DEGREE PROJECT IN COMMUNICATION SYSTEMS, SECOND LEVEL
STOCKHOLM, SWEDEN 2015

Data Transfer and Management through the IKAROS platform

Adopting an asynchronous non-blocking event driven approach to implement the Elastic-Transfer's IMAP client-server connection

NIKOLAOS GKIKAS

KTH ROYAL INSTITUTE OF TECHNOLOGY
INFORMATION AND COMMUNICATION TECHNOLOGY

Data Transfer and Management through the IKAROS platform

Adopting an asynchronous non-blocking event driven approach to implement the Elastic-Transfer's IMAP client-server connection

Nikolaos Gkikas
gkikas@kth.se

2015-05-14

Master's Thesis

Examiner and Academic adviser
Professor Gerald Q. Maguire Jr. <maguire@kth.se>

Industrial adviser
Dr. Christos Filippidis <filippidis@inp.demokritos.gr>
Institute of Nuclear Physics
National Center for Scientific Research (NCSR) "Demokritos"
15310 Agia Paraskevi, Attica, Greece

KTH Royal Institute of Technology
School of Information and Communication Technology (ICT)
Department of Communication Systems
SE-100 44 Stockholm, Sweden

Abstract

Given the current state of input/output (I/O) and storage devices in petascale systems, incremental solutions would be ineffective when implemented in exascale environments. According to the "The International Exascale Software Roadmap", by Dongarra, et al. existing I/O architectures are not sufficiently scalable, especially because current shared file systems have limitations when used in large-scale environments. These limitations are:

- Bandwidth does **not** scale economically to large-scale systems,
- I/O traffic on the high speed network can impact on and be influenced by other unrelated jobs, and
- I/O traffic on the storage server can impact on and be influenced by other unrelated jobs.

Future applications on exascale computers will require I/O bandwidth proportional to their computational capabilities. To avoid these limitations C. Filippidis, C. Markou, and Y. Cotronis proposed the IKAROS framework.

In this thesis project, the capabilities of the publicly available elastic-transfer (eT) module which was directly derived from the IKAROS, will be expanded.

The eT uses Google's Gmail service as an utility for efficient meta-data management. Gmail is based on the IMAP protocol, and the existing version of the eT framework implements the Internet Message Access Protocol (IMAP) client-server connection through the ''Inbox'' module from the Node Package Manager (NPM) of the Node.js programming language. This module was used as a proof of concept, but in a production environment this implementation undermines the system's scalability and there is an inefficient allocation of the system's resources when a large number of concurrent requests arrive at the eT's meta-data server (MDS) at the same time. This thesis solves this problem by adopting an asynchronous non-blocking event-driven approach to implement the IMAP client-server connection. This was done by integrating and modifying the ''Imap'' NPM module from the NPM repository to suit the eT framework.

Additionally, since the JavaScript Object Notation (JSON) format has become one of the most widespread data-interchange formats, eT's meta-data scheme is appropriately modified to make the system's meta-data easily parsed as JSON objects. This feature creates a framework with wider compatibility and interoperability with external systems.

The evaluation and operational behavior of the new module was tested through a set of data transfer experiments over a wide area network environment. These experiments were performed to ensure that the changes in the system's architecture did not affected its performance.

Keywords

parallel file systems, distributed file systems, IKAROS file system, elastic-transfer, grid computing, storage systems, I/O limitations, exascale, low power consumption, low cost devices, synchronous, blocking, asynchronous, non-blocking, event-driven, JSON.

Sammanfattning

Givet det nuvarande läget för input/output (I/O) och lagringsenheter för system i peta-skala, skulle inkrementella lösningar bli ineffektiva om de implementerades i exa-skalamiljöer. Enligt ”The International Exascale Software Roadmap”, av Dongarra et al., är nuvarande I/O-arkitekturer inte tillräckligt skalbara, särskilt eftersom nuvarande delade filsystem har begränsningar när de används i storskaliga miljöer. Dessa begränsningar är:

Bandbredd skalar inte på ett ekonomiskt sätt i storskaliga system,

I/O-trafik på höghastighetsnätverk kan ha påverkan på och blir påverkad av andra orelaterade jobb, och

I/O-trafik på lagringsservern kan ha påverkan på och bli påverkad av andra orelaterade jobb.

Framtida applikationer på exa-skaladatorer kommer kräva I/O-bandbredd proportionellt till deras beräkningskapacitet. För att undvika dessa begränsningar föreslog C. Filippidis, C. Markou och Y. Cotronis ramverket IKAROS.

I detta examensarbete utökas funktionaliteten hos den publikt tillgängliga modulen elastic-transfer (eT) som framtagits utifrån IKAROS.

Den befintliga versionen av eT-ramverket implementerar Internet Message Access Protocol (IMAP) klient-serverkommunikation genom modulen ”Inbox” från Node Package Manager (NPM) ur Node.js programmeringsspråk. Denna modul användes som ett koncepttest, men i en verklig miljö så underminerar denna implementation systemets skalbarhet när ett stort antal värdar ansluter till systemet. Varje klient begär individuellt information relaterad till systemets metadata från IMAP-servern, vilket leder till en ineffektiv allokering av systemets resurser när ett stort antal värdar är samtidigt anslutna till eT-ramverket. Denna uppsats löser problemet genom att använda ett asynkront, icke-blockerande och händelsedrivet tillvägagångssätt för att implementera en IMAP klient-serveranslutning. Detta görs genom att integrera och modifiera NPM:s ”Imap”-modul, tagen från NPM:s katalog, så att den passar eT-ramverket.

Eftersom formatet JavaScript Object Notation (JSON) har blivit ett av de mest spridda formaten för datautbyte så modifieras även eT:s metadata-struktur för att göra systemets metadata enkelt att omvandla till JSON-objekt. Denna funktionalitet ger ett bredare kompatibilitet och interoperabilitet med externa system.

Utvärdering och tester av den nya modulens operationella beteende utfördes genom en serie dataöverföringsexperiment i en wide area network-miljö. Dessa experiment genomfördes för att få bekräftat att förändringarna i systemets arkitektur inte påverkade dess prestanda.

Nyckelord

parallella filsystem, distribuerade filsystem, IKAROS filsystem, elastic-transfer, grid computing, lagringssystem, I/O-begränsningar, exa-skala, låg energiförbrukning, lågkostnadsenheter, synkron, blockerande, asynkron, icke-blockerande, händelsedriven, JSON.

Acknowledgments

I would like to acknowledge the following persons whose help and support have made the completion of my thesis possible.

Firstly, I would like to express my gratitude to my academic adviser Professor Gerald Q. "Chip" Maguire Jr. who gave me the opportunity to conduct my master thesis in the NCSR “Demokritos”, in my hometown in Athens, Greece. His kindness, patience, and continuous guidance throughout entire process of writing this master's thesis has been greatly appreciated. Additionally, I would like to deeply thank my industrial adviser in the NCSR “Demokritos”, Dr. Christos Filippidis who gave me the opportunity to conduct my thesis project alongside with his research team and who willingly shared his precious time during all the stages of writing this thesis. Without his suggestions and help, the completion of this thesis would have been impossible. Moreover, I would like to express my gratitude to my colleague Spiros Danousis in the NCSR “Demokritos”, for helping me to deeply understand some parts of the eT framework's source code that were incomprehensible to me and for his willingness to assist me whenever I needed it. Furthermore, the contribution of Jimmy Svensson, master's student, who translated the abstract of my thesis into Swedish is highly appreciated. Last but not least, I would like to thank my family and Katerina Asimakopoulou for their continuous support throughout my life.

Athens, May 2015
Nikolaos Gkikas

Table of contents

List of Figures

List of Tables

List of acronyms and abbreviations

ASCAC	Advanced Scientific Computing Advisory Committee
AFS	Andrew File System
BDEC	"Big Data" and Extreme-scale Computing
CDC	Control Data Corporation
CMS	Compact Muon Solenoid
CERN	European Organization for Nuclear Research
DFS	Microsoft's Distributed File System
DOE	(US) Department of Energy
EGI	European Grid Infrastructure
ESFRI	European Strategy Forum on Research Infrastructures
eT	Elastic Transfer
EU	European Union
FIFO	first in, first out
FQL	Facebook Query Language
FTP	File Transfer Protocol
GPFS	General Parallel File System
GridFTP	Grid File Transfer Protocol
HDFS	Hadoop Distributed File System
HPC	high-performance computer/computing
HPCA	High-Performance Computer Architecture
IESP	International Exascale Software Project
iMDS	IKAROS meta-data service
I/O	input and output
JSON	JavaScript Object Notation
LAN	local area network
LHC	Large Hadron Collider
MDS	meta-data server
NAS	Network Attached Storage
NAT	network address translator
NFS	Network File System
NPM	Node Package Manager
PFS	parallel file system
pNFS	Parallel NFS
POHMELFS	Parallel Optimized Host Message Exchange Layered File System
PVFS	Parallel Virtual File System
PVFS2	Parallel Virtual File System, version 2
SCM	storage class non-volatile semiconductor memory
SMB	Server Message Block
SMP	symmetric multiprocessing
SOHO	Small Office/Home Office
SSD	solid state disk
UI	user interface
US	United States (US) (of America)
VO	Virtual Organization Policy
WAN	wide area network
WLCG	Worldwide LHC Computing Grid

1 Introduction

Global collaborative experiments generate datasets that are increasing exponentially in both complexity and volume. These experiments adopt computing models that are implemented by heterogeneous infrastructures, varying from local clusters, to data centers, high-performance computers (HPCs), clouds, and grids. The collaborative nature of these experiments demands very frequent wide area network (WAN) data transfers between these systems, however the heterogeneity between these systems usually makes scientific collaboration limited and inefficient.

The IKAROS framework was developed in order to overcome the limitations of today's systems, fulfilling the demands of future international collaborative experiments. To achieve this, IKAROS tries to combine features from both parallel and distributed file systems. More specifically concerning parallel file systems, IKAROS uses a similar architecture consisting of client-compute nodes, input and output (I/O)-storage nodes, and a meta-data management service. This meta-data service ''sits'' out of the data path between client and I/O nodes, thus it only controls meta-data and coordinates data access. Hence, the I/O procedures occur directly between clients and I/O storage nodes.

The IKAROS meta-data service plays a key role in the system's architecture. This service handles the meta-data differently based on the client's needs and can respond to a client's requests in three different ways. The required meta-data information may be found in the client's cache, in a local meta-data server (MDS), or in an external infrastructure. The external infrastructures IKAROS can exploit include existing cloud infrastructures for dynamically managing meta-data. For example Facebook or Gmail can be used as an external meta-data management utility.

In our case, the Elastic Transfer (eT) framework adopts the same logic and uses Google's Gmail service as an utility for efficient meta-data management. Gmail is based on the IMAP protocol, a common protocol for e-mail retrieval and storage. In this thesis project the Gmail IMAP client-server connection will be modified in order to achieve a more suitable meta-data management in real world environment.

This chapter gives a general introduction regarding the two predominant client-server architectures as well as the two common techniques to establish client-server communication. The chapter defines the specific problem that this thesis project addresses, and then presents the purpose and goals of this thesis project.

1.1 Background

A typical client-server architecture model consists of clients and one or more servers. Clients initiate a communication session with a server, which in turn provides some service(s) to the clients. Both parties may reside in the same computer; however, in most cases they are executing on different systems and communicate via computer networks. Clients use an interface to send their requests to a server. The server must be waiting for incoming requests from clients. A server provides a standardized interface for clients to communicate with it, thus these clients are unaware of the specific hardware or software utilized by the server. To provide its service(s), a server needs to execute one or more programs, while the clients can direct their requests to a specific service running on the server according to their needs.

Servers are categorized according to the services that they provide. For example, in its simplest form, web services on the Internet are based on a client-server model. The web server provides web pages and/or data to clients (which are typically browsers). Similarly, an e-mail server receives e-mail messages from e-mail client over the Internet and delivers them to e-mail clients, while a file server utilizes a shared disk to enable clients to store and retrieve data. Moreover, a physical server may provide more than one service; for example acting as web server, e-mail server, and a File Transfer Protocol (FTP) server at the same time.

Today, two predominant server architectures exist. The first is based on threads/processes and the second on events.

In the thread thread/process-based architecture all incoming requests have to be served sequentially through a specific thread or process that is dedicated to their execution. This model presents major scalability issues, since every single thread or process requires some amount of random access memory (RAM) for handling each request. Furthermore, this synchronous blocking I/O model leads to low system performance and limits the system's performance when a large number of concurrent requests arrive at the same server [1].

The event-driven server architecture was proposed as an alternate to the thread/process-based model. This architecture adopts an asynchronous non-blocking approach for managing incoming requests. Specifically, a single thread is responsible for handling client requests. In this model ‘‘event emitters’’ emit specific events, then these events are placed into an queue. Each event is coupled to a specific piece of code or procedure that awaits execution. This model achieves better performance with regard to I/O concurrency and reduces the resources that are required when a large number of connections from clients to the server are open at the same time[1].

In the client-server architecture two basic communication approaches are used:‘‘pull’’ and‘‘push’’.

The ‘‘pull’’[2] technique is is based on the request/response paradigm and it is typically used to perform data polling. Clients continuously request specific information from a server which has to serve all of these incoming requests. However, when the same client makes several sequential requests within a small time interval or when multiple clients make requests at the same information, then the system's resources become a bottle-neck. Moreover, if a high rate of requests persists for a period of time, then the server overloads.

The ‘‘push’’[3] model was developed to avoid these problems, hence this model is based on the opposite logic. In this model, the server initiates a connection with each of the clients, pushing them specific information. This model is based on the publish/subscribe/distribute paradigm and helps to conserve network bandwidth and avoid system overloading[2].

It is important to mention that both of the server architectures and both the ‘‘pull’’ and ‘‘push’’ approaches have their advantages and disadvantages. As a result different hybrid models combining elements from these technologies have been developed. Nevertheless, the adoption of the appropriate technique always depends upon the type of application that should be supported.

More information regarding server architectures and the ‘‘push’’ and ‘‘pull’’ approaches are included in Chapter 3.

1.2 Problem Definition

As mentioned earlier, eT uses Google's Gmail service as an utility for efficient meta-data management. Gmail uses IMAP protocol, for e-mail retrieval and storage. The existing version of the system, implements a traditional IMAP server-client scenario using ‘‘pull’’ logic and this approach was used for a proof of concept prototype when the system was initially developed. As a result, continuous requests arrive to the MDS asking for specific meta-data information.

However in a real-world use case, eT does not implement a typical client-server model scenario. The server that is used for the meta-data management may be surrounded by a number of peers which continuously update or request for specific meta-data information. The problems that arise due to the adoption of the pure ‘‘pull’’ logic can be understood by examing the cases in the following paragraphs.

When one user-client is connected to the system, this client may request meta-data from the server at specific time intervals. Nevertheless, if between these time intervals a large number of data transfers are performed between other peers, then the meta-data on the server is updated and the user-client will miss some parts of this updated meta-data. This shown in Figure 1-1.

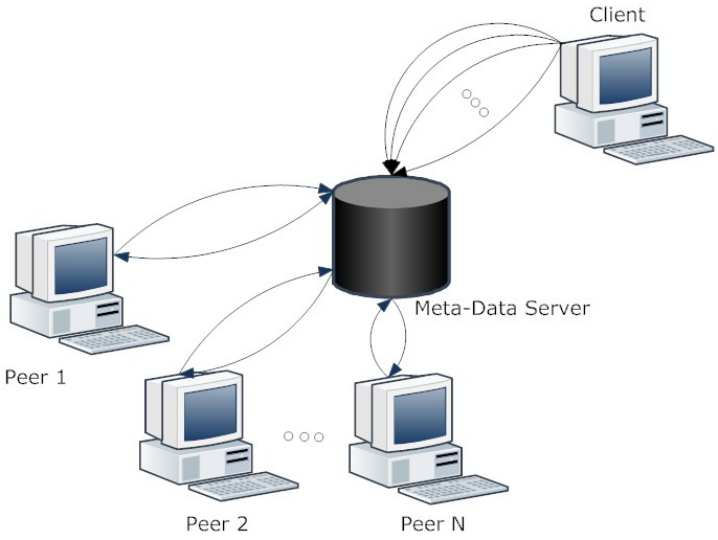

Figure 1-1: Client's continuous requests between specific time intervals and peers' data transfers/server's meta-data information update between this specific time intervals

A solution to this problem is to use the Network Time Protocol (NTP) for clock synchronization between client and MDS, or to save all states at every peer. However, when the number of peers increases, the worklow becomes complex and inefficient and the system may become overloaded.

