 - 74

71 Thomas, R.H.: 'A majority consensus approach to concurrency control for multiple copy databases', ACM Transactions on Database Systems (TODS), 1979, 4, (2), pp. 180-209

72 Barbara, D., Garcia-Molina, H., and Spauster, A.: 'Increasing availability under mutual exclusion constraints with dynamic vote reassignment', ACM Transactions on Computer Systems (TOCS), 1989, 7, (4), pp. 394-426

73 Godfrey, P., Shenker, S., and Stoica, I.: 'Minimizing churn in distributed systems' (ACM, 2006. 2006)

74 Chidambaram, V., Pillai, T.S., Arpaci-Dusseau, A.C., and Arpaci-Dusseau, R.H.: 'Optimistic crash consistency', in Editor (Ed.)^(Eds.): 'Book Optimistic crash consistency' (ACM, 2013, edn.), pp. 228-243

75 Ignat, C., and Norrie, M.C.: 'Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems', in Editor (Ed.)^(Eds.): 'Book Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems' (2002, edn.), pp.

76 Peluso, S., Romano, P., and Quaglia, F.: 'Score: A scalable one-copy serializable partial replication protocol', in Editor (Ed.)^(Eds.): 'Book Score: A scalable one-copy serializable partial replication protocol' (Springer-Verlag New York, Inc., 2012, edn.), pp. 456-475

77 Patil, S., Gibson, G.A., Ganger, G.R., Lopez, J., Polte, M., Tantisiriroj, W., and Xiao, L.: 'In search of an API for scalable file systems: Under the table or above it?', in Editor (Ed.)^(Eds.): 'Book In search of an API for scalable file systems: Under the table or above it?' (2009, edn.), pp.

78 Vaquero, L.M., Rodero-Merino, L., Caceres, J., and Lindner, M.: 'A break in the clouds: towards a cloud definition', ACM SIGCOMM Computer Communication Review, 2008, 39, (1), pp. 50-55

79 Mell, P., and Grance, T.: ‘The NIST definition of cloud computing’, 2011

80 Foster, I., Zhao, Y., Raicu, I., and Lu, S.: ‘Cloud computing and grid computing 360-degree compared’, in Editor (Ed.)^(Eds.): ‘Book Cloud computing and grid computing 360-degree compared’ (Ieee, 2008, edn.), pp. 1-10

81 Sallé, M.: ‘IT Service Management and IT Governance: review, comparative analysis and their impact on utility computing’, Hewlett-Packard Company, 2004, pp. 8-17

82 Malhotra, L., Agarwal, D., and Jaiswal, A.: ‘Virtualization in cloud computing’, J. Inform. Tech. Softw. Eng, 2014, 4, (2)

83 Kephart, J.O., and Chess, D.M.: ‘The vision of autonomic computing’, Computer, 2003, (1), pp. 41-50

84 Zahariev, A.: ‘Google app engine’, Helsinki University of Technology, 2009, pp. 1-5

85 Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A.: ‘Xen and the art of virtualization’, in Editor (Ed.)^(Eds.): ‘Book Xen and the art of virtualization’ (ACM, 2003, edn.), pp. 164-177

86 Dantas, R., Sadok, D., Flinta, C., and Johnsson, A.: ‘Kvm virtualization impact on active round-trip time measurements’, in Editor (Ed.)^(Eds.): ‘Book Kvm virtualization impact on active round-trip time measurements’ (IEEE, 2015, edn.), pp. 810-813

87 Infrastructure, V.: ‘Resource management with VMware DRS’, VMware Whitepaper, 2006, 13

88 Al-Fares, M., Loukissas, A., and Vahdat, A.: ‘A scalable, commodity data center network architecture’, in Editor (Ed.)^(Eds.): ‘Book A scalable, commodity data center network architecture’ (ACM, 2008, edn.), pp. 63-74

89 Hosting, C.: ‘CLoud Computing and Hybrid Infrastructure from GoGrid’, in Editor (Ed.)^(Eds.): ‘Book CLoud Computing and Hybrid Infrastructure from GoGrid’ (2012, edn.), pp.

90 Prodan, R., and Ostermann, S.: ‘A survey and taxonomy of infrastructure as a service and web hosting cloud providers’, in Editor (Ed.)^(Eds.): ‘Book A survey and

taxonomy of infrastructure as a service and web hosting cloud providers' (IEEE, 2009, edn.), pp. 17-25

91 Qian, L., Luo, Z., Du, Y., and Guo, L.: 'Cloud computing: An overview', in Editor (Ed.)^(Eds.): 'Book Cloud computing: An overview' (Springer, 2009, edn.), pp. 626-631

92 Zhang, Q., Cheng, L., and Boutaba, R.: 'Cloud computing: state-of-the-art and research challenges', Journal of internet services and applications, 2010, 1, (1), pp. 7-18

93 Gonzalez, J.U., and Krishnan, S.: 'Building Your Next Big Thing with Google Cloud Platform: A Guide for Developers and Enterprise Architects' (Apress, 2015. 2015)

94 Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., and Simitci, H.: 'Windows Azure Storage: a highly available cloud storage service with strong consistency', in Editor (Ed.)^(Eds.): 'Book Windows Azure Storage: a highly available cloud storage service with strong consistency' (ACM, 2011, edn.), pp. 143-157

95 Huo, J., Qu, H., and Wu, L.: 'Design and implementation of private cloud storage platform based on OpenStack', in Editor (Ed.)^(Eds.): 'Book Design and implementation of private cloud storage platform based on OpenStack' (IEEE, 2015, edn.), pp. 1098-1101

96 Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A.: 'kvm: the Linux virtual machine monitor', in Editor (Ed.)^(Eds.): 'Book kvm: the Linux virtual machine monitor' (Dttawa, Dntorio, Canada, 2007, edn.), pp. 225-230