When multiple users-clients connect to the system, the system's operation may also become unstable when the server has to serve multiple requests within a short period of time and the requests arrive at such a rate that not cannot they be serviced immediately. Moreover, even when there is a queue to enqueue the requests, some of them may be rejected if there is insufficient space for all of the requests. When there was enough space to enqueue them all, some of the enqueued requests could time out if the previous requests that are being service need a lot of time to be served (i.e., requests to databases and other operations that take a long time). This shown in Figure 1-2.

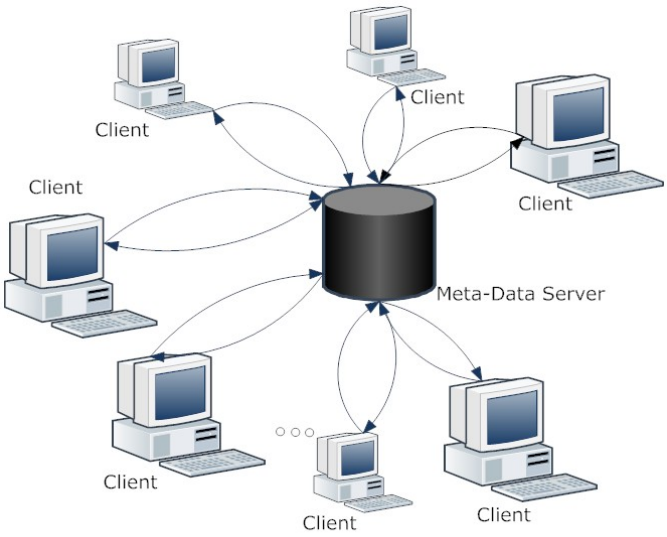

Figure 1-2: Multiple clients request the MDS for specific information within a short period of time

It is important to mention that even when the number of the active users is *low*, a very *large* number of jobs/requests can be generated from them. For example in the European Grid Infrastructure (EGI) just a relatively small number of users (thousands) generated at about 1.5 million jobs/requests on average per day in 2014 [4]. These jobs/requests can in turn trigger new I/O requests etc.

The following problems arise due to the current implementation of eT:

- The number of meta-data updates there are versus time (this is is proportional to the number of files created and their replication due to requests).
- The framework exhibits low scalability and cannot support a large number of active peers (users-clients, storage-I/O nodes).
- Even when the number of active users-clients is small there may be inefficient system resources, such as central processing unit (CPU) and RAM, since the MDS may have to serve a very large number of concurrent requests within a short period of time.
- Since the files are ‘‘spread out’’ to an increasingly large number of I/O servers there is not an I/O bottleneck, but rather a data distribution bottleneck.
- eT does not use a widely accepted meta-data scheme, which leads to low interoperability and incompatibility with other systems.

In this thesis project the term “client” may reffer, to specific client-users or I/O nodes, or to the requests in general that are generated from them.

Rather than adopting the ‘‘pull’’ logic, greater flexibility is achieved and the workflow of the system is simplified by adopting “push” logic - since the MDS informs all peers when the meta-data is updated.

1.3 Purpose

The purpose of this thesis is to expand the eT′s capabilities without affecting its performance. Initially, the eT framework will be modified to adopt an asynchronous non-blocking event-driven architecture for implementing the IMAP client-server connection. Furthermore, the new implementation will utilize ‘‘push’’ logic. This will be done by integrating and modifying the ‘‘Imap’’ module[5] from the

NPM[6] repository to suit the eT framework. This specific modification will fundamentally change the system's philosophy when a large number of requests simultaneously arrive to the MDS of the eT framework.

Additionally, to enhance eT′s features, the meta-data scheme of the system will be changed. Since JSON[7] has become one of the most widespread data-interchange formats and has become a commonly accepted standard for web applications and distributed infrastructures, eT′s meta-data scheme will be modified to suit this notation. After these changes the meta-data of the system will be easily to present as JSON objects. This feature will create a framework with higher visibility, along with greater compatibility and wider interoperability [8] with other systems.

1.4 Goals

The main goals of this thesis project are:

- To further develop the eT framework by changing the previous client-server implementation which was used as a proof of concept into an asynchronous event-driven approach that could be more suitable for real world use cases.
- To increase the system's interoperability and compatibility, enabling it able to interconnect with other external systems.

The changes have to be made without affecting the system's operational performance. To evaluated the performance of the modified module a set of data transfer experiments will be performed in a wide area network (WAN) environment. The experiments will be conducted to ensure that the system operates smoothly after the changes and no unexpected system behavior occurs while its is running. No significant changes in the data transfer performance of the system are expected since both versions of the system use the same protocol for data transmissions, i.e., Transmission Control Protocol (TCP).

1.5 Structure of the Thesis

Chapter 2 provides further background for the reader and summarizes the related work that has previously been done. Chapter 3 describes the methodology selected for this thesis project. Chapter 4 describes the changes that have been made in the eT source code to implement the event-driven asynchronous logic. Furthermore, the relevant changes in the meta-data scheme are presented. Chapter 5 describes the experimental procedure that was followed and includes the relevant experimental results. Lastly, Chapter 6, begins with a brief conclusion based on the experimental results, contains suggestions for future work that could be done to further boost eT′s capabilities and presents the required reflections of this thesis project.

2 Background

Chapter 2 gives a deeper introduction to the area. The four scientific paradigms are presented, a brief history of supercomputers is given, the significance the exascale systems follows, as well as a summery of the new technological and scientific challenges that have arisen over the past decade. This is followed by a ‘‘Data Challenges’’ section, giving a more precise definition of ‘‘Big Data’’ and a typical knowledge discovery process using ‘‘Big Data’’ is presented. Finally, the intertwined requirements for ‘‘Big Compute’’ and ‘‘Big Data’’ are introduced as well as the existing research projects and consortia that are working on the development of exascale computing.

Section 2.7 reviews the network, the parallel and the distributed file systems and the philosophy that lies behind their implementation. Thereafter, a brief presentation of the Grid File Transfer Protocol (GridFTP) is given in Section 2.7, since it consists one of the most well-known and widespread systems for remote data transfers. Additionally, the current frameworks' limitations are presented with a focus on the I/O system's bottlenecks.

Section 2.11 presents the IKAROS framework's which was developed in order to overcome the today's systems limitations, fulfilling the demands of the future international collaborative experiments. The IKAROS' approach and design goals from theoretical and technical aspect are presented as well as the framework's architecture and system design.

Finally, 2.12 provides some general information regarding the eT framework and 2.13 describes the fundamental aspects of the Node.js programming language.

2.1 The Four Scientific Paradigms

Historically, the two most significant paradigms for scientific research have been experiments and theory[9]. The former, ‘‘empirical science’’ was the paradigm that was used thousand years ago for the description of natural phenomena, while the latter emerged during the last few hundred years and used mathematical models, laws, equations, generalizations, etc. for the same purposes. During the last few decades, the development of large-scale applications for simulation of complex phenomena led to a third paradigm: ‘‘large-scale computer simulations’’/computational science[10]. Today's scientific computing capabilities have led to significant breakthroughs and large-scale experiments that were impossible to run several years ago, now create huge datasets. However, an unceasing production of information does not always lead to scientific discoveries. Data complexity and heterogeneity have undermined efficient data management and processing, becoming major challenges in the 21st century.

Over the past decade, a new paradigm for scientific discovery has emerging due to the exponentially increasing volumes of data generated from large instruments and collaborative projects. This paradigm is often referred to as ‘‘Big Data’’/‘‘data-intensive’’ science[9]. This new paradigm is tightly coupled to the concept of exascale computing. Unfortunately, existing technological limitations undermine a smooth transition to the era of ‘‘Big Data’’.

This fourth paradigm tries to exploit information ‘‘buried’’ in large datasets and has been introduced as a complement to the three existing paradigms. The complexity and challenge of this fourth paradigm arises from the increasing velocity, heterogeneity, and volume of data generation[9] from various sources such as instruments, sensors, supercomputers, large-scale projects, etc. Large amounts of data are usually accompanied by the challenges of ‘‘data-intensive’’ computing which synthesizes and unifies theory, experiment, and computation using statistics; where applications devote most of their execution time to input and output (I/O) when mining crucial information from massive datasets. The complexity of this processing increases when data search and computational analysis should be performed simultaneously.

Many different tools have been developed for data searching, analysis, and visualization, but new techniques and methods are required to simplify the workflow of ‘‘data-intensive’’ computing. Additionally, new challenges related with the optimization of data transfers need to be overcome. These new approaches have become even more vital to achieve an effective transition to the exascale computing era. The approaches that are examined to overcome these current limitations include[9]:

- fast data output from a large simulation for future processing/archiving;
- minimization of data movement across levels of the memory hierarchy and storage;
- optimization of communication across nodes using fast and low latency networks and optimization of this communication; and
- effective co-design, usage, and optimization of all system components from hardware architectures to software.

Seymour Cray and Ken Batcher are both believed to have independently stated that ‘‘a supercomputer can be defined as a device for turning compute-bound problems into I/O-bound problems’’[11] This half-serious, half-humorous definition is confirmed to be true today more than ever. While supercomputers gain parallelism at exponential rates and achieve high computational performance, their storage systems evolve at a significantly lower rate [12]. Therefore, storage has become the new bottleneck for large-scale systems or other collaborative projects that require efficient data management and transfer between computing and storage subsystems. Hence, an effective ‘‘data-intensive’’ computing approach is vital for the evolution of modern science, since the existing storage infrastructures faces a growing gap between capacity and bandwidth and future exascale computers will require I/O bandwidth proportional to their computational capabilities. Therefore, it is imperative to introduce new technologies that will lead to efficient data handling, visualization, and interpretation[12]. Existing shared file systems have limitations when used in large-scale environments, because [13]:

- Bandwidth does **not** scale economically to large-scale systems,
- I/O traffic on the high speed *network* can impact on and be influenced by other unrelated jobs, and
- I/O traffic on the *storage server* can impact on and be influenced by other unrelated jobs.

The IKAROS framework was developed to avoid these limitations. IKAROS combines in one thin layer utilities that span from data and meta-data management to I/O mechanisms and WAN data transfers. By design, IKAROS is capable of increasing or decreasing the number of nodes of the I/O system on the fly, without stopping current processing or losing data. IKAROS is capable of deciding upon a file partition distribution schema, by taking into account requests from users or applications, as well as applying a domain or a Virtual Organization policy (VO)[8].

The IKAROS framework is used by the Greek National Center for Scientific Research – “Demokritos” as a data transfer and management utility and it provides its services to local users, as well as to international experiments, such as the Compact Muon Solenoid (CMS)[14]of the Large Hadron Collider (LHC) [15], built by the European Organization for Nuclear Research (CERN)[16] and the KM3NeT consortium[17]. The KM3NeT experiment (both a European Strategy Forum on Research Infrastructures (ESFRI) project and a CERN recognized experiment) will introduce a distributed network of neutrino telescopes with a total volume of several cubic kilometers at the bottom of the Mediterranean Sea.

2.2 Exascale Computing Vision

Exascale computer development would be a major achievement in computer science. This generation of supercomputers would trigger the development of modern applications that could potentially be used to solve big scientific problems.

This following subsections present a brief history of supercomputers, the significance of future exascale systems, as well as the most significant technological challenges that have emerged in this evolution.

2.2.1 A Brief History of Supercomputers

By the middle of the 1940's, the world's first digital general purpose computer, ''ENIAC',' was developed in order to perform complex ballistic calculations for the United States Army. By the 1960's, electronic computers became more widespread and this led scientists to integrate them into their research projects, since they contributed to efficiently solving complex computational problems [18]. In 1964 the world's first supercomputer computer, called the ''CDC 6600'', was released by Control Data Corporation. The ''CDC 6600'' was a series of systems that through innovative techniques and parallelism achieved high performance and clearly exceeded earlier computational performance limits [19].

By the 1970's, further evolution of technology led to the development of more advanced systems with greater software and hardware capabilities. New challenges and opportunities led large numbers of engineers to contribute their improvements to computers, since computers could be integrated in various scientific areas by expanding the existing research methods[18]. These systems were implemented to address problems in various large-scale military, financial, and scientific fields. The ''Cray-1'' was released in 1976 and became the world's most successful supercomputer [19]. The next decade brought more sophisticated ''Cray-based'' computer systems, although each one of them used only a few processors.

In the 1990's, new approaches to HPCs resulted in the release of machines with thousands of interconnected processors, thus increasing the peak computational performance to gigaflop scale [18].

During the last fifteen years, HPC evolved further and the transition from the gigascale to terascale era occurred. A petascale system was introduced in 2008 having the ability to perform 10^{15} operations per second. Today petascale computing systems are widely used, for performing complex computations in various scientific fields, including climate simulation, astrophysics, cosmology, nuclear simulations, high-energy physics, etc.

2.2.2 Future Exascale Systems ("Big Compute")

The initiative for the development of exascale platforms has been endorsed by two United States (US) (of America) agencies[20], the Office of Science and the National Nuclear Security Administration, both of which are part of the US Department of Energy (DOE). In addition, the importance of exascale computing is also affirmed by the fact that the US government made the development of exascale systems a top priority[21], investing US$126 million in this area, in 2012[22].

Additionally, Japan, Europe, and various international scientific communities understood that exascale implementation would be a significant step towards solving today's complex scientific and technical issues. In exascale environments millions of computer nodes are interconnected and billions of concurrent I/O requests and threads are executed [21]. The significance of these systems to mankind becomes obvious if one considers that their processing capabilities will be similar to a human brain[21]. Consequently, the release of exascale computers will create new challenges for scientists and technological innovators, while offering solutions to many existing open problems including climate change modeling and understanding, weather prediction, drugs discovery, etc. [20], and expanding scientific research in various fields such as mathematics, engineering, biology, economics, and national security.

The amount of digital information has been increasing exponentially over the past decade. Only exascale platforms will be able to efficiently handle and analyze this large amount of information. The dimension of the problem becomes obvious when one considers that the the annual global IP traffic will exceed the zettabyte (10^3 exabytes) scale in 2016. More specifically, global IP traffic is expected to reach 91.3 exabytes per month in 2016 and 131.6 exabytes per month by 2018[23]. In general, global Internet traffic in 2018 will be equivalent to 64 times the volume of the entire global Internet in 2005[23] .

Today more and more projects, consortiums, companies, and governments aim to develop cutting edge exascale computing technologies.

2.2.3 Emerging Technological Challenges

It is obvious that every major technological transition creates new opportunities and challenges. Consequently, moving towards an exascale computing vision is unlikely to be an exception.

The area of High-Performance Computer Architecture (HPCA) emerged several decades ago, as researchers moved from the gigascale to the petascale computing era. In HPCA storage is completely segregated from computing resources and is connected via a interconnect network. Unfortunately, this approach will not scale up by several orders of magnitude in terms of concurrency and throughput, thus HPCA prevents the transition from petascale to exascale systems [21]. The requirements of current applications have changed and systems need to be re-architected in order to satisfy the ever increasing demands of researchers. Additionally, high power consumption and the latency of off-chip data transfer from CPU to RAM introduce additional major problems that need to be solved. Today, memory performance and high energy demands undermine the effectiveness of current technologies[22]. Systems without appropriate resources in terms of memory, processors, disks, and software are incapable of operating smoothly[24]. Moreover, existing grid computing implementations are not user friendly and the VO concept is inefficient for individual groups or small organizations[8].

According to DOE's Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, only coordinated research in different fields will offer feasible solutions. Co-design of applications, software, and hardware will lead scientists to much better use of the opportunities of exascale computing [24] . They further note that technological innovators should[24] :

Reduce energy consumption	Existing technologies can not be effectively applied to future exascale systems as the energy demands are so large that one gigawatt of power would be required for each machine. New technologies should be deployed to solve this problem.

Handle run-time errors	Since, exascale platforms will consist of a billion processing entities, even a small error frequency will lead to errors occurring much more frequently that for existing systems, hence error identification and correction would be an extremely time consuming process. Thus new techniques are needed to solve this problem.
Reclaim parallelism concept	Existing algorithmic, mathematical, and software concepts should be further developed in order to reach higher levels of concurrency. This would enable a major advance towards exascale implementation of solutions.

2.3 Data Challenges

The evolution of technology over the last decades, led to the development of large-scale computers, expanding scientific knowledge, and discovery in various fields. These systems are mainly applied in collaborative projects and generate a massive amount of data. Advanced instruments, such as colliders, telescopes, sensors, analysis, and simulation systems produce a huge amount of information, usually distributed over a heterogeneous collection of devices, in geographically dispersed areas.

This amount of data sometimes becomes so large that the complexity of dealing with this data exceeds the processing capabilities of traditional systems. Consequently, new challenges related to data management have arisen, as existing technologies are sometimes unable to effectively store, organize, access, analyze, process, and transfer these massive datasets.

The following subsection provides information related to these large, complex, structured or unstructured datasets that is referred to as ‘‘Big Data’’. In addition a typical knowledge discovery life-cycle for ‘‘Big Data’’ is presented.

2.3.1 ‘‘Big Data’’

The minimum amount of information that could be characterized as ‘‘Big Data’’ depends on the existing hardware and software capabilities and varies from several petabytes (1024 terabytes) to hundreds of exabytes (1024 petabytes). Unfortunately, conventional software and hardware solutions are incapable of offering feasible solutions when ‘‘Big Data’’ expands to exascale size[25].

The problem of ‘‘Big Data’’ management becomes clearer, when the experiments at CERN are examined. More specifically the LHC generates colossal amounts of data which annually total 30 petabytes[26]. The problems of ‘‘Big Data’’ will become more central in the near future since scientists have estimated that the amount of observational and simulation data related to climate issues will reach the exabyte scale by 2021 [9].

Buddy Bland, the project director at the Oak Ridge Leadership Computing Facility, stated that “there are serious exascale-class problems that just can not be solved in any reasonable amount of time with the computers that we have today” [27]. Consequently, innovative technologies need to be deployed in order to overcome existing limitations.

2.3.2 Knowledge Discovery Life-Cycle for ‘‘Big Data’’

The fourth paradigm exploits information that is ‘‘buried’’ in huge datasets by attempting to derive new knowledge and to trigger new scientific discoveries. The complexity of this process is directly related to the ever increasing amount and heterogeneity of data that is being generated. A typical knowledge discovery life-cycle for ‘‘Big Data’’ consists of the four following phases[9] (as shown in Figure 2-1):

1. **Data Generation**: The first phase of the cycle is concerned with data generation by instruments (such as telescopes, colliders, and sensors), computer simulations, or other sources. During this process, various procedures (including information reduction, analysis, and processing) could occur concurrently.
2. **Data Processing and Organization:** The second phase includes data transformation, organization, processing, reduction, and visualization. This phase is also related to external or historical data collection, distribution, and sharing. This phase includes data combination, which creates ‘‘data warehouses’’ for future use. Hence, during a discovery process, scientists want to be able to access, share, and exploit this existing data.
3. **Data Analytics, Mining, and Knowledge Discovery:** Given the size and complexity of datasets, sophisticated mining algorithms and software are deployed in order to find associations, relationships, and correlations between data. This processing can include performing predictive modeling and overall bottom-up/top-down knowledge discovery. The latter is generally adjusted according the problem that is under consideration.
4. **Actions, Feedback, and Refinement:** The last phase ‘‘closes’’ the knowledge discovery life-cycle. Using feedback from the three first phases, this phase derives results and conclusions that may in turn be forwarded to the first phase of the cycle. Consequently, this information may generate a new dataset for future experiments, influence upcoming simulations, observations, etc.

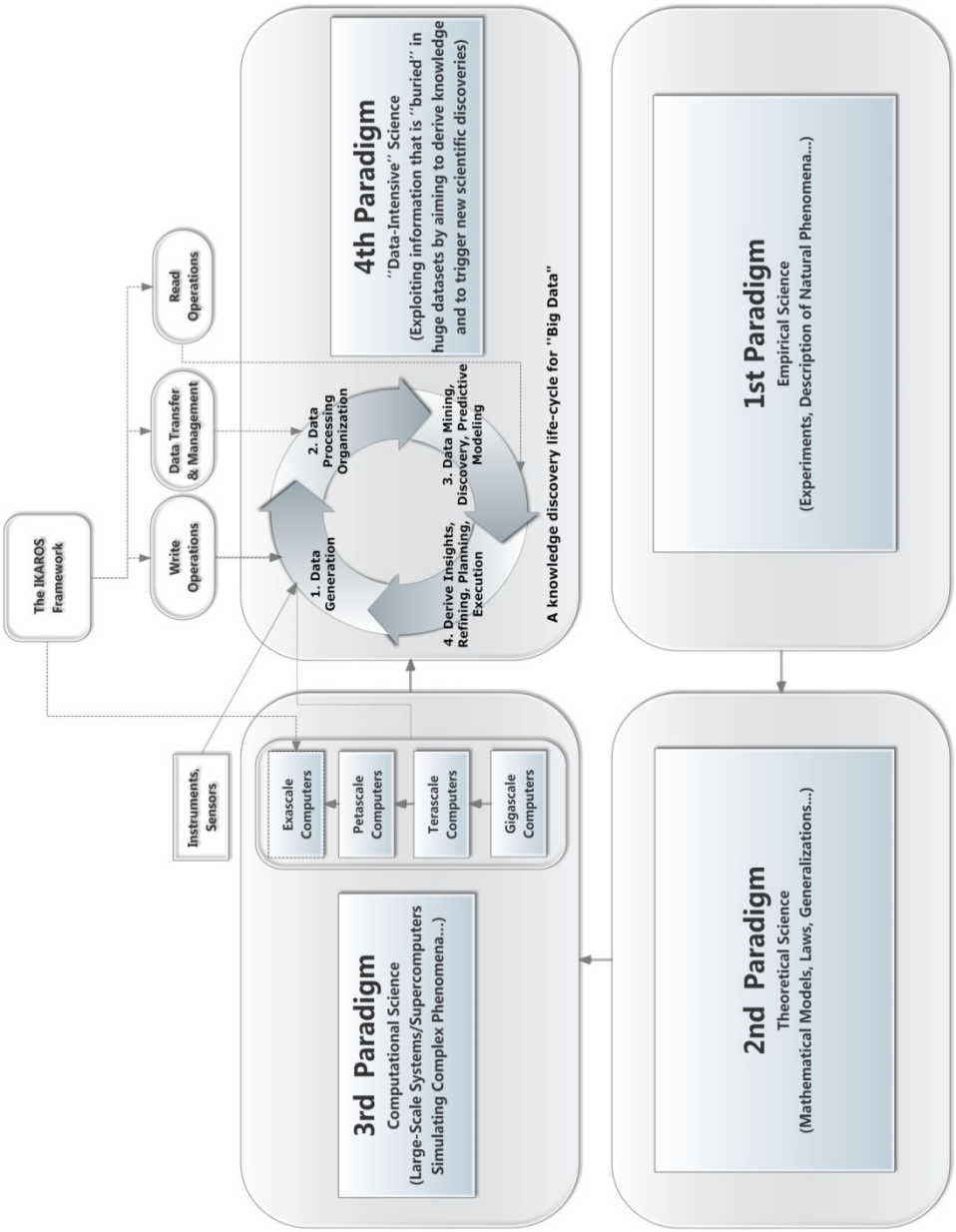

Figure 2-1: A conceptual depiction of the four scientific paradigms and their fundamental elements (The knowledge discovery life-cycle for ''Big Data'' has been adapted from Figure 2.1 of [9]).

2.5 Intertwined Requirements for ‘‘Big Compute’’ and ‘‘Big Data’’

The third and the fourth paradigm are based on the ‘‘Big Computing’’ and ‘‘Big Data’’ concepts. Nevertheless, it would be wrong to approach these two fields independently, as both contribute to shared scientific efforts and share requirements that are tightly coupled to the development of an exascale platform [9]. It is essential to bridge these two disciplines if scientists want to strengthen the exascale computing vision.

“Data-intensive’’ simulations on ‘‘Big Compute’’ exascale systems will generate very large amounts of data, just as existing large-scale experimental projects do. Similarly, the massive datasets that are produced by the ‘‘data-driven’’ paradigm need to be analyzed by “Big Compute” exascale systems [9].

‘‘Data-intensive’’ and ‘‘data-driven’’ mindsets have evolved independently, although they face common challenges, especially related to data movement, management, and reduction processes[9]. Consequently, it is crucial to exploit the synergies of these concepts.

Since the “data-driven” paradigm is relatively recent, current systems have been designed using workloads that are focused more on computational requirements than on data requirements [9].

The ‘‘compute-intensive’’ concept aims to maximize computational performance and memory bandwidth, assuming that most of working dataset will fit into main memory; while the ‘‘data-intensive’’ mindset focuses on a tighter integration between storage and computational components in order to effectively handle datasets that exceed traditional memory capabilities [9]. Consequently, memory hierarchies of future exascale platforms should be designed with greater flexibility in order to support both ‘‘Big Compute’’ and ‘‘Big Data’’ concepts. Early exascale systems are expected to be based mainly on ‘‘compute-intensive’’ architectures, however mature exascale implementations should integrate both concepts [9].

The need for “data-intensive” architectures is motivated by the fact that past architecture designs have been based on established workloads that pre-date the fourth paradigm [9]. The characteristics and requirements of data analytics and mining applications that are central to the fourth paradigm have not as yet had a major impact on the design of computer systems, but that is likely to change in the near future [9].

The US Department of Energy (DOE) ASCAC [28] is investigating the synergistic challenges in both exascale and ‘‘data-intensive’’ computing. Scientists have affirmed that there is a strong interrelation between these two fields and there are investment opportunities that can benefit both ‘‘Big Compute’’ and ‘‘Big Data’’ areas [9].

During the past, the simulation of a project’s processes was initially performed, and then the ‘‘off-line’’ data analysis followed. Today scientists have the ability to handle and explore petabytes of data, but exascale simulations will require data analysis to take place while information is still in ‘‘memory’’. The integration of data analytics with exascale simulations represents a new kind of workflow that will impact both ‘‘data-intensive’’ and exascale computing. Innovative memory designs, more efficient data management, and better solutions with respected to networking capabilities, algorithms, and applications should be proposed. Efficient data exploration, processing, and visualization will create a tighter correlation between data & simulation and increase future systems’ productivity, by better managing an ever-increasing processing workload [9].

In conclusions, it is obvious that co-design that includes requirements from ‘‘compute-intensive’’, ‘‘data-intensive’’, and ‘‘data-driven’’ applications should guide the design of future computing systems [9]. Based upon the discussion above a conceptual depiction of the four paradigms and of their components was presented in Figure 2-1.

2.6 Research Projects and Consortiums

Exascale platform development and has been endorsed the US, Japan, and Europe. Although, since exascale technology is still in an early stage, many research and development strategies at different levels of systems architecture and software are required in order to upgrade existing technologies. The US, Japan, and Europe have created consortiums and projects including CRESTA [29], International Exascale Software Project (IESP) [30], "Big Data" and Extreme-scale Computing (BDEC) [31], European Exascale Software Initiative [32], etc. in order to move towards the same goal: developing and evaluating potential solutions for a stable transition to the new exascale computing era.

CRESTA is an EU research collaboration project that aims to increase European competitiveness, making it the leader in world-class science problem solving, by deploying cutting-edge technologies. CRESTA's objective is based on two integrated solutions. The first focuses on building and evaluating advanced systemware, tools, and applications for exascale environments, while the second aims to deploy co-designed applications. Co-design is expected to provide guidance and feedback to the systemware development process[29].

The IESP consortium focuses on hardware and software co-design techniques as well as on the development of innovative and radical execution models. In addition, the group aims to produce an overview of the existing state of various national and international projects regarding exascale development by the end of the decade [30].

The BDEC community is premised on the idea that the recent emergence of "Big Data" in various scientific disciplines constitutes a new shift that may affect existing research and current approaches to the exascale vision. According to BDEC, "Big Data" should be systematically mapped out, accounting for the ways in which the major issues associated with them intersect and potentially change national and international plans and strategies for achieving exascale platforms [31].

In summary, the most significant motivation and research challenges that all relevant work has shown until now is that [8]:

- existing petascale systems are unlike to scale to exascale environments, due to the disparity among computational power, machine memory, and I/O bandwidth, and due to their increasing power consumption;
- traditional grid infrastructures are not user friendly and efficient, for small groups and individuals; and
- users demand efficient data sharing, mobility, and autonomy which leads to independent, but not exclusive control of resources.

2.7 File systems

This section presents some common types of file systems, introducing their fundamental principles and their major bottlenecks & limitations. More specifically, network, parallel, and clustered/distributed file systems are presented. Following this, the GridFTP protocol is reviewed, since it is one of the most widely used protocols for WAN based data transfers [22]. Lastly, the most significant limitations in current frameworks are presented, especially those limitations that relate to I/O issues.

2.7.1 Existing File Systems

Given the fact that random access memory is volatile and only stores information temporarily, computer systems require secondary storage devices, such as hard drives, optical discs, flash memories, etc., that can permanently retain information. Unfortunately, these secondary memories are characterized by high complexity, thus requiring operating systems to provide appropriate mechanisms for reliable and efficient data management.

A file system (see Chapter 40 of [33][*]) is the part of a computer's operating system that is responsible for data management. The file system implements a specific structure and logic for data storage and retrieval. Without a file system, all stored information would be part of an immense information object. File systems are responsible for splitting information into data chunks, then giving each piece a name in order for it to be easily identified and managed. Consequently, a file system is responsible for storage space allocation & arrangement, organizing data into files, and for accessing, retrieval, and modification operations. In addition, a file system creates and manages the meta-data related to files. This meta-data describes the contents of files, access rights, origins, date and time of creation/modification/access, etc.

Different types of file systems exist and each one is designed for specific circumstances, with its own properties regarding its speed, flexibility, security, maximum size, etc. File systems can be categorized into disk, flash, tape, database, transactional, network, shared, special purpose, minimal/audio-cassette storage, flat file, … file systems. Additionally, "virtual" file systems exist in order to provide compatibility between different file systems technologies, acting as a "bridge" between them.

To provided some of the necessary background for this thesis, the following paragraphs will review network, parallel, and distributed file systems.

2.7.1.1 Network File Systems

Network file systems allow a user on a client computer to use a remote file access protocol to access files over the network from a file server. Generally this file access protocol enables the client to manage files as if they are mounted locally. Examples of popular network file systems include: Network File System (NFS), Andrew File System (AFS), Server Message Block (SMB) protocols, and file system using clients utilizing the File Transfer Protocol (FTP) and WebDAV. NFS is a representative file system, hence it will be further analyzed in the following paragraphs.

NFS [34] is one of the most widespread centralized network file system protocols. NFS was originally developed by Sun Microsystems in 1984. NFS implementations are available for various operating systems. NFS is used for sharing resources between devices on a local area network (LAN), allowing users to have remote data access capabilities similar to how local storage is accessed[†]. Large amounts of data can be stored by a centralized server, but easily accessed by all clients.

The NFS file system server is responsible for delivering the requested file data to clients. This is done through a typical client-server network infrastructure via remote procedure calls. When a client wants to access a file it first queries the MDS which provides client a map of where to find the data. In a traditional NFS the MDS "sits" in the data path between client and I/O nodes and controls meta-data and coordinates access.