97 Gilbert, S., and Lynch, N.: 'Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services', Acm Sigact News, 2002, 33, (2), pp. 51-59

98 Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R.E.: 'Bigtable: A distributed storage system for structured data', ACM Transactions on Computer Systems (TOCS), 2008, 26, (2), pp. 4

99 Khetrapal, A., and Ganesh, V.: 'HBase and Hypertable for large scale distributed storage systems', Dept. of Computer Science, Purdue University, 2006, pp. 22-28

100 Chodorow, K., and Dirolf, M.: 'MongoDB: The Definitive Guide O'Reilly Media', September 2010, 2010

101 Koçi, A., and Çiço, B.: ‘DLMCC-Distributed Lock Management in Cloud Computing’, ‘Book DLMCC-Distributed Lock Management in Cloud Computing’ (2015, edn.), pp. 52-58

102 Koçi, A., and Çiço, B.: ‘ADLMCC–Asymmetric distributed lock management in cloud computing’, International Journal On Information Technologies And Security, 2018, 10, (3), pp. 37-52

103 Koci, A.C., Betim: ‘Resource Starvation in Asymmetric Distributed Lock Management in Cloud Computing’, Albanian Journal of Natural and Technical Sciences, 2018, 47, (2), pp. 91-102

104 Koçi, A., and Çiço, B.: ‘Performance Evaluation of the Asymmetric Distributed Lock Management in Cloud Computing’, Performance Evaluation, 2018, 180, (49)

# APPENDIX

## REQUESTS MODULE CODE OF THE ASYMMETRIC DISTRIBUTED LOCK MANAGEMENT ALGORITHM

```
package main.entities;
import java.util.logging.Level;
import java.util.logging.Logger;
import main.communication.RequestLock;
public class Client {
        public final static Logger LOGGER = Logger.getLogger(Client.class.getName());
        private String id;
        public Client(String id){
                this.id=id;
        }
        //Random Server
        //STEP 1 - (1)
        public RequestLock issueRequest(RequestLock.LOCK_MODE lockMode,String
filePath) {
                //Create new FileNode object with the requested filePath
                FileNode requestedFile = new FileNode(filePath);
                Server initialServer = Cluster.getRandomServer();
                //The requester = client
          return new RequestLock(this.id, initialServer, lockMode, requestedFile,
System.currentTimeMillis());
        }
        //Specified Server
        //STEP 1 - (1)
```

```
	public void issueRequest(RequestLock.LOCK_MODE lockMode,String filePath, Server initialServer) {
		//Create new FileNode object with the requested filePath
		FileNode requestedFile = new FileNode(filePath);
		//The requester = client
	    RequestLock request = new RequestLock(this.id, initialServer, lockMode, requestedFile, System.currentTimeMillis());
	}
	//Specified Server
	//STEP 1 - (1)
	public RequestLock issueRequest(String lockMode,FileNode requestedFile, Server initialServer) {
		RequestLock request;
		if (lockMode.equals("Read")) {
			return new RequestLock(this.id, initialServer, RequestLock.LOCK_MODE.Read, requestedFile, System.currentTimeMillis());
		} else if  (lockMode.equals("Write")) {
			return new RequestLock(this.id, initialServer, RequestLock.LOCK_MODE.Write, requestedFile, System.currentTimeMillis());
		} else {
			RequestLock.LOCK_MODE randMode = RequestLock.getRandomLockMode();
			return new RequestLock(this.id, initialServer, randMode, requestedFile, System.currentTimeMillis());
		}
	}
	public String getId() {
		return id;
	}
	public void setId(String id) {
		this.id = id;
	}
	@Override
	public String toString() {
		return "Client[" + this.id + "]";
	}
}
```