However the fact that every bit of data flows through the NFS server imposes major scalability issues. The system's scalability varies, depending on the server type and the infrastructure that is being used [35]. In some cases scalability issues occur, not so much due to software or system support, but rather media bottlenecks. For example when competing with heavy network traffic or when a large number of NFS requests arrive at a storage node within a short period of time, NFS slows down. Performance implications of sharing exist even with an extremely fast hard disk when there are hundreds of users. Scalability issues should be taken into consideration, especially when building large-scale infrastructures, where it is doubtful if NFS can reasonably support them [36].

A typical network using the NFS file system is depicted in Figure 2-2.

[*] http://pages.cs.wisc.edu/~remzi/OSTEP/file-implementation.pdf
[†] A primary reason for NFS's initial development was the relative cost of a network interface versus that of a high capacity disk. Since disks were expensive, it made sense to share them.

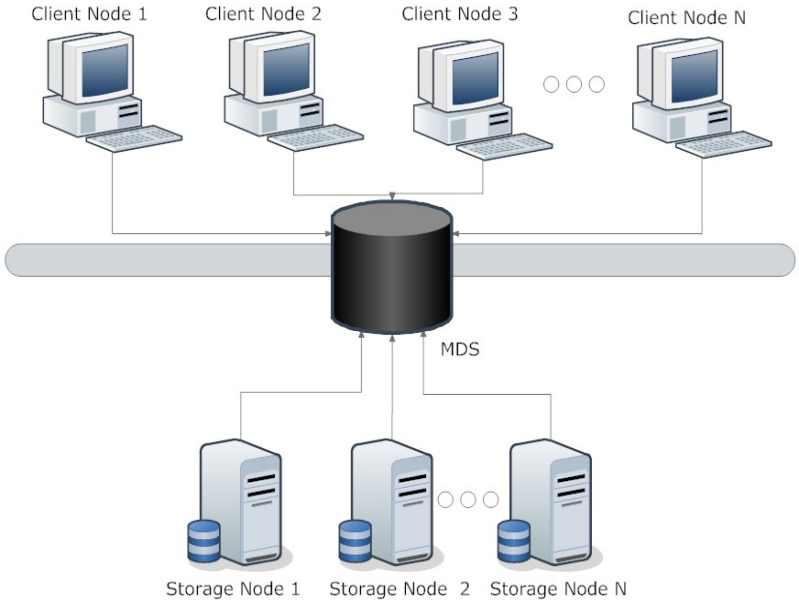

Figure 2-2: A typical network infrastructure using the NFS file system where the MDS "sits" in the data path between client and storage nodes.

2.7.1.2 Parallel File Systems

The I/O system's performance in HPC infrastructures has not kept in pace with their processing and communications capabilities. Limited I/O performance can severely negatively affect the overall system's performance, particularly in multi-teraflop clusters [37]. Moreover, HPC infrastructures share large datasets across multiple nodes and require coordinated high bandwidth I/O processes.

Parallel file systems are well suited for HPC cluster architectures, thus increasing their scalability and enhancing their overall capabilities. Parallel file systems are scalable as they distribute the data associated with a single object across multiple storage nodes. Parallelism makes concurrent data management from multiple clients feasible, allowing execution of concurrent and coherent read and write processes [37]. Parallel file systems are implemented on architectures where compute nodes are separated from storage nodes and where applications share access across multiple storage devices.

A parallel file system offers persistent data storage, especially when memory capabilities are limited, and provide a global shared namespace. Parallel file systems are designed to operate efficiently, with high performance over high speed networks, and optimized I/O processes to achieve maximum bandwidth [38].

A typical parallel file system implementation is depicted in Figure 2-3. It is consisted of client computing nodes, a centralized MDS, and storage I/O nodes. The I/O systems are ''grouped'' together, providing a global namespace, while the MDS contains information about how data is distributed across the I/O nodes. Moreover, the MDS includes information related to file names, locations, and owners [37]. When a compute node requests information, it sends a query to the MDS which replies with the requested file's location. Subsequently, the compute node retrieves the requested file from the relevant I/O nodes. Some parallel file systems use a dedicated server for the MDS, while others distribute the functionality of the MDS across the I/O nodes.

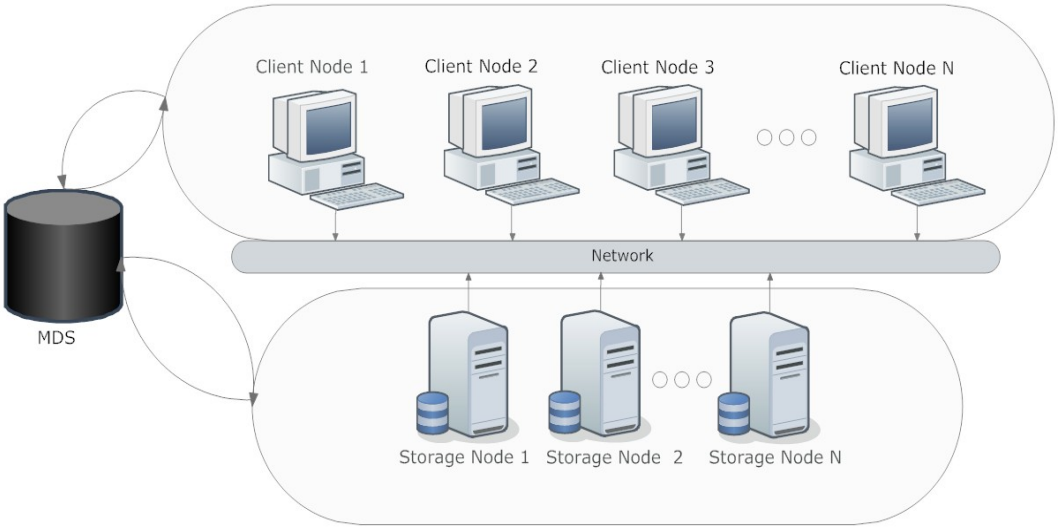

Figure 2-3: A typical shared and parallel file system with central MDS

Typical parallel file systems can operate smoothly up to petascale environments, although they cannot be effectively scaled to exascale platforms. As a result, to improve scalability the central MDS is replaced with decentralized meta-data formation (the computing and the storage nodes are implemented as in a typical parallel file system similar to that presented above). These distributed interconnected meta-data management nodes are better able to handle multiple client requests; however, significant synchronization errors still occurred when the number of concurrent requests expands to exascale size [21], [39].

The widely known Parallel Virtual File System (PVFS), General Parallel File System (GPFS), and Lustre provide scalability for the ever increasing demands of today. However, these file systems target homogeneous systems with similar hardware and software implementations[22]. Nevertheless, grid computing, legacy software, and other factors contribute to a heterogeneous group of customers, creating a gap between these file systems and their users [40]. The features of these file systems may be limited when applied to grid computer infrastructures, as these infrastructures are characterized by diverse software and hardware solutions. Unfortunately, the performance of these paralle file systems decreases as the variety of the underlying technologies increases.

Additionally, the majority of widespread shared and parallel file systems, including Andrew File System (AFS), PVFS, GPFS, Lustre, Panases, Microsoft's Distributed File System (DFS), GlusterFS, OneFS, Parallel Optimized Host Message Exchange Layered File System (POHMELFS), and XtreemFS employ a POSIX-like interface and have been adapted to clusters, grids, and supercomputing infrastructures. However, the fact that the underlying computing components are unaware of the data locality of the underlying storage system leads to significant criticism [21]. Additionally, since these protocols assume that the number of I/O devices and storage systems is much smaller than the number of the clients or the number of nodes accessing the file system, there is an unbalanced architecture for a ''data-intensive'' workload[21]. Parallel NFS (pNFS) is one of the most popular shared and parallel file systems, hence it is reviewed in the following paragraphs.

pNFS[41] is a step forward from the standard NFS v4.1 protocol, expanding its capabilities. It maintains NFS's advantages; however, it addresses its scalability and performance weaknesses. Since pNFS is capable of separating data and meta-data it can ''move'' the MDSs out of the data path. Additionally, pNFS gives clients the ability to access storage servers both directly and in parallel [42], thus fully exploiting the available bandwidth of a parallel file system [41]. It supports cluster infrastructures where simultaneous and parallel data management is required, allowing very high throughputs and ensuring a more balanced data load to meet clients' requirements [43], [44].

The rationale for pNFS is similar to that of the IKAROS framework, in that it tries to be universal, transparent, and interoperable; while taking advantage of NFS's widespread implementations. The functionality of pNFS is based on the NFS client understanding how a clustered system handles data.

Additionally, pNFS is designed to be used both for small and large data transfers. Data access is available via other non-pNFS protocols, because pNFS is **not** based upon an attribute of the data, but rather is an agreement between the server and the client [43], [44].

The pNFS architecture mainly consists of data and MDSs, clients, and parallel file system (PFS) storage nodes [41], [43], [44]. Since pNFS is representative of the state of the art parallel file systems, the rationale that unlies its implementation is typical for a parallel file system. When a client requests information, the MDSs replies with specific layouts, these provide information about the location of the corresponding data. In case of conflicting client requests, servers can recall these data layouts. If data exists in multiple data servers, clients' access can occur through different paths. Additionally, pNFS supports multiple layouts and defines the appropriate protocols between clients and servers [43], [44].

Nevertheless, pNFS requires a kernel rebuild against the pNFS utilities, because even when its default modules are loaded, additional adjustments should be made in order for it to operate smoothly. More specifically, pNFS requires an underlying clustered/distributed file system. Consequently, configuring pNFS requires considerable effort. Moreover, pNFS was designed for petascale systems, hence its meta-data entity does *not* scale up to future exascale storage systems.

2.7.2 Distributed File Systems

Distributed file systems store entire files on a single storage node and often run on architectures where storage is co-located with applications. These file systems are responsible for fault-tolerance and are geared for loosely coupled distributed applications. Moreover, distributed file systems are able to balance the load of the file access requests across multiple servers [45]. Different distributed file system protocols having been developed in order to support ‘‘data-intensive’’ computing, these include: GFS, Hadoop Distributed File System (HDFS), Sector, Chirp, MosaStore, Past, CloudStore, Ceph, GFarm, MooseFS, Circle, and RAMCloud. However, many of these are closely connected to specific execution frameworks, such as Hadoop. Consequently, applications that do not use these execution frameworks need to be modified in order to be compatible with these non-POSIX compliant file systems [21]. Nevertheless, even those distributed file systems that provide POSIX-like interfaces lack distributed meta-data management. Even those file systems that do support meta-data management fail to decouple data and meta-data, leading to inefficient data localization [21]. A typical infrastructure which includes network, parallel, and distributed file systems is depicted in Figure 2-4.

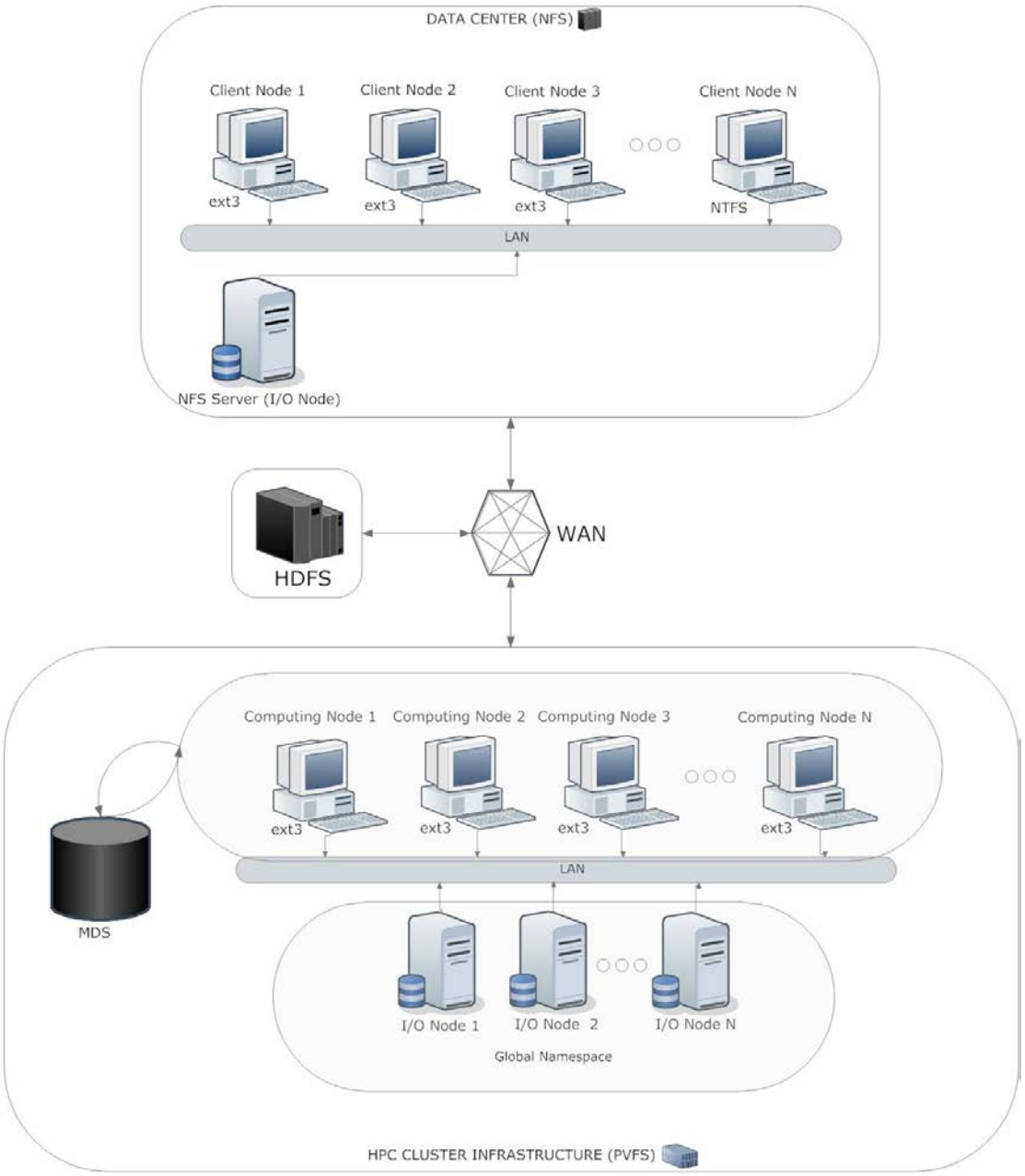

Figure 2-4: A typical infrastructure including network, parallel and distributed file systems.

2.8 The GridFTP WAN Data Transfer Protocol

Different protocols exist to provide fast and reliable data transfers via WANs. GridFTP is one of the most widespread protocols for large data transfers and it is further analyzed in the following paragraphs.

GridFTP [46] is an extension of the standard FTP protocol [47]. By default, FTP creates separate data and control TCP connections. GridFTP extends this feature, using multiple parallel TCP streams to facilitate multiple data streams, leading to increased data throughput. In addition, GridFTP's stripped servers increase its efficiency [48]. GridFTP provides authentication, data confidentiality, and integrity leading to secure and reliable data transfers. GridFTP supports third party high-throughput transfers between distributed sites interconnected by WANs, extending FTP's capabilities [46]. Furthermore similar to FTP, GridFTP supports partial file transfers and has the ability to restart failed data transfers [48]. Moreover, since GridFTP is based on the globally spread FTP protocol, GridFTP provides a universal data access method. These key features have made GridFTP one of the most widely used WAN protocols globally, where it has been utilized in large-scale collaborative scientific projects and grid computing infrastructures with high-throughput demands.

2.9 Limitations of Existing Frameworks

Filippidis, Markou, and Cotronis have identified the most significant limitations in current frameworks as [49]:

- While supercomputers gain increasing parallelism at exponential rates, the storage infrastructure performance is increasing at a significantly lower rate.
- The data management and data flow between the storage and compute resources is becoming the new bottleneck for large-scale applications.
- Bandwidth does not scale economically for large-scale systems.