# CLUSTER INFRASTRUCTURE CODE OF THE ASYMMETRIC DISTRIBUTED LOCK MANAGEMENT ALGORITHM

```
package main.entities;
import java.io.File;
import java.util.ArrayList;
import java.util.Random;
import java.util.concurrent.ThreadLocalRandom;
import java.util.logging.Level;
import java.util.logging.Logger;
public class Cluster {
	public static final Logger LOGGER = Logger.getLogger(Cluster.class.getName());
	//add info about the created servers in the static SERVER_NODE_TABLE (serverId-ServerStatus-SwitchOverStatus)=ServerNodeInfo
	public static ArrayList<File> STORAGE_NODES = new ArrayList<File>();
	public static ArrayList<FileNode> FILE_NODES = new ArrayList<FileNode>();
	public static ArrayList<Client> CLIENTS = new ArrayList<Client>();
	public static ArrayList<Server> SERVERS = new ArrayList<Server>();
	public static String DIRECTORY;
	public Cluster(){
	}
	public static void clear() {
		Cluster.SERVERS.clear();
		Cluster.FILE_NODES.clear();
		Cluster.STORAGE_NODES.clear();
		Cluster.CLIENTS.clear();
	}
	public static int checkLockBit(String filePath){
		for (FileNode file : FILE_NODES){
			if (file.getAbsolutePath().compareTo(filePath)==0)
				return file.getLockBit();
		}
		//file not found in the array - add EXCEPTION
		return -1;
	}
	public static FileNode getFileNodeFromPath(String filePath){
		for (FileNode file : FILE_NODES){
			if (file.getAbsolutePath().compareTo(filePath)==0)
				return file;
		}
		//file not found in the array - add EXCEPTION
		return null;
	}
	public static Server getRandomServer() {
		int	randomNum	=	ThreadLocalRandom.current().nextInt(0,
SERVERS.size());
		Server server = SERVERS.get(randomNum);;
		while (!server.isAlive()) {
			randomNum	=	ThreadLocalRandom.current().nextInt(0,
SERVERS.size());
			server = SERVERS.get(randomNum);
		}
		return server;
	}
	public static Client getRandomClient() {
		int randomNum = ThreadLocalRandom.current().nextInt(0,
CLIENTS.size());
		Client client = CLIENTS.get(randomNum);;

		return client;
	}
	public static FileNode getRandomFileNode() {
		int randomNum = ThreadLocalRandom.current().nextInt(0,
FILE_NODES.size());
		FileNode fileNode = FILE_NODES.get(randomNum);

		return fileNode;
	}
	public static FileNode getFileNode(String filePath) {
		for (FileNode file : FILE_NODES) {
			if (file.getAbsolutePath().equals(filePath)) {
				return file;
			}
		}
		//Enter a new fileNode with this filePath in FILE_NODES
		FileNode newFile = new FileNode(filePath);
		FILE_NODES.add(newFile);
		return newFile;
	}
	public static boolean existsDirectory(String path){
		if (path == null || path.equals("") || path.isEmpty()) return false;
		File f = new File(path);
		if (f.exists() && f.isDirectory()) {
			return true;
```

```
                }
                return false;
        }
        public static boolean existsFile(String path){
                if (path.equals("") || path.isEmpty() || path == null ) return false;
                File f = new File(path);
                if (f.exists() && f.isFile()) {
                        return true;
                }
                return false;
        }
        public static void cleanDirectory(File dir) {
           if (dir.isDirectory()) {
              File[] files = dir.listFiles();
              if (files != null && files.length > 0) {
                 for (File aFile : files) {
                    removeDirectory(aFile);
                 }
              } else {
                 LOGGER.log(Level.INFO,   "Folder   '"+   dir.getAbsolutePath()   +"'   is
already empty!");
              }
           }
        }
        public static void removeDirectory(File dir) {
           if (dir.isDirectory()) {
              File[] files = dir.listFiles();
              if (files != null && files.length > 0) {
                 for (File aFile : files) {
                    removeDirectory(aFile);
                  }
              }

              if (dir.delete()){
                  LOGGER.log(Level.INFO, "Deleting folder: deleting containing folder
'"+ dir.getAbsolutePath() +"'");
              }
              else {
                  LOGGER.log(Level.SEVERE, "Deleting folder: ERROR containing
folder '"+ dir.getAbsolutePath() +"' cannot be deleted!");
              }
           } else {
              if (dir.delete()) {
                  LOGGER.log(Level.INFO, "Deleting folder: deleting containing file '"+
dir.getAbsolutePath() +"'");
              }
              else {
                  LOGGER.log(Level.SEVERE, "Deleting folder: ERROR containing file
'"+ dir.getAbsolutePath() +"' cannot be deleted!");
              }
           }
        }
        public static void deleteEntireFolder(File folder) {
                  if (Cluster.existsDirectory(folder.getAbsolutePath())) {
                        LOGGER.log(Level.INFO,     "Deleting     folder:     "     +
folder.getAbsolutePath());

                        File[] files = folder.listFiles();
                        if(files!=null) { //some JVMs return null for empty dirs
                            for(File f: files) {
                                if(f.isDirectory()) {
                                    LOGGER.log(Level.INFO, "Deleting folder: deleting
containing folder '"+ f.getAbsolutePath() +"'");
                                    deleteEntireFolder(f);
                                } else {
                                    System.out.println("Deleting folder: deleting containing
file '"+ f.getAbsolutePath() +"'");
                                    f.delete();
                                }
                            }
                        }
                        else {
                                LOGGER.log(Level.INFO, "Deleting folder: folder is already
empty!");
                        }
                        deleteEntireFolder(folder);
                    }
                    else {
                            LOGGER.log(Level.SEVERE, "Cluster Directory
'"+Cluster.DIRECTORY+"' is not a valid directory, cannot be cleared!");
                    }
        }
        public static void cleanFolder(String path) {
                    if (Cluster.existsDirectory(path)) {
                            File folder = new File(path);
                            cleanDirectory(folder);
                    }
```

```
                else {
                        LOGGER.log(Level.SEVERE, "Cluster Directory
'"+Cluster.DIRECTORY+"' is not a valid directory, cannot be cleared!");
                }
        }
        public static String getClusterInformation() {
                return "Cluster has: " + STORAGE_NODES.size() + " Storage Nodes, "
+FILE_NODES.size() + " Files, " +
                                CLIENTS.size() + " Clients, " + SERVERS.size() + "
Servers.";
        }

}
```

# SERVERS MODULE CODE OF THE ASYMMETRIC DISTRIBUTED LOCK MANAGEMENT ALGORITHM

```
package main.entities;

import java.util.logging.Level;
import java.util.logging.Logger;
package main.communication;
import main.communication.RequestLock;

public class Server {
        public static final Logger LOGGER = Logger.getLogger(Server.class.getName());
        private String id;
        private LockManager lockManager;
  public static enum SERVER_STATUS {
    Running,
    Suspended,
    Idle
  }

  public Server (String serverId){
                this.id = serverId;

                this.lockManager = new LockManager();
        }

  //STEP 1 - (2)
  public void startRequest(RequestLock request){

        LOGGER.log(Level.INFO, "[04] "+request.getRequestId() + ": Checking if exists in RLT of
initial " + this);
created the  file
        int indexRequestInRLT =
this.lockManager.requestExistsInRequestingLockTable(request);
        if (indexRequestInRLT != -1) {

                LOGGER.log(Level.INFO, "[05] "+request.getRequestId() + ": This request EXISTS
in RLT of initial " + this);
```

```
                        LOGGER.log(Level.INFO, "[06] "+request.getRequestId() + ": ADDING this
request in RLT of initial " + this);
                        LOGGER.log(Level.INFO, "[07] "+request.getRequestId() + ": ADDING this
request in LFL of initial " + this);

                        //this means the server has requested this file earlier, so it is in its RLT
                        //add this request in RLT, and then in LFL - after adding in LFL the request start
asking for permissions
                        this.lockManager.getRequestingLockTable().add(request);

            LockManager.LOCKED_FILE_LIST.get(request.getRequestedPath()).addRequestLock(req
uest);
            }
            else {



                        LOGGER.log(Level.INFO, "[08] "+request.getRequestId() + ": This request
DOESN'T EXISTS in RLT of initial " + this);
                        LOGGER.log(Level.INFO, "[09] "+request.getRequestId() + ": Searching the
requested path '"+request.getRequestedPath()+"' in FD");
                        //The first procedure is to look up FD and find the node of the lock manager
responsible for the file.
                        //If the Server exists alive in the Cloud, the lock manager on that server is
responsible for the file.
                        Server ownerServer =
LockManager.findFileOwner(request.getRequestedPath());
                        LOGGER.log(Level.INFO, "[10] "+request.getRequestId() + ": Found owner
Server in FD, "+ownerServer);
                        //The next one is to check SNT to find out if the server is alive and still in the
cloud
            if (LockManager.getServerStatus(ownerServer) != Server.SERVER_STATUS.Running) {
                        LOGGER.log(Level.INFO, "[11] "+request.getRequestId() + ": Owner
"+ownerServer+" is down");
                        ownerServer = LockManager.getSwitchOverServer(ownerServer);
                        LOGGER.log(Level.INFO, "[12] "+request.getRequestId() + ": Getting
SwitchOver: "+ownerServer + " in SNT");
            }
            //STEP 1 - (3)
            if (this.equals(ownerServer)){
                    LOGGER.log(Level.INFO, "[13] "+request.getRequestId() + ": Found owner
"+ownerServer+" is the same as initial "+this);
                    LOGGER.log(Level.INFO, "[14] "+request.getRequestId() + ": ADDING this
request in RLT of initial/owner " + ownerServer);
                    LOGGER.log(Level.INFO, "[07] "+request.getRequestId() + ": ADDING this
request in LFL");
                    //add this request in RLT, and then in LFL - after adding in LFL the request start
asking for permissions
                    this.lockManager.getRequestingLockTable().add(request);
          LockManager.getLockedFileInfoFromPath(request.getRequestedPath()).addRequestLoc
k(request);
          }
          else {
                    LOGGER.log(Level.INFO, "[15] "+request.getRequestId() + ": Found owner
"+ownerServer+" is NOT the same as initial "+this);
                    request.setOwnerServer(ownerServer);

                    //put the request on M-out T of the owner Server and MIGRATE
                    if (!
ownerServer.lockManager.getMigrateOutTable().containsKey(request.getRequestedPath())) {
          ownerServer.lockManager.getMigrateOutTable().put(request.getRequestedPath(),
request.getInitialServer());
                              LOGGER.log(Level.INFO, "[17] "+request.getRequestId() + ": Request is
not in MoT of owner "+ownerServer+", ADDING it");
                    //add this request in RLT, and then in LFL of the ownerServer
                    ownerServer.lockManager.getRequestingLockTable().add(request);

                    LOGGER.log(Level.INFO, "[17] "+request.getRequestId() + ": ADDING this
request in RLT of owner " + ownerServer);
                    LOGGER.log(Level.INFO, "[07] "+request.getRequestId() + ": ADDING this
request in LFL");
          LockManager.getLockedFileInfoFromPath(request.getRequestedPath()).addRequestLoc
k(request);
                    } else if (this ==
ownerServer.lockManager.getMigrateOutTable().get(request.getRequestedPath())) {
                              LOGGER.log(Level.INFO, "[40] "+request.getRequestId() + ": Owner " +
ownerServer + " has granted rights to this server, Initial " + this
                                        + ". Continuing execution with Final Executor " + this);
                              request.setExecutorServer(this);
                              LOGGER.log(Level.INFO, "[07] "+request.getRequestId() + ": ADDING
this request in LFL");


          LockManager.getLockedFileInfoFromPath(request.getRequestedPath()).addRequestLoc
k(request);
                    }
                    else {
                              Server serverMigratedTo =
ownerServer.lockManager.getMigrateOutTable().get(request.getRequestedPath());
                              LOGGER.log(Level.INFO, "[17] "+request.getRequestId() + ": Request
EXISTS in the MoT of owner "+ownerServer
                                                  + " with Migrated "+ serverMigratedTo +". Passing this
request to MIGRATED " + serverMigratedTo);

                              //MIGRATE
                              LOGGER.log(Level.INFO, "[40] "+request.getRequestId() + ": This
request is now handled by the Final Executor, MIGRATED server " + serverMigratedTo);
                              request.setExecutorServer(serverMigratedTo);
                              serverMigratedTo.startRequestAsMigrated(request);
                    }

          }
          }
   }

   //In case of the final executor, just execute it in LFL
   public void startRequestAsMigrated(RequestLock request){

                    LOGGER.log(Level.INFO, "[06] "+request.getRequestId() + ": ADDING this
request in RLT of migrated " + this);
          this.lockManager.getRequestingLockTable().add(request);

                    //After adding in LFL the request start asking for permissions
                    LOGGER.log(Level.INFO, "[07] "+request.getRequestId() + ": ADDING this
request in LFL of Final Executor " + this);
                    this.lockManager.getRequestingLockTable().add(request);
```

```
            LockManager.getLockedFileInfoFromPath(request.getRequestedPath()).addRequestLoc
k(request);
    }
    public void removeRequestFromRLT(RequestLock request) {
            if (this.getLockManager().getRequestingLockTable().contains(request)){
                    this.getLockManager().getRequestingLockTable().remove(request);
            }
    }
    public void removeServerFromMiT(String requestedPath) {
            if (this.getLockManager().getMigrateInTable().contains(requestedPath)){
                    this.getLockManager().getMigrateInTable().remove(requestedPath);
            }
    }

    public void removeServerFromMoT(String requestedPath) {
            if (this.getLockManager().getMigrateOutTable().contains(requestedPath)){
                    this.getLockManager().getMigrateOutTable().remove(requestedPath);
            }
    }
    public boolean isAlive() {
            if (LockManager.getServerStatus(this) == Server.SERVER_STATUS.Running)
                    return true;
            return false;
    }
            @Override
            public boolean equals(Object obj) {
                    if (this == obj)
                            return true;
                    if (obj == null)
                            return false;
                    if (getClass() != obj.getClass())
                            return false;
                    Server other = (Server) obj;
                    if (id == null) {
                            if (other.id != null)
                                    return false;
                    } else if (!id.equals(other.id))
                            return false;
                    return true;
```

```
	}
	public void getMigratedServerForFile(String filePath) {

	}
	public String getId() {
		return id;
	}
	public void setId(String id) {
		this.id = id;
	}
	public LockManager getLockManager() {
		return lockManager;
	}
	public void setLockManager(LockManager lockManager) {
		this.lockManager = lockManager;
	}
	@Override
	public String toString() {
		return "Server[" + id + "]";
	}
}
```