2.10 Limitations in I/O Systems

As mentioned earlier, the exponential growth of digital information that emerged over the last decade, made data storing, mining, analysis, and processing an extremely complex process. Data analytics and mining are interdisciplinary subfields of computer science and are highly related with the fourth paradigm. Unfortunately, their requirements did not have a major impact on systems design until now [9]. This is likely to change in the near future since the disparity among computational power, machine memory, and I/O bandwidth are significant bottlenecks that undermine the effective transition to exascale platforms. The existing mindset in petascale environments is to write data to persistent storage on hard disks and then later read this data from these disks for further analysis. The widening gap between I/O and computational capacity in exascale systems makes existing supercomputers incapable of efficiently handling the desired amount of data, hence systems are unable to process this data in a reasonable amount of time as data access is severely limited by low performance I/O [9].

Raicu, Foster, and Beckman believe that storage systems in future exascale systems will be their Achilles heel, unless these storage systems are re-architected to provide scalability to millions of nodes and potentially billions of concurrent I/O requests [21]. The existing weaknesses can be clearly understood by trying to extend a supercomputing infrastructure, such as the Blue Gene/P to a million nodes. Taking into consideration its booting time on 256 processors is 85 seconds, while on 160K processors it takes 1090 seconds. Unfortunately, the machine's boot time grows linearly with the number of nodes, translating to potentially over 25K seconds (7^{+} hours) boot-time for 1M nodes [50]. Additionally, during the booting period the system has to communicate with and initialize all the corresponding nodes before informing the user that the file system is ready for use. The system has to check whether all nodes are up and running, if errors occur and if so, how these errors will be

recovered. Even when errors do not occur during the system's initialization, frequent errors may occur during the system's use.

Since incremental improvements to the current I/O and storage systems technologies would be inapplicable to an exascale environment, current infrastructures needed to be re-designed and evolve. Additionally, the current approaches to architectures, software, and applications should change in order to exploit the full advantages of exascale systems. Traditionally, I/O architectures operate separately from computing components which undermines their scalability. I/O procedures are viewed and implemented as independent activities, executed before or after the main simulation or analysis computation, or periodically for activities such as checkpointing, but still incur their own separate overhead [51].

While parallel file systems have reached high I/O throughput rates, they are targeted to specific operating systems and hardware. Moreover, they frequently lack reliable security mechanisms and most of the present systems fail to provide transparent and scalable remote data access [41]. When designing new hardware and software systems, I/O activities should be architected as an integral part of the design activity. File and storage system interfaces, or even higher-level data libraries in some cases, mostly ignore the purpose of the I/O by an application, despite this being important for scaling I/O performance, especially when millions of cores simultaneously access the I/O system [51]. Resiliency of an application to failures in an exascale system will depend greatly on the characteristics of I/O systems, because saving the state of the system in the form of checkpoints is likely to continue [51]. It is crucial to mention that the important metrics of I/O systems are performance, capacity, scalability, adaptability to applications, programmability, fault resiliency, and support for end-to-end data integrity [51].

In any case, the advent of new technologies such as solid state disks (SSDs), storage class non-volatile semiconductor memories (SCMs), etc. should create new opportunities to improve existing I/O architectures, systems, ans software performance, as well as to reduce power consumption.

2.11 The IKAROS Framework

The aim of IKAROS is to maintain the interoperability in grid infrastructures that was achieved during the past decade, while at the same time expanding their existing capabilities and developing a framework that will allow individual users or groups to create synergies. IKAROS attempts to avoid the limitations mentioned in the previous sections, while giving applications the ability to run on a wide variety of computing environments, ranging from a strictly defined cluster computer structure to a large grid computing environment [22].

Large collaborative experiments typically use computing environments consisting of local clusters, data centers, HPCs, cloud, and grid infrastructures. Each computing model is made up of different ‘‘tiers’’ that in turn consisted of several computer centers, each of which provide different services and use different software solutions for data processing. The computing varies from serial to multi-parallel or GPU-optimized jobs [52].

In addition to problems meeting the I/O challenges described in the previous sections, the existing grid tools and frameworks cannot fulfill the requirements of next generation collaborative experiments. The nature of these experiments demands efficient meta-data management and frequent remote data transfers, along with sharing and publishing between different users or groups. These issues should be taken into consideration when building large-scale distributed infrastructures, but existing grid infrastructures do not give users or individuals the ability to fully exploit their features. Most of the time, computing resources do not adopt a common philosophy, thus scientists have to deal with different policies and technologies [52].

Although, grid technology was developed to provide an effective de-centralized structure, it still uses centralized control for data handling, especially when it comes to the VO level. Centralized

control provides decentralized control management of CPU slots and storage, but most VOs still handle data in a centralized way, a fact that leads to significant criticism [52]. Cloud and Web 2.0 technologies enable dynamic management and sharing of data, similar to how allocation of CPU slots and storage space is now performed.

Consequently, a hybrid model should be created, with the ability to fully exploit the strengths of clouds and Web 2.0, while at the same time maintaining interoperability, high quality of service, and multi-institutional/multi-domain cooperation. These characteristics are inherited from grid infrastructures, such as the EGI and gLite (used by the CERN LHC experiments and other scientific domains). IKAROS was developed to provide flexibility, based on tools that can fulfill the demands of next generation international collaborative experiments [52].

Details of the IKAROS framework will be presented in the following subsections. Specifically, the first subsection presents the system's approach to existing problems, whereas the second and the third subsections review the IKAROS framework's design goals from a theoretical and technical points of view. The final subsection analyzes the framework's fundamental operating principles, architecture, and system design, and presents typical upload and download data scenarios.

2.11.1 The IKAROS Framework's Approach

The IKAROS file system was developed to overcome the bottlenecks described in the previous sections. The IKAROS framework takes the following general approach [49]:

- Remove the barriers to the overall data flow,
- Permit *ad hoc* nearby storage formations,
- Provide better interaction with users, and
- Use a huge number of low performance, low power consumption I/O nodes in order to increase the available bandwidth, while decreasing the overall power consumption.

Each of these elements will be described in the following subsections in terms of the design goals needed to realize this framework.

2.11.2 IKAROS Framework's Design Goals

In this subsection, the design goals and challenges of the IKAROS framework are outlined [8], [22], [53]. These goals provided the guidelines for the framework's development. In particular, the IKAROS platform aims to:

- Remove the barriers to the overall data flow (including local and remote access),
- Create synergies between wider scientific communities,
- Economically scale bandwidth in relation with the storage system's capacity, creating infrastructures that consume dramatically lower amounts of energy, and
- Overcome existing scaling limitations regarding meta-data mechanisms.

Each of these will be described in the following paragraphs.

2.11.2.1 Remove the barriers to the overall data flow (including local and remote access)

Usually, the processes that are required to provide services in large-scale distributed computing infrastructures are developed independently, unlike the processes that are executed in order to provide services within the narrow limits of a local distributed infrastructure. Consequently, in a local infrastructure processes are strictly defined and general-purpose techniques are *not* used to develop and implement the complete infrastructure. Inarguably, the development of specialized technologies to meet specific requirements could lead to excellent results when applied to small-scale infrastructures.

However, it seems that the extension of this thinking to large-scale infrastructures is ineffective, as it quickly reaches its limits and is unable to deliver significantly better results. This fact becomes more obvious when considering the case of grid infrastructure technology. The development of grid computing was aspired by the Internet, and was expected to evolve from an infrastructure which simply distributes information to an infrastructure that would be capable of sharing computing and storage power in a transparent way. When developing large or global scale distributed infrastructures, the logic of the fragmented development of services and of their full ‘‘isolation’’, was adopted. Technically this was implemented based upon interoperability of the individual heterogeneous infrastructures.

Existing distributed infrastructures adopt different data management methods, depending on their network environment. In local scale distributed infrastructures applications handle data by performing I/O via a parallel file system. In contrast, in WAN environments when there is a reference to data, the performance is mostly focused on data transfer via the network rather than I/O performance. In this way isolated ‘‘islets’’ are created, requiring communication bridges or multiple synchronization layers. The parallel streaming channels for remote data management that have been implemented, lead to very high data transfer rates. However, the parallel data transfer concept cannot be maintained in the next logical layer, which concerns I/O processes in the file system layer since the two infrastructures are ‘‘isolated’’ from each other, with the local network being agnostic to the resources and structure of the remote network. Although, modern file systems operate mainly using parallelism techniques, in reality two different phases are required to carry out the process. During the first phase, the data are routed in the remote network, but since the I/O nodes of the local infrastructure typically use private IP addresses they are *invisible* to the remote network. Hence, a continuous data flow between the local and the remote network cannot be achieved. As a result, there is an intermediate synchronization stage where the data flow is ‘‘reorganized’’ in order to be routed to the I/O nodes of the local infrastructure [52]. The results is that in reality ‘‘two levels of parallelization’’ and multiple levels of meta-data management exist. As the local and remote data access functions are ‘‘isolated’’, remote data must initially be transferred and only then can it be processed.

In contrast, IKAROS has been built as a thin layer to provide its services to multiple logical layers. In this way, a continuous data flow is achieved using parallelism techniques that are applied over the whole ‘‘route’’, without requiring intermediate synchronization levels. IKAROS bypasses the server bottleneck, enabling users to access storage directly [22]. The overall data flow can be managed efficiently since the framework provides unified logic between local and remote data access [53]. Users are able to route data through the network layer, since the framework gives the ability to expose the I/O data at any level. The I/O nodes of the local infrastructure use private IP addresses, but the reverse HTTP techniques that IKAROS uses provide the ability to transfer data directly from the local to the remote network.

IKAROS has been compared with the Parallel Virtual File System, version 2 (PVFS2) and HDFS, since they are both state of the art parallel and distributed file systems. They both have a similar high level design as user-level file systems and store data & meta-data in different types of servers [22]. Hence, IKAROS shares some features of these two file systems.

The logical difference between these two technologies can be better understood by examining Figure 2-5 and Figure 2-6. Figure 2-5 presents a data transfer scenario through a typical network infrastructure, whereas Figure 2-6 depicts a data transfer case through the IKAROS platform.

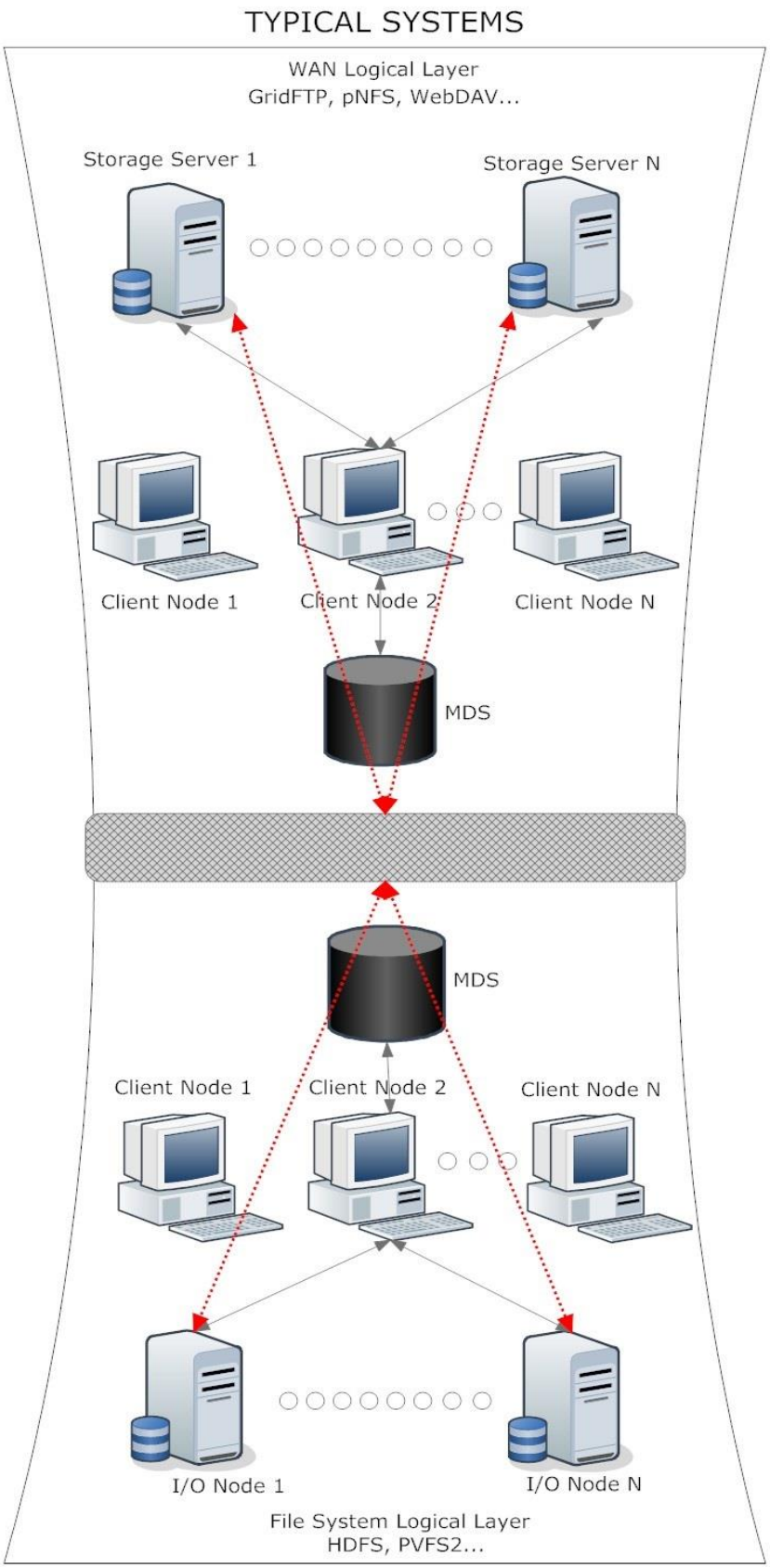

Figure 2-5: A typical data transfer scenario through traditional systems (Adapted from Figure 1 of [53]).

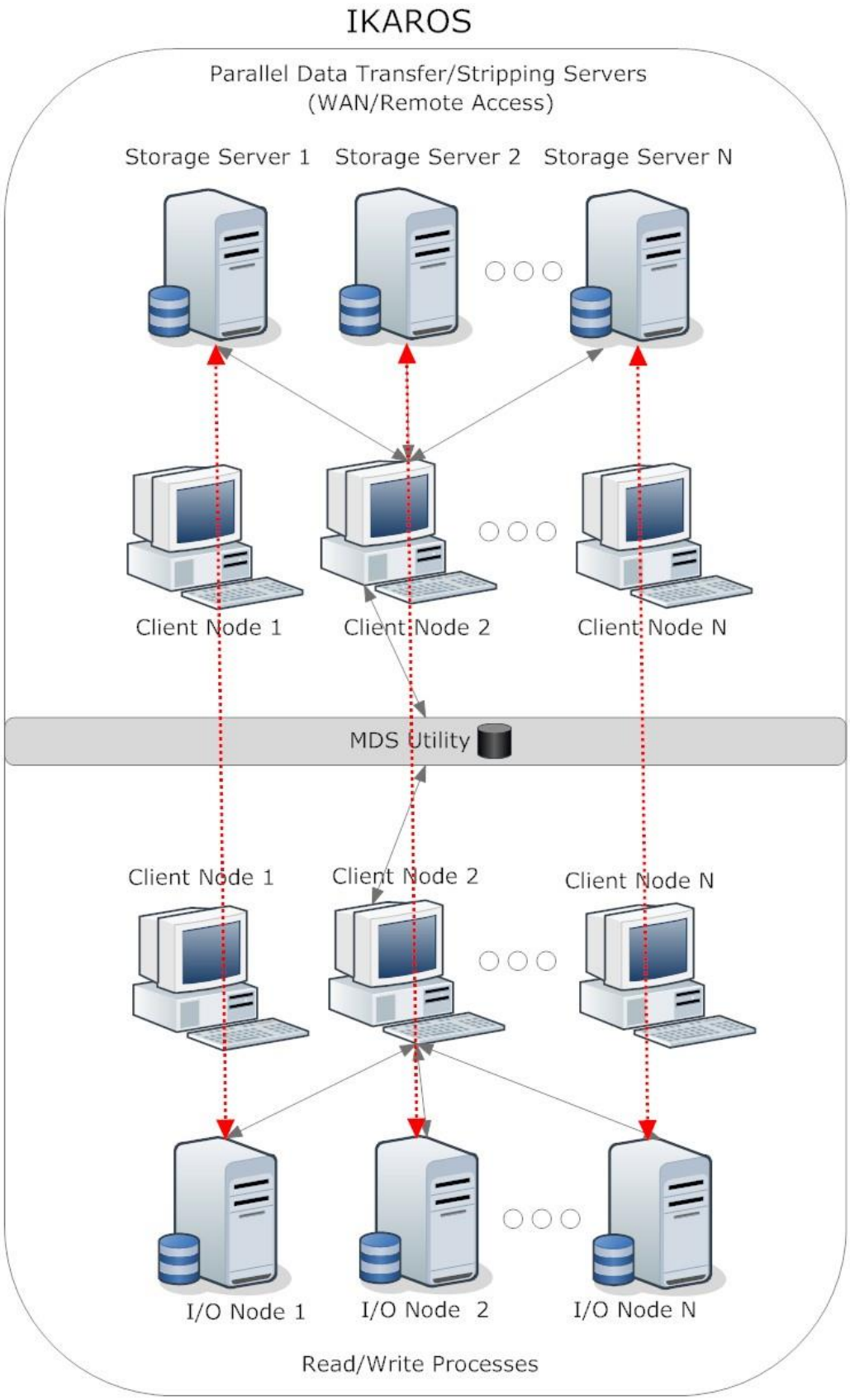

Figure 2-6: A data transfer case using the IKAROS framework (Adapted from Figure 1 of [53]).

2.11.2.2 Create synergies between wider scientific communities

It is obvious that infrastructures can not always be homogeneous , thus the services that are implemented should be able to operate independently — rather than depending on other implementation layers. This ‘‘isolation’’ is not accidentally, as older implementations that attempted to address common issues and depended upon services of a layer did not have the desired results. This service specialization led to high performance with respect to individual issues and allowed systems to meet application requirement’s of the large-scale applications of the last decade.

As mentioned earlier, scientific groups have indicated that the existing infrastructures will not be able to meet the application requirements that needed to be implemented within the decade. Consequently, current technologies should be redesigned. This means that current petascale systems need to evolve to exascale infrastructures. The broader scientific community has concluded that it is crucial to place the interoperability between heterogeneous infrastructures in a wider context, in order to exploit synergies across wider communities.

The rationale behind the development the IKAROS framework is that it aspires to handle these issues, by creating a platform that affords greater synergies. IKAROS tries to identify functional similarities between *different* logical layers. Additionally, this framework aims to maintain interoperability over heterogeneous infrastructures. This interoperability is actually provided through the autonomy of different layers’ services. In this layering there is an attempt to recognize common features and ‘‘behaviors’’ that will lead to the creation of infrastructures characterized by common practices and low cost implementation.

HTTP is the basic mechanism underlying the IKAROS platform. This choice gives IKAROS the ability to create synergies between wider communities and from a software perspective provides the ability to realize uniform operation in all logical layers. At the same time, the autonomy of different layers’ services can be maintained. It is strongly believed that the choice of HTTP will have a dominant role in whether the infrastructure helps to create wider synergies or not.

Additionally, security breaches do not usually occur due to erroneous security protocol choices, but rather occur due to the fact that those who are entrusted with securing data or infrastructures are not the ones who will suffer the consequences of such a breach. This implies that mechanisms that provide users with the ability to directly manage their data and the underlying infrastructure should be developed. The IKAROS framework fully embraces this approach, by allowing the creation of hybrid infrastructures that combine both the non-trivial quality of service of existing cloud computing infrastructures and the high usability of Web 2.0 technologies.

2.11.2.3 Economically scale bandwidth in relation with the storage system’s capacity, creating infrastructures that consume dramatically lower amounts of energy

With the development of large scientific experiments, such as the LHC, it was observed that the processes that are implemented through these experiments depend more and more on data in order to carry out the processes. As the amount of data required for a process’s execution expands to the terabyte scale, it is obvious that the respective processes should be transferred and implemented *within* the data centers that host the actual data collection(s). While in contrast, moving the data to where the processing will take place would be inefficient.

The distribution of data and resources offers a number of advantages, compared with a centralized computing or data storing infrastructure. This grid infrastructure approach, offers the opportunity to concurrently execute applications, in *multiple* servers, rather than executing them in a central symmetric multiprocessing (SMP) server. Additionally, a central SMP server would cost far more than a multiple server infrastructure. Consequently, this approach leads to better use of existing hardware infrastructures and with many small systems spread across various locations, facilitates collaboration between different organizations. Moreover, through grid computing, a very efficient use of idle resources can be achieved, as processes can be directed to idle systems. In addition, the existence of

multiple resources may prevent a general failure from occurring as might occur in a single centralized infrastructure. If one system within the grid fails, its workload can be forwarded to another system. Moreover, this approach offers scalability as new systems need to be added to or removed from the infrastructure. Furthermore, processes can be executed in parallel, leading to increased performance — allowing sophisticated problems to be solved in a shorter time. Policies are managed through grid software. Grid software performs the necessary calculations, monitors the resources, and distributes the work load over the available resources, according to the VO's internal policies. Hence, the user submits a process to the system *without* having to bind to a specific machine for its execution. Lastly, upgrading a system can be done on the fly *without* the need to shut all of the existing subsystems down. In this way, updates can be cascaded, without affecting the operation of the whole system.

When this operating model is further analyzed, it becomes clear that the existing distributed computing infrastructures should be converted from global-scale infrastructures with geographically distributed resources into a centralized facility limited to just a few active computing centers. In order to enable an infrastructure to accommodate the expected volume of data and to enable the infrastructure to support the necessary growth, specialized storage infrastructures and skilled personnel are required. Today these conditions cannot be met by small or medium size organizations.

The LHC has been funded and built in collaboration with over 10,000 scientists and engineers from over 100 countries, as well as hundreds of universities and laboratories. The LHC experiment generates 600 million particle collisions per second, producing about 30 petabytes of data per year[26]. The computing power of the whole infrastructure is equivalent to 100,000 of today's computer systems with the total storage space consisting of 400,000 large disks. It is obvious that a centralized infrastructure or a small number of mega-data stores would be cost-prohibitive, energy inefficient, unstable, and inflexible.

Only a hierarchical distributed implementation model exploiting parallelism, will be able to analyze this amount of data efficiently. As a result the Worldwide LHC Computing Grid (WLCG) was formed to address this issue. WLCG is a global collaborative project consisting of a grid-based network infrastructure. As of 2013, the system has become the world's largest computing grid, with over 400,000 CPU cores at over 350 sites, in over 50 countries around the globe. It is crucial to mention that LHC's problems were solved at a technical level in the past decade using grid computing technologies.

The problem of data management becomes more imperative when considering the problem of scaling the existing infrastructure to exascale environments. Data management and data flow between computational and storage resources, imposes major limitations on large-scale applications. This happens because storage systems face a huge gap between capacity and available bandwidth. Moreover, if nodes, with similar power consumption performance to the existing petascale computer needed to expand to a scale of millions of such computers, it would impossible to meet the total system’s energy requirements. The challenges that have arisen are enormous and have major economic, environmental, and technical impacts. Issues related to the location and the cooling of the infrastructure have a central role in the actual implementation. Scientists have reached a dead end, as the number and capacity of the available storage nodes needs to dramatically increase in order to meet the system’s requirements; however, at the same time making this increase using today's technology would increase the system’s energy demands to prohibitive levels.

IKAROS, responding to these challenges, exploits low performance and low energy consumption storage devices, to create high performance storage formations that allow a “loose” relation among themselves. The aim is to disentangle functions, as well isolate the devices from the malfunctions that may occur, within the scope in which different appliances function.

As to the low performance storage appliances, we refer to low performance storage devices, such as Small Office/Home Office (SOHO) and Network Attached Storage (NAS) systems. The present file systems, even when they claim to support the use of common commercial products, do not consider these type of appliances. In contrast, we assume the use of medium and high performance systems

that are widely available in the market (i.e, they are commodities). The use of a large number of SOHO-NAS appliances to create a high performance storage system, may be highly efficient, as each of the devices is designed to have a very low cost of ownership, very low energy consumption, and these devices are considered to be plug-and play appliances. Moreover, the use of very large numbers of these devices can increase the total available aggregate bandwidth. More specifically, a SOHO-NAS system costs as little as 1/5 of the cost of a typical storage system with the same storage capacity, while consuming 1/3 of the energy required by a typical storage system of the same capacity [22].

For example, in order to create a storage system of 100 TB, using SOHO-NAS appliances costs approximately 6,000 €. The whole system is estimated to consume 330 Watts of power. In contrast, a typical storage system of the same size may cost more than 35,000 € and consumes more than 800 Watts of electrical energy. This is an example of a relatively small system, but expanding this system to petascale size only linearly increases the cost and the power consumption. In this way, a one-million-node network could be created, while the electric power consumption of the whole system decreases and the available bandwidth will be sufficient. Moreover, with such a system there is no need for initialization and restructuring as there is in a typical storage system.

The IKAROS framework can efficiently manage low performance storage devices, such as SOHO and NAS systems, and fully exploits their advantages. The IKAROS approach leads to a more balanced architecture, where storage nodes are comparable to clients or computing nodes [22].

2.11.2.4 Overcome existing scaling limitations regarding meta-data mechanisms

Existing file systems organize digital files under a hierarchical directory tree. Although, this data management method becomes inefficient when the scale of data increases, various tools have been developed in order to address the relevant issues, thus providing a more efficient data organization by correlating the relevant data with explicit meta-data attributes. These mechanisms allow users to manage their files more effectively; although, the usage of a single meta-data entity can become a severe performance bottleneck, especially when applied in large-scale storage systems. Existing petascale systems use multiple meta-data nodes to improve scalability; however, even this implementation presents significant drawbacks due to the multi-tiered architecture of the storage devices. As data is moved between the various tiers of storage and/or modified, the overhead incurred for maintaining consistency between these tiers and the meta-data nodes becomes very large. As scientific systems continue to expand towards exascale this problem will become more central [54] and initialization of meta-data entities may be impossible.

The present meta-data mechanisms operate statically without taking into consideration the dynamics of the application. The meta-data entity of IKAROS allows us to build structures that initialize their subsystems independently, depending on their needs. The meta-data mechanisms of IKAROS play a dominant role in the whole infrastructure, as they allow the creation of common methods of troubleshooting of the general data flow, allowing at the same time for independent realization of different services. Moreover, the logic of the meta-data entity leads to the creation of structures that reduce the gap between capacity and available bandwidth. IKAROS aims to be a framework which allows the creation of a new generation of storage systems, which can satisfy the demands (described above), with regard to initialization, efficiency, and energy consumption. The overall goal can be achieved by building techno-economic mechanisms that allow cooperation between broader communities.

2.11.3 IKAROS Framework's Design Goals from a Technical Perspective

These following design goals guided the development of the IKAROS framework [22]:

- Provide interoperability between heterogeneous operating and storage systems.
- Able to add new resources to the existing infrastructure without increasing its complexity. New nodes can be added to or removed from the existing I/O components.
- Operate efficiently in both LAN and WAN environments. The architecture should be optimized to work effectively at scales ranging from a cluster to a grid computing infrastructure.
- Able to create different types of meta-data for the same data, depending on the specifics of the network environment.
- Reach high data transfer rates (i.e., throughput), satisfying even the most demanding applications.
- Allow the physical and logical views of a file to be independently configured.
- Give users and applications the ability to to choose their own file distribution system or to decide upon the distribution schema according to their requests, as well as domain and VO policies.
- Support third-party data transmission.
- Place the lowest possible workload on the low performance and low power consumption devices.
- Include all of its distributed file systems utilities as well as LAN and WAN properties in a *single* layer (in contrast with other common file systems); leading to higher performance with less configuration effort.
- Independently scale its subsystems.

2.11.4 IKAROS Framework's Architecture and System Design

IKAROS is an HTTP based distributed file system and management service for efficient data transfers. Since HTTP is the basic building protocol for IKAROS, the system is compatible with various operating systems. The choice of HTTP as a base gives the system the ability to use different protocols, operating systems, applications, and technologies that are already used on a world wide web scale, enabling the system to interact with millions of nodes. IKAROS has the ability to add new or remove existing nodes on the fly, without bringing everything down and without causing data loss. Due to the multitude of HTTP clients the storage devices can be accessed in many ways , including through the WebDAV (Davfs) (an extension of the HTTP protocol) which provides interoperability, without having to build an extra layer to support POSIX-like file system utilities.

IKAROS adopts a decentralized approach, separating compute and I/O nodes. The advantage of this framework is that it operates without needing multiple levels of data synchronization or reorganization of the overall data flow (local-remote access). This method, as well as the use of client-side buffering leads to higher bandwidth and guarantees durability and consistency for the write processes. The client-side caching that IKAROS framework supports, fully exploits HTTP's optimization for small files. Additionally, IKAROS exposes the mapping of chunks to applications and users, thus providing optimized concurrent writes to different regions of a file and accepting full writes anywhere.

This framework provides parallel data channels and exploits stripping server techniques with the same efficiency as GridFTP. It also supports third-party data transfers and has the ability to communicate with the GridFTP through the GridFTP application user interface (API) [22]. Although it is uncommon to combine LAN and WAN features, the framework adopts this mindset, thus making *every* node part of a global network infrastructure, bridging geographically dispersed and heterogeneous networks.

IKAROS has the ability to co-locate compute and storage on the same physical node or to separate them in the underlying infrastructure. The framework is able to create *ad hoc* nearby storage formations, through multiple instances of small capacity, high bandwidth storage utilities. By placing high bandwidth storage close to compute nodes the interference between I/O traffic and other unrelated jobs can be avoided. Furthermore, this scheme can provide greater aggregate bandwidth compared to the bandwidth that an external globally shared file system can provide. These file systems are efficient when applied to high performance systems; however, when the bandwidth requirements expand to exascale size, even more scalable solutions are required. The nearby storage approach provides flexibility and scalability in terms of cost and bandwidth and can be applied to both small and large infrastructures. Nevertheless, this technique does not provide file cache coherency, distributed locking, or other semantics that other shared filed systems provide (moreover, sometimes these semantics are not required).

When comparing IKAROS with other systems, a big advantage of the framework is the way it implements write operations. At the same time IKAROS performs read operations well, by using a reversed read technique the framework can fully utilize the available bandwidth. Moreover, compared to existing technologies the framework needs less configuration effort and provides a more cost and power effective solution.

With regard to the IKAROS architecture, the framework consists of three different types of nodes: user interface (UI)/client nodes, meta-data nodes, and I/O nodes. Each of these nodes is a peer and depending upon the user requests the node acts at as UI/Client, meta-data, or I/O node. Therefore, a single node can acts any combination of these different types of nodes at the same time. IKAROS is designed as an Apache Dynamic SharedObject (DSO) [55]. The UI/Client node type is not a typical client, but similar to a gLite UI provides services to many users. IKAROS can be accessed directly from a browser or another HTTP client, such as curl.

When a user sends a request to the IKAROS client (for example from a browser, another HTTP client, or UI/Client node) this triggers the IKAROS module to download data from the storage system to the client. The module communicates with the relevant meta-data node which provides information regarding the file partition distribution schema. Finally, the client establishes communication with the relevant I/O nodes and the data transfer begins. This process is presented in Figure 2-7 which depicts the high-level view of the IKAROS system.