# SERVERS INFORMATION MAINTENANCE CODE OF THE ASYMMETRIC DISTRIBUTED LOCK MANAGEMENT ALGORITHM

```
package main.entities;
public class ServerNodeInfo {
        private String serverId;
        private Server.SERVER_STATUS status;
        private Server switchOverServer;
        public ServerNodeInfo (String serverId, Server.SERVER_STATUS status, Server
switchOverServer){
                this.serverId = serverId;
                this.status = status;
                this.switchOverServer = switchOverServer;
        }
        public String getServerId() {
        return serverId;
        }
        public void setServerId(String serverId) {
                this.serverId = serverId;
        }
        public Server.SERVER_STATUS getStatus() {
                return status;
        }
        public void setStatus(Server.SERVER_STATUS status) {
                this.status = status;
        }
        public Server getSwitchOverServer() {
                return switchOverServer;
        }
        public void setSwitchOverServer(Server switchOverServer) {
                this.switchOverServer = switchOverServer;
        }
}
```

# LOCK MANAGER CODE OF THE ASYMMETRIC DISTRIBUTED LOCK MANAGEMENT ALGORITHM

```
package main.entities;

import java.util.ArrayList;
import java.util.Hashtable;
import java.util.logging.Logger;
import main.communication.RedirectRequest;
import main.communication.RequestLock;
public class LockManager {
        public static final Logger LOGGER = Logger.getLogger(LockManager.class.getName());
        //private Hashtable<String, Node> ServerNodeTable = new Hashtable<>();
        public static ArrayList<ServerNodeInfo> SERVER_NODE_TABLE = new
ArrayList<ServerNodeInfo>();
        public static Hashtable<String, Server> FILE_DIRECTORY = new Hashtable<String,
Server>();
        //to use: RequestLock or RedirectRequest ?
        private Hashtable<String, Server> MigrateInTable = new Hashtable<String, Server>();
        private Hashtable<String, Server> MigrateOutTable = new Hashtable<String, Server>();
        //RLT keeps track of all locks a server has requested
  private ArrayList<RequestLock> RequestingLockTable = new ArrayList<RequestLock>();
  //LFL includes all necessary information about locks for which a lock manager is managing
and is responsible for
  public static Hashtable<String, LockedFileInfo> LOCKED_FILE_LIST = new Hashtable<String,
LockedFileInfo>();
  public LockManager() {
        this.RequestingLockTable = new ArrayList<RequestLock>();
  }
  /**
   * @param request - the request to be checked if exists in the RLT
   * @return return -1 if not found, otherwise the index of the found request in the ArrayList
RLT
   */
  public int requestExistsInRequestingLockTable(RequestLock request){
        if (this.RequestingLockTable == null || this.RequestingLockTable.isEmpty()) {
            return -1;
        }
        int i=this.RequestingLockTable.size()-1;
        while (i>=0 && this.RequestingLockTable.get(i).compareTo(request)!=0){
            i--;
        }

        return i;
  }

  public static Server findFileOwner(String filePath) {
        Server foundServer = FILE_DIRECTORY.get(filePath);
        return foundServer;
  }
  public static Server.SERVER_STATUS getServerStatus(Server server){
            int i = 0;
            while(i<LockManager.SERVER_NODE_TABLE.size() &&
!LockManager.SERVER_NODE_TABLE.get(i).getServerId().equals(server.getId())){
                    i++;
            }
            return LockManager.SERVER_NODE_TABLE.get(i).getStatus();
        }
        public static Server getSwitchOverServer(Server server){
            int i = 0;
            while(i<LockManager.SERVER_NODE_TABLE.size() &&
!LockManager.SERVER_NODE_TABLE.get(i).getServerId().equals(server.getId())){
                    i++;
            }
            return LockManager.SERVER_NODE_TABLE.get(i).getSwitchOverServer();
        }
        //will add a new LockedFileInfo if the path doesn't exists
        public static LockedFileInfo getLockedFileInfoFromPath(String path) {
            if (LOCKED_FILE_LIST == null || LOCKED_FILE_LIST.isEmpty()) {
                    LOCKED_FILE_LIST.put(path, new LockedFileInfo());
            } else {
                    LockedFileInfo lockedFileInfo = LOCKED_FILE_LIST.get(path);
                    if (lockedFileInfo == null) {
                        LOCKED_FILE_LIST.put(path, new LockedFileInfo());
                    }
```

```
		}
		return LOCKED_FILE_LIST.get(path);
	}
	public Hashtable<String, Server> getMigrateInTable() {
		return MigrateInTable;
	}
	public void setMigrateInTable(Hashtable<String, Server> migrateInTable) {
		MigrateInTable = migrateInTable;
	}
	public Hashtable<String, Server> getMigrateOutTable() {
		return MigrateOutTable;
	}
	public void setMigrateOutTable(Hashtable<String, Server> migrateOutTable) {
		MigrateOutTable = migrateOutTable;
	}
	public ArrayList<RequestLock> getRequestingLockTable() {
		return RequestingLockTable;
	}
	public void setRequestingLockTable(ArrayList<RequestLock> requestingLockTable) {
		RequestingLockTable = requestingLockTable;
	}
}
```