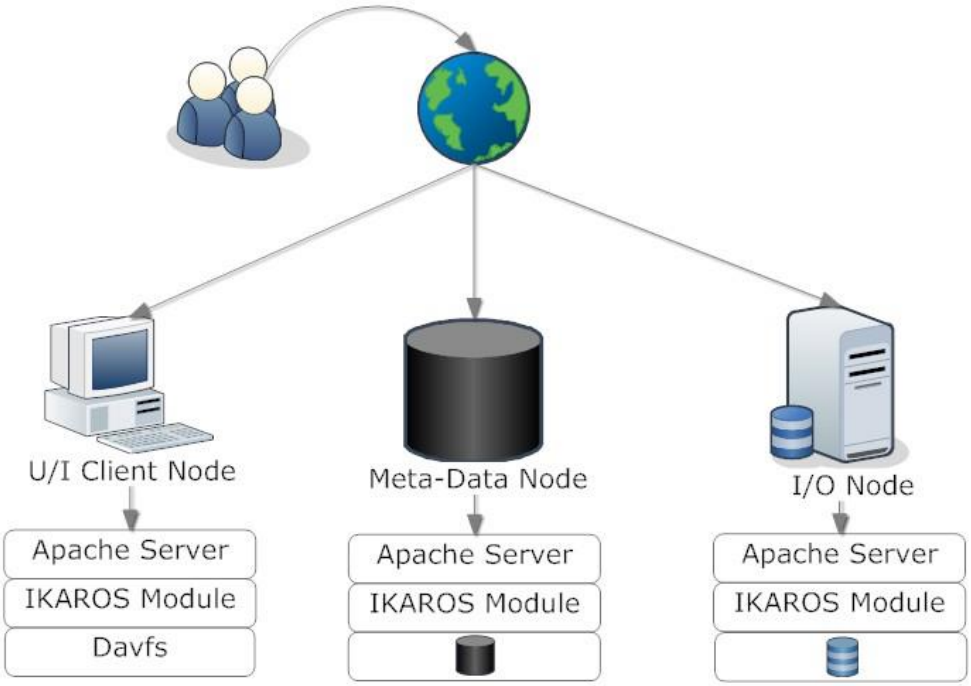

Figure 2-7: The IKAROS framework architecture (Adapted from Figure 1 of [22]).

The following paragraphs describe two basic data transfer scenarios. The first one describes a data upload from a client to a storage device, while the second one presents a data download case from a storage device to a client.

2.11.4.1 Data upload from a client to a storage device

A typical data upload procedure consists of the six following phases (depicted inFigure 2-8):

1. The UI/Client node sends a request to the IKAROS framework which in turns sends a request to the meta-node, asking for information regarding the available I/O nodes.
2. The meta-data node reply provides the requested information regarding the available I/O nodes.
3. Based on the I/O nodes' availability, the UI/Client node decides upon the appropriate file partition distribution scheme.
4. Then, the UI/Client node triggers the I/O nodes to send an "HTTP" request.
5. Based upon the file partition distribution scheme, the I/O nodes perform an "HTTP GET" request asking for a file partition.
6. Finally, the UI/Client node, informs the meta-data node about the data distribution to the relevant I/O nodes.

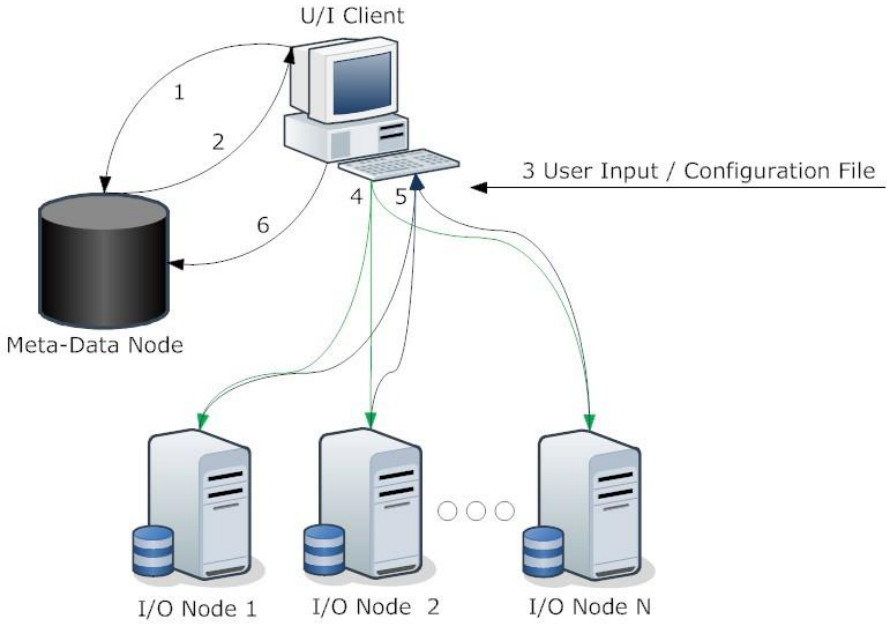

Figure 2-8: **A typical upload file case from a client to the storage nodes (Adapted from Figure 6 of [22]).**

The reversed read technique was used in steps 4 and 5 of this file upload scenario, as the UI/Client instead of performing a HTTP PUT request, triggers the I/O nodes to perform an HTTP GET request to the UI/Client. As a result the client-server's roles are reversed. This reversal also means that the storage node can be located in a private IP address space behind a network address translator (NAT) or firewall. Figure 2-9 depicts the requests between the nodes as well as the corresponding data flow.

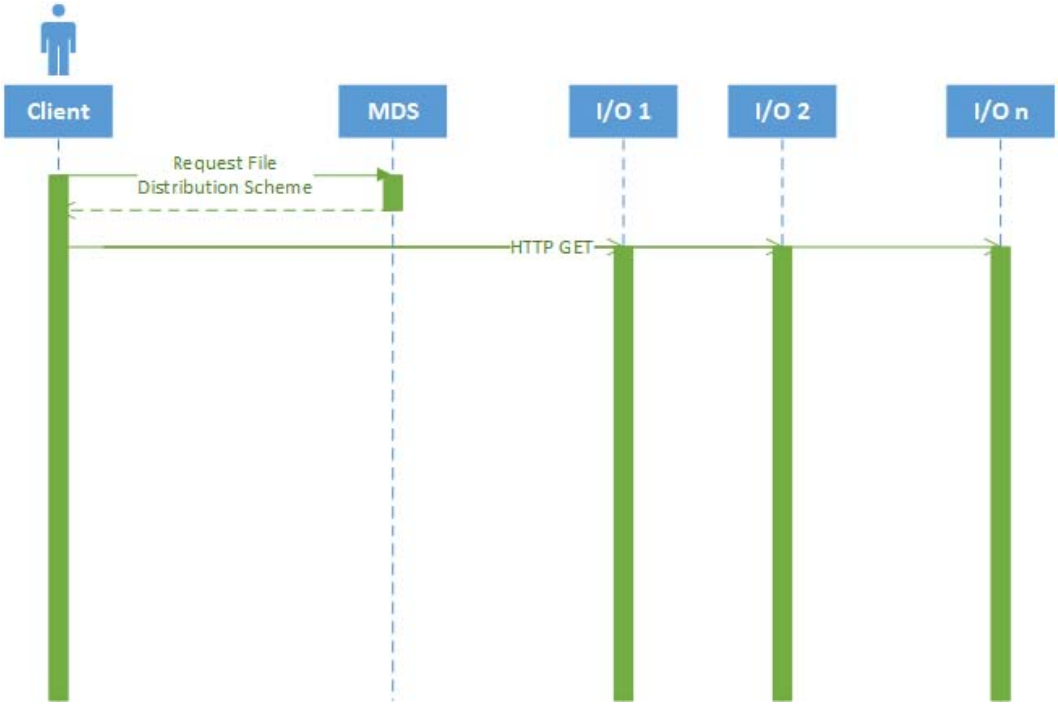

Figure 2-9: **The sequence diagram of a typical upload file scenario from a client to the storage nodes.**

2.11.4.2 Data download from a storage device to a client

A typical data download procedure is presented Figure 2-10 and consists of the following phases:

1. The UI/Client sends a request to the IKAROS module which in turns sends a request to the meta-data node asking for the relevant file partition distribution scheme.
2. The meta-data node replies to UI/Client, providing the requested information regarding the file partition distribution scheme.
3. The UI/Client sends an ‘‘HTTP GET” request to the I/O nodes.

Finally, the UI/Client node copies the HTTP buffer to the corresponding position of the local file. The relevant sequence diagram which describes the download file scenario is presented in Figure 2-11.

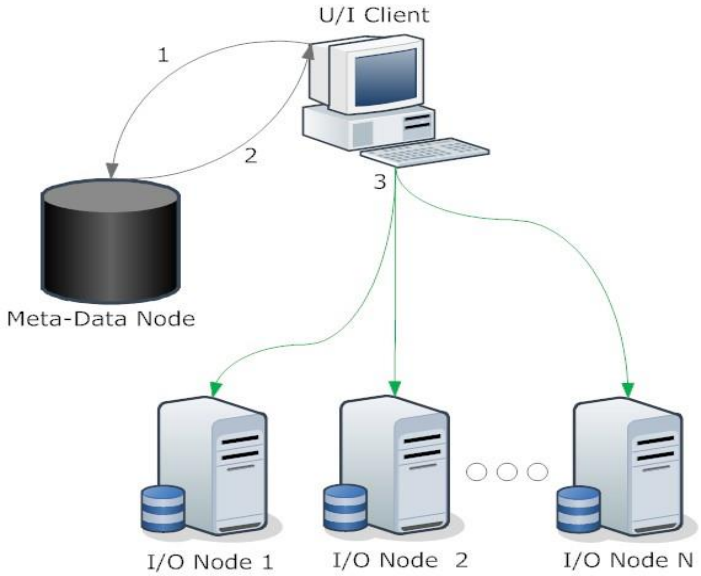

Figure 2-10: A typical download file case from the storage nodes to a client (Adapted from Figure 7 of [22]).

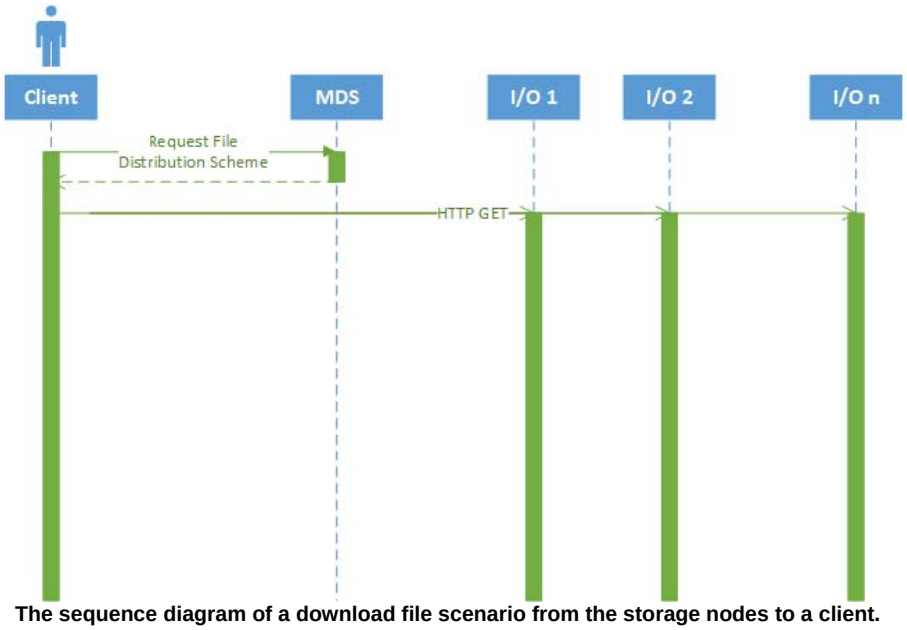

Figure 2-11: The sequence diagram of a download file scenario from the storage nodes to a client.

The IKAROS meta-data service (iMDS) plays a significant role in the system's architecture, allowing the meta-data sub-systems to be handled *independently* according to users' needs. This service allows users to create virtual scalable storage formations, giving users the ability to unify different computing facilities, from personal storage devices, to local computing clusters, to data centers, to HPCs, and to grid infrastructures [52]. IKAROS allows users to create user-driven facilities, thus enabling users to have a more active role in governance and placing ''Big Data'' at the center of scientific discovery [52]. Furthermore, this meta-data management approach provides several configuration parameters to the framework's I/O mechanisms.

The framework may respond to a request by accessing its own cache, by searching locally at a nearby MDS, or by searching externally at another MDS. Consequently a flexible and scalable model is built, providing cost effective solutions for both small and large-scale networks. For the purpose of this thesis, we focus on the framework's MDS utility in order to extend its capabilities with respect to its underlying file system. Further details regarding the system's meta-data concept are depicted in Figure 2-12.

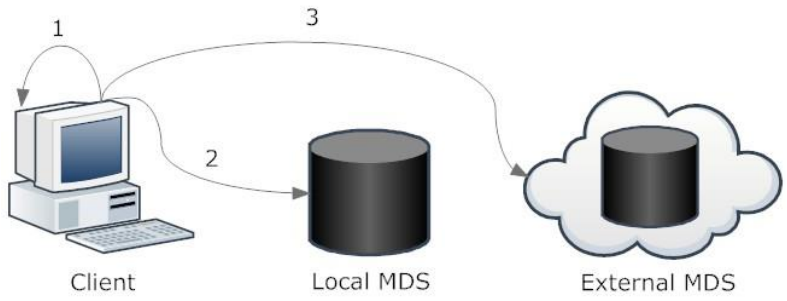

Figure 2-12: The IKAROS meta-data management service (Adapted from Figure 2 of [53]).

By using existing social networking services or cloud infrastructures, which act as external MDS utilities, it is possible to dynamically manage, share, and publish meta-data. More specifically, by adopting Facebook's or Gmail's meta-data management system, the existing infrastructures can be fully exploited and it is unnecessary to build new utilities for searching, sharing, and publishing meta-data. Users have the ability to use existing infrastructures to create efficient *ad hoc* storage formations.

IKAROS has adopted the JSON standard format (which is a very common Web 2.0 technology) for populating meta-data. More specifically the meta-data are created as JSON objects and are stored local in the relevant meta-data nodes. Users or groups can dynamically change their meta-data management theme according to their requirements. IKAROS is responsible only for the core services and Facebook* (or another customized meta-data management system) provides only the meta-data utilities. IKAROS keeps the responsibilities separated; thus the two systems can scale independently of each other and better performance of both read and write operations is achieved.

Figure 2-13 depicts how IKAROS successfully connects and interacts with the Facebook social network, which scales to millions of users. As a result, users are able to create storage formations on the fly and scale the platform subsystems independently based on their own needs. In this hybrid model, the IKAROS client initially connects with the Facebook App for authentication. After a successful authentication, the client requests the file distribution scheme from the Facebook Query Language (FQL) API, which in turn sends a reply to the client providing the relevant information. Afterwards the IKAROS client performs an ''HTTP GET'' request to the I/O nodes and then performs a copy for each HTTP buffer. Finally, the IKAROS client informs the Facebook Graph API of the relevant changes.

* It is crucial to mention that the framework is not constrained to using Facebook.

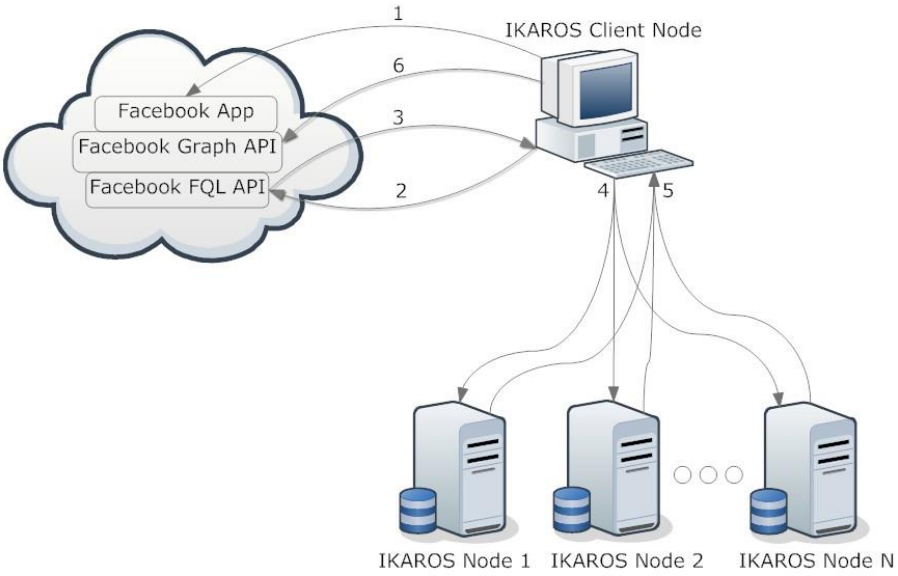

Figure 2-13: The interaction between the IKAROS framework and the Facebook social network (Adapted from Figure 1 of [8] and from Figure 3 of [53]).