# FILE ATTRIBUTE CODE OF THE ASYMMETRIC DISTRIBUTED LOCK MANAGEMENT ALGORITHM

```
package main.entities;

import java.io.File;

public class FileNode extends File {
	/**
	 *
	 */
	private static final long serialVersionUID = 1L;
	private short lockBit;

	public FileNode (String pathname){
		super(pathname);
		this.lockBit = (short) 0;
	}
	public FileNode (String pathname, short lockBit){
		super(pathname);
		this.lockBit = lockBit;
	}
	public short getLockBit() {
		return lockBit;
	}
	public void setLockBit(short lockBit) {
		this.lockBit = lockBit;
	}
	public int compareToFilePath(String filePath){
		return this.getName().compareTo(filePath);
	}
	@Override
	public String toString() {
		return this.getName();
	}
	}
```

# STORAGE NODE CODE OF THE ASYMMETRIC DISTRIBUTED LOCK MANAGEMENT ALGORITHM

```
package main.entities;

import java.util.ArrayList;

/**
 * This is a shared folder
 */
public class Node {
	private String nodePath;

	private ArrayList<FileNode> files;

	public Node(String nodePath){
		this.nodePath = nodePath;
	}

	public String getNodePath() {
		return nodePath;
	}

	public void setNodePath(String nodePath) {
		this.nodePath = nodePath;
	}

	public ArrayList<FileNode> getFiles() {
		return files;
	}

	public void setFiles(ArrayList<FileNode> files) {
		this.files = files;
	}
}
```

# FILE LOCK MANAGEMENT CODE OF THE ASYMMETRIC DISTRIBUTED LOCK MANAGEMENT ALGORITHM

```
package main.entities;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;
import java.util.logging.Level;
import java.util.logging.Logger;
import gui.ClusterScene;
import main.communication.RequestLock;
import settings.Factory;
import settings.Info;
public class LockedFileInfo {
        //for each File there will be one LockedFileInfo -> which contains 2 lists and the 2 above
fields
        public static final Logger LOGGER = Logger.getLogger(LockedFileInfo.class.getName());
        public static int STARVATION_NUMBER = 10000;
        //V1.9 Starvation 100
        public static final int STARVATION_NUMBER_DEFAULT = 100;
        public LockedFileInfo(){
                GrantedLocks = new LinkedList<RequestLock>();
                BlockedLocks = new LinkedList<RequestLock>();
        }
  private Queue<RequestLock> GrantedLocks;
  private Queue<RequestLock> BlockedLocks;
  public void addRequestLock(RequestLock request){
        LOGGER.log(Level.INFO, "[21] "+request.getRequestId() + ": Checking BitLock if
permission for requested file can be granted...");
        int lockBit = Cluster.checkLockBit(request.getRequestedPath());
```

```
            if (lockBit == 0) {
            LOGGER.log(Level.INFO, "[22] "+request.getRequestId() + ": Requested file has BitLock
= 0");
            } else if (lockBit == 1) {
            LOGGER.log(Level.INFO, "[31] "+request.getRequestId() + ": Requested file has BitLock
= 1");
            } else {
            LOGGER.log(Level.INFO, "[32] "+request.getRequestId() + ": Requested file has BitLock
= 2");
            }
            RequestLock.LOCK_MODE lockMode = request.getLockMode();
            if (lockMode == RequestLock.LOCK_MODE.Read) {
            LOGGER.log(Level.INFO, "[23] "+request.getRequestId() + ": Requested LockMode is
READ");
            } else {
            LOGGER.log(Level.INFO, "[33] "+request.getRequestId() + ": Requested LockMode is
WRITE");
            }
            //HERE CHECK IF PERMISSION CAN BE GRANTED - rr/ww/repREP/rw/repR/repW/
            //only when there is a WRITE, cannot be granted
            if ((Cluster.checkLockBit(request.getRequestedPath())>0 &&
request.getLockMode()==RequestLock.LOCK_MODE.Write)
                        ||
                        (Cluster.checkLockBit(request.getRequestedPath())>1 &&
request.getLockMode()==RequestLock.LOCK_MODE.Read)
                  ) {
                  addBlockedLock(request);
            }
            else {
                  addGrantedLock(request);
            }
    }
            public void addBlockedLock(RequestLock request){
                  //Check first for starvation
                  //V1.9 WITH STARVATION GENERIC
                  if (ClusterScene.starvationIncludedCBox.isSelected()) {
                        checkForStarvation();
                  }
                  this.BlockedLocks.add(request);
```

```
                LOGGER.log(Level.INFO, "[24] "+request.getRequestId() + ": Added in Blocked
List");
        }

        //adds the request in the GrantedLocks and removes it from BlockedLocks
        public void addGrantedLock(RequestLock request){
                request.setGrantedTimestamp(System.currentTimeMillis());
                switch (request.getLockMode()){
                case Read:
                        //change lockBit to 1 (lockBit remains 1 of there was another server
reading it already)
        Cluster.getFileNodeFromPath(request.getRequestedPath()).setLockBit((short) 1);
                        break;
                case Write:
        Cluster.getFileNodeFromPath(request.getRequestedPath()).setLockBit((short) 2);
                        break;
                }
                if (this.BlockedLocks.contains(request)) {
                        this.BlockedLocks.remove(request);
                }
                this.GrantedLocks.add(request);
                LOGGER.log(Level.INFO, "[25] "+request.getRequestId() + ": Added in Granted
Queue");
                //get RequestedFile from Cluster to be used for THREAD lock object
                FileNode lockObject = Cluster.getFileNode(request.getRequestedPath());
                synchronized (lockObject) {
                        try(FileWriter fw = new FileWriter(request.getRequestedPath(), true);
                            BufferedWriter bw = new BufferedWriter(fw);
                            PrintWriter out = new PrintWriter(bw))
                        {
                                if (request.getLockMode() == RequestLock.LOCK_MODE.Read)
{
                                out.println("Request " + request.getRequestId() + " by client " +
request.getRequesterId() + " READ this file at " + System.currentTimeMillis()+"...");
                                LOGGER.log(Level.INFO, "[26] "+request.getRequestId() + ":
File READ successfully!");
                                }
                                else {
                            out.println("Request " + request.getRequestId() + " by client " +
request.getRequesterId() + " WROTE this file at " + System.currentTimeMillis()+"...");
                            LOGGER.log(Level.INFO, "[27] "+request.getRequestId() + ": File
WRITTEN successfully!");
                                }
                        } catch (IOException e) {
                                LOGGER.log(Level.SEVERE, "IOException thrown for request " +
request.getRequestId() + " while WRITING requested file: " + e.getMessage());
                        } catch (NullPointerException e) {
                                LOGGER.log(Level.SEVERE, "NullPointerException thrown for
request " + request.getRequestId() + " while WRITING requested file: " + e.getMessage());
                        }
                        //The reading, writing was done, notify other threads in BlockedList
                        this.GrantedLocks.remove(request);
                        LOGGER.log(Level.INFO, "[34] "+request.getRequestId() + ": Request
removed from Granted List");
                        //The initiator server AND ownerServer removes the request from its
RLT/MoT/MiT table
        //initiatiorServer.getLockManager().getRequestingLockTable().remove(request);
                        request.clearRequestFromSystem();
                        //set LockBit of the requester file to 0
        Cluster.getFileNodeFromPath(request.getRequestedPath()).setLockBit((short) 0);
                        if (this.GrantedLocks.isEmpty()){
                                //means we can now insert
                                LOGGER.log(Level.INFO, "[35] "+request.getRequestId()+":
Granted Lock List for File " + request.getRequestedFile().getName()
                                                                        +" is empty, releasing Blocked
Requests");
                                if (! this.BlockedLocks.isEmpty()) {
                                        if (this.BlockedLocks.peek() != null) {
                                                RequestLock.LOCK_MODE mode =
this.BlockedLocks.peek().getLockMode();
                                                //if the first req waiting is a READ, get all
READs until a WRITE appears, ELSE get only the first WRITE
                                                if (mode == RequestLock.LOCK_MODE.Read){
                                                        LOGGER.log(Level.INFO, "[37]
"+request.getRequestId()+": Releasing Blocked Requests - the first Blocked request is READ
mode."
```

```
                                                                                    +" Getting all Blocked
READ requests up to a WRITE request");
                                                                    while (! this.BlockedLocks.isEmpty()
&& this.BlockedLocks.peek() !=null &&
this.BlockedLocks.peek().getLockMode()==RequestLock.LOCK_MODE.Read){
                                                                            RequestLock req =
this.BlockedLocks.poll();
                                                                            LOGGER.log(Level.INFO, "[38]
"+request.getRequestId()+": Releasing Blocked Request - releasing " + req.getRequestId()
                                                                                    + ", " +
req.getLockMode() + " mode");
                                                                            if (req != null) {
                                                                                    addGrantedLock(req);
                                                                            }
                                                                    }
                                                            }
                                                            else if (mode ==
RequestLock.LOCK_MODE.Write) {
                                                                    RequestLock req =
this.BlockedLocks.poll();
                                                                    LOGGER.log(Level.INFO, "[39]
"+request.getRequestId()+": Releasing Blocked Request - releasing the first WRITE request: "
                                                                    + req.getRequestId());

                                                                    addGrantedLock(req);
                                                            }
                                                    } else {
                                                            LOGGER.log(Level.INFO, "[36]
"+request.getRequestId()+": Releasing Blocked Requests - there are no Blocked requests for
File "
                                                                                    +
request.getRequestedFile().getName());
                                                    }
                                            }
                                    }
                                    //stop this thread and FINISH here
                                    request.setFinishTimestamp(System.currentTimeMillis());
                                    LOGGER.log(Level.INFO, "[28] "+request.getRequestId() + ": FINISHED! -
" + request);
```

```
				LOGGER.log(Level.INFO, "[29] "+request.getRequestId() + ": 
TIMESTAMPS: Start=" + Info.millisToTime(request.getStartTimestamp()) + ", "
								+ 
"Granted="+Info.millisToTime(request.getGrantedTimestamp()) + ", Finished=" + 
Info.millisToTime(request.getFinishTimestamp()) );
				String timestampDif = (request.getFinishTimestamp() - 
request.getStartTimestamp()) + "";
				LOGGER.log(Level.INFO, "[30] "+request.getRequestId() +": Execution 
time = " + timestampDif + " milliseconds");
				request.setRequestedFile(null);
			}
		}
		public void checkForStarvation() {
			int waitingRequestsNr = this.BlockedLocks.size();
			setStarvationNumber();
			if (waitingRequestsNr >= STARVATION_NUMBER) {
				while (! this.GrantedLocks.isEmpty()) {
					RequestLock requestToBeDroped = this.GrantedLocks.poll();
					requestToBeDroped.interrupt();
					LOGGER.log(Level.INFO, "[41] 
"+requestToBeDroped.getRequestId() + ": Request is dropped due to starvation, "+
							waitingRequestsNr+" requests are waiting for 
permissions for file "+requestToBeDroped.getRequestedFile());
					requestToBeDroped.setGrantedTimestamp(0L);
					requestToBeDroped.setStartTimestamp(0L);
					requestToBeDroped.setRequestedFile(null);
					requestToBeDroped = null;
				}
				//Set the lock bit of requested File to 0
				FileNode requestedFile = this.BlockedLocks.peek().getRequestedFile();
				requestedFile.setLockBit((short)0);
				LOGGER.log(Level.INFO, "[42] Lock Bit of file "+ requestedFile + " is set 
to 0 due to starvation");
				//start the first request on the Blocked List
				RequestLock requestToBeStarted = this.BlockedLocks.poll();

				LOGGER.log(Level.INFO, "[43] "+requestToBeStarted.getRequestId()+": 
Request is started after starvation of file "+requestedFile);
				addGrantedLock(requestToBeStarted);
		}
	}

	/**
	 * Get the starvation number from the gui txt field - to be moved to another class
	 */
	public void setStarvationNumber() {
		try {
			if (ClusterScene.starvationNumberTxt.getText() == null ||
ClusterScene.starvationNumberTxt.getText().equals("")){
				STARVATION_NUMBER = STARVATION_NUMBER_DEFAULT;
			} else {
				int starvationNumber =
Integer.parseInt(ClusterScene.starvationNumberTxt.getText());
				STARVATION_NUMBER = starvationNumber;
			}
		}
		catch (NumberFormatException ex) {
			Info.alertError("Number Format Exception", "Please insert an integer
for the random requests number!\n"+ex.getMessage());
			STARVATION_NUMBER = STARVATION_NUMBER_DEFAULT;
		}
	}
	public Queue<RequestLock> getGrantedLocks() {
		return GrantedLocks;
	}
	public void setGrantedLocks(Queue<RequestLock> grantedLocks) {
		GrantedLocks = grantedLocks;
	}
	public Queue<RequestLock> getBlockedLocks() {
		return BlockedLocks;
	}
	public void setBlockedLocks(Queue<RequestLock> blockedLocks) {
		BlockedLocks = blockedLocks;
	}
}
```

# CURRICULUM VITAE

## PERSONAL INFORMATION

**Surname, Name: Koçi, Artur**
**Nationality: Albanian**
**Date and Place of Birth: 08/01/1981 Burrel**
**Marital Status: Married**
**Phone: +35569 60 60 604**
**E-mail: akoci@epoka.edu.al**

## EDUCATION

| | |
|---|---|
| March 2013 – January 2019 | PhD Studies<br>Epoka University, Department of Computer Engineering |
| October 2009 – June 2011 | Master of Second Level in Computer Engineering<br>Epoka University, Department of Computer Engineering |
| October 2001 – July 2006 | Second Cycle Integrated Diploma in Electronic Engineering<br>Polytechnic University of Tirana, Faculty of Electronic Engineering, Department of Computer Engineering |

## ACADEMIC EXPERIENCE

| | |
|---|---|
| 25 October 2018 – 15 March 2019 | Part-time Lecturer<br>Metropolitan University of Tirana |
| 03 February 2014 –16 November 2015 | Part-time Lecturer<br>Epoka University |

## AWARDS

| | |
|---|---|
| Best Paper Award | 8th International Scientific Conference Computer Science`2018 |

## FOREIGN LANGUAGES

| Language | Speaking | Listening | Writing | Grammar |
|---|---|---|---|---|
| English | C1 | C1 | C1 | C1 |
| Albanian | C2 | C2 | C2 | C2 |
| Italian | B2 | B2 | B2 | B2 |

## PUBLICATIONS (Journals)

- Artur Koci, Betim Cico - ADLMCC – Asymmetric Distributed Lock Management in Cloud Computing. International Journal on Information Technologies and Security, No. 3 (vol. 10), 2018, pp. 37-52.
- Artur Koci, Betim Cico- Performance Evaluation of the Asymmetric Distributed Lock Management in Cloud Computing. International Journal of Computer Applications 180(49):35-42, June 2018.US
- Artur Koci, Betim Cico - Resource Starvation in Asymmetric Distributed Lock Management in Cloud Computing. Albanian Journal of Natural and Technical Sciences (47):91-102, November 2018. Albania

## ORAL PRESENTATIONS (Conferences)

- Artur Koci, Betim Cico Distributed Lock Management in Cloud Computing: Performance and Challenges – International Scientific Conference Computer Science'2018, 13-15 September 2018, Kavala - Greece.

- Artur Koci, Betim Cico. DDCMCC - Distributed Data Consistency Management in Cloud Computers - 7th International Scientific Conference Computer Science`2015 08-10 September 2015, Durres-Albania.

- Artur Koci, Betim Cico. Storage Based Cloud Computing and Disaster Recovery at DSC2014 - 9th Annual South East European Doctoral Student Conference that will be held on 25 - 26 September 2014 at Thessaloniki – Greece.