Figure 2-14 represents the corresponding sequential diagram of the interaction between the IKAROS system and the Facebook social network service. It illustrates the interactions between the nodes as well as the relevant data flow between the IKAROS client and the I/O nodes.

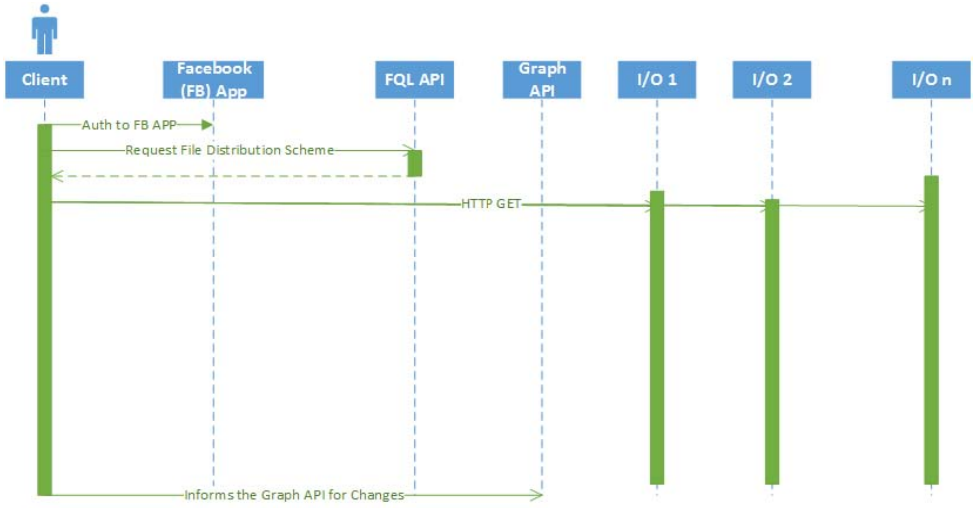

Figure 2-14: The sequence diagram presenting the interaction between the IKAROS system and the Facebook social network.

In our case, eT framework uses Gmail service as an external meta-data service utility.

2.12 The Elastic Transfer (eT) Module

Elastic Transfer (eT) [56] is a publicly available innovating file management platform derived from IKAROS. Today, eT is available for various operating systems, such as Microsoft's Windows, Linux, and Apple's Mac OS. It is a unified platform for achieving efficient data management and sharing. The eT module was developed in order to give users the ability to easily manage, share, and transfer their files between their systems.

The eT module provides flexible data and storage sharing since it:

- Allows file transfers from a workstation to another system, even if the system's firewall is not set to accept incoming traffic and the system does not have a public DNS resolvable name. This is possible because eT automatically performs the appropriate configuration operations to allow users to directly transfer their files between computers.
- Supports third-party data transfers. An eT user can easily access his/her computer through a third-party system and can easily access, share, and transfer his/her files to another end system.
- Bypasses a number of firewalls, allowing users to efficiently share their files or a web application with another system, using reverse HTTP techniques.
- Gives the user the ability to create a personal storage cloud, by exploiting all the benefits that such a cloud architecture provides, but without its bottlenecks. An eT user has the ability to store, manage, share, and transfer his or her files under a trustworthy unified distributed storage space. This virtual storage system infrastructure allows a user to distribute files to different systems according to his or her needs, achieving better performance and higher transfer rates, since files can be downloaded in parallel. Additionally, a given file can be replicated on different systems for redundancy. Finally, eT allows users to store their files via a network that they administrate and trust, without having to pay for cloud storage space.

In this thesis project the capabilities of the eT module will be expanded. A set of experiments will be conducted in order to evaluate and test the updated module.

The eT module has been developed on the Node.js platform. This platform is further described and analyzed in the next section.

2.13 The Node.js Platform

Most web applications have a client and a server side implementation. Single threaded operation and the same utility functions are used in both implementations; however, there are also significant differences between client and server side programming. The client side software is developed in HTML and JavaScript, whereas the server side implementation is based on static programming languages. Server side programming has usually been a complex process where expert knowledge and skills are required. Developers have to use multiple programming languages to bridge the gap between client and server side programming [57].

The implementation of a web server in JavaScript was undreamt of a few years ago. To realize this, Ryan Dahl "conciliated" the two sides by embedding Google's V8 JavaScript engine into an operating system's integration layer that featured asynchronous interfaces to the underlying operating system [57].

This approach gave birth to Node.js [58] which is a cross-platform that allows programmers to easily develop efficient and scalable network applications on both client and server side. The platform is based on Google's V8 virtual machine, which is the same runtime environment that Google Chrome uses for JavaScript. Hence, the platform's modules can be developed in JavaScript or in any other language that compiles to JavaScript. The Google V8 JavaScript engine is used for code execution. The JavaScript is the same as used in client-side applications, but unlike JavaScript in a browser, node.js a development environment must be set up. Node.js is simply JavaScript without a browser. It provides

a high-performance server-side platform ideal for high level web development, maximizing efficiency and throughput. Applications can run directly within the Node.js runtime on different operating systems such as, Microsoft's Windows, Linux, or Apple's Mac OS.

In traditional programming, processing can not continue until an operation finishes, and users have to finish a process before moving to the execution of the next one. This model of programming was adopted from the first time-sharing systems in which every process corresponded to a single user. However, this model does not scale efficiently to the sophisticated requirements of today[59] .

The multi-threading approach was an alternative to the sequential programming mindset. In this model, every process consists of different threads which share the same memory resources and can be executed concurrently. For example, if one thread is waiting for an I/O process to finish, another thread can use the CPU's resources. Moreover, systems with more than one processor can execute more than one thread in parallel[59] .

However, a significant problem arises in this multi-threading programming model. Programmers need to be aware of all threads that are executed at a given time, in order to synchronize their access to memory resources. Although the scheduling of the threads can be managed with the use of locks or semaphores, programmers need to foresee and prevent possible errors that may occur during their concurrent execution. Last but not least, if an application relies heavily on a shared state between threads, random difficult to detect errors may occur[59].

In contrast, Node.js is not a multi-threaded application, thus the platform's users do not have to work with threads or separate processes, but rather Node.js is implemented as a single-threaded server. Node.js adopts the event-driven or asynchronous programming model, where the flow of execution is determined by events which are in turned handled by event handlers of callback functions. These event handlers are called by the system when some specific event occurs. As a result a process does not block when an I/O operation is taking place, hence multiple I/O procedures may occur in parallel, with the relevant callback function being invoked after their end .

Node.js adopts this event-driven, non-blocking I/O model by default, using asynchronous events [59], [60]. Node.js is ideal for web applications that are I/O heavy but computationally simple. Additionally, this approach can be used to easily develop ‘‘data-intensive’’ real-time applications that run across distributed systems. The asynchronous I/O approach means that applications do not wait for an I/O to finish, before continuing with the next line of application code. As a result, node developers do not have to worry about an application blocking any further processing while waiting for a file to finish loading or a database to finish updating [60].

The asynchronous I/O library for file, socket, and HTTP communication that the platform contains, allows the creation of efficient and lightweight applications that can act independently as a web server, without having to install software such as Apache HTTP Server or Microsoft's IIS. It is widely known, that in a traditional web server, such as Apache, every time a request arrives, a new thread has to be created in order to handle the respective request. This approach is efficient when the server's traffic is low; however, errors occur when popular applications adopt this model. One of the reasons for these problems are that a large amount of RAM resources are bounded to processing of each request, leading to performance issues [60].

In contrast, Node.js does not create a new thread for every request but it waits and ‘‘reacts’’ to specific events. Node.js handles its events according to the first in, first out (FIFO) approach, without blocking any requests while waiting for an event to complete [60]. The single-thread model supports tens of thousands of concurrent connections, without using large amounts of RAM, giving the developer increased capability for concurrent applications.

These capabilities have made Node.js an ideal tool for developing websocket servers (such as chat servers), fast file upload clients, command-line tools and scripts, ad servers, real-time data applications, communication links between a server and a browser, or any other application that that can be written in JavaScript.

Node.js includes a large number of modules that are part of its core. NPM is the default package manager for Node.js. NPM manages dependencies for an application and allows developers to use Node.js applications that are available in the NPM registry. Hence, programmers can easily develop complex applications by using pre-existing modules which are released and frequently updated by third parties. NPM provides efficient module management, by downloading packages, resolving dependencies, running tests, and installing command-line utilities [57]. Thus, Node.js code is more readable, portable, and easily modifiable when compared to traditional programming languages.

3 Method

The purpose of this chapter is to provide information regarding the method that was used for further developing the eT framework. Initially, the ‘‘pull’’ and the ‘‘push’’ techniques, and the two predominant server architectures that were presented in Chapter 1 are further analyzed in Sections 3.1 and 3.2 respectively.

Section 3.3 proposes that a combination of the ‘‘push’’ approach and event-driven asynchronous programming will be used to boost the eT′s capabilities.

3.1 ‘‘Pull’’ and ‘‘Push’’ Techniques

A typical IMAP client-server implementation uses the ‘‘pull’’ approach, where clients demand from the server a specific type of information (i.e., information about the arrival of a new e-mail or the message itself). Thus, clients initiate a connection with the server then make requests, while the server replies to requests. Requests could be made automatically according to a predefined time interval. Nevertheless, in our case the main problem that arises with this approach is that frequent polling leads to an inefficient use of the network since the server has to provide its services to a large number of requests in a short period of time. A simple solution to this problem would be to poll the server after longer time intervals. However, with this approach the arrival of a new e-mail notification would not be ‘‘immediate’’, but will depend on the polling frequency. In our case, it is obvious that a different approach should be made and a new implementation of a client-server IMAP connection is needed. In this approach fewer server resources should be consumed when multiple clients simultaneously are connected to the same mailbox or when the same user sends successive requests to the server, or when in general the MDS has to server a large number of requests in a short period of time.

The ‘‘push’’ method could overcome these limitations, by implementing the reverse approach to the pull method presented above. In this approach the server initiates a connection to the clients when a specific change occurs at the server’s side (i.e., the occurrence of an event). Hence instead of initiating consecutive connections from clients to server, which may ceaselessly request information about the arrival of a new e-mail message, and might possibly cause the IMAP server to time out; the server itself informs the clients of the arrival of a new e-mail message.

Nevertheless, it is important to mention that both technologies have their own advantages and trade-offs, always depending on the specific setting where they are applied. Bozdag, Mesbah, and van Deursen did an experimental comparison in 2007 [61] between ‘‘pull’’ and ‘‘push’’ techniques in the case of asynchronous JavaScript and XML (AJAX). The experiment showed that the ‘‘push’’ approach should be implemented if high data coherence and high network performance is required. However, this approach lead to low scalability and the CPU usage was seven times higher than the CPU usage when the ‘‘pull’’ approach was implemented. Additionally, the server started to saturate when 350-400 users were simultaneously connected to the system, and load balancing and server clustering techniques were required to ensure the system’s stability[61]. In contrast, when the ‘‘pull’’ technique was implemented total data coherence with high network performance was not achieved. When the ‘‘pull’’ time interval was higher than the publish interval, some data loss occurred, and when the ‘‘pull’’ time interval was lower than the publish interval, the network performance decreased. This method would perform with ideal efficiently only if the ‘‘pull’’ interval is equal to the publish interval. Unfortunately, if the publish interval is not static or predictable, then ‘‘pull’’ can not be efficiently applied in applications where data are published ‘‘randomly’’[61].

Different hybrid models that combine both techniques have been developed [2], [3] to achieve high efficiency. Nevertheless, the choice of the appropriate technique should be based on the type of application that is to be developed.

The current version of the eT has been implemented through the ‘‘pull’’ approach as a proof of concept. However, in a real world situation if a large number of concurrent clients connect to the

IMAP server, or in general if a large number of requests has to be served, the server's resources are limited, a fact that leads to inefficient service provision. It becomes obvious that the system's design has to be changed to overcoming these problems. In general, the ‘‘push’’ approach increases the potential that users receive *irrelevant* data while not necessarily receiving the information they need [62]. In our case each client needs to receive all of the meta-data and information that is communicated to all of the other clients. All nodes should be aware of all of the meta-data information. For example, let us assume that a file has been transferred from a source host to three different hosts in three data chunks and a new client connects to the system. If this client is unaware of this specific data transfer between the original host and the three destination hosts and this client wants to receive the same file, this new client has to request it from the initial source host and cannot request to receive it in parallel from the three other hosts. So, if the initial source host is not active at a specific moment and the new client is not aware of the latest status of the meta-data it will not be able to ask and receive the file, even though this file exists in the three other hosts. The ‘‘push’’ technique is a good choice for data delivery in distributed information systems [62]. Hence for the purpose of this thesis this method has been adopted as it increases the framework's scalability in terms of handling a larger number of concurrent requests, thus avoiding overloading the system.

3.2 Typical Server Architectures

As mentioned in Chapter 1, two predominant server architectures exist: threads and events. Of course more sophisticated architectures combining both threads and events have been developed, such as the Staged Event-driven Architecture (SEDA) [63] etc.

Since the dispute as to whether threads or events are better for achieving high web server performance still exists, this chapter describes these two predominant server architectures alongside with their major scalability issues.

3.2.1 Thread/Process-Based Server

In a typical client-server connection the following steps have to be followed:

1. The client opens a connection and sends a request to the server.
2. The server sends a response back to the client.
3. The connection is closed.

In this architecture, the synchronous blocking I/O model is adopted for dealing with I/O procedures. This model of computation is supported by many programming languages and all incoming requests are handled sequentially by a thread or process. However, this model limits concurrency since a separate thread or process has to be dedicated to each request. High concurrency can only be achieved if multiple threads/processes run in parallel, serving multiple tasks at the same time. It becomes obvious that multi-threaded and multi-process server models are based on the same principle, where each new connection is served by a separate activity.

In general, when multiple threads or processes have to be executed, a sequential execution model is adopted. The operating system implements a pre-emptively scheduling model where tasks are sorted according to priority. The highest priority processes are executed first. This architecture is ideal when a small number of CPU-bound operations have to be executed or when multiple CPU cores exist, as the later are able to serve multiple threads or processes at the same time.

However, when a large number of requests arrive at the same time, a large amount of memory is needed since each request should be served independently through a separate, dedicated thread/process. Of course a system could limit the number of threads/processes and simply enqueue the other requests; however, if there a large number of requests have previously arrived, then some of the enqueued requests will have to wait for a long time to be served and they will probably time out.

Additionally, when a higher priority task arrives, the CPU has to interrupt the process that is currently running and switch to another processing context. This model leads to low performance and ties up resources when multiple requests have to be served [1].

3.2.2 Event-Driven Server

The event-driven architecture was developed as an alternative to the thread/process-based design. In this model the asynchronous non-blocking approach is adopted, thus the server uses a single thread to handle all concurrent connections and requests. This single thread processes all the arriving events requesting I/O operations.

This architecture uses ‘‘event emitters’’ which sort the relevant events in a queue, thus these events are multiplexed into a single flow. Events are processed according to their order in the queue by a so called ‘‘event loop’’. This loop waits for new events to arrive and then processes the first event in the queue. An event may trigger more events which are in turn are added to the ‘‘event queue’’. Even if multiple events arrive (nearly) simultaneously, they are added to the ‘‘event queue’’ and then processed one by one.

An event is either related to an ‘‘event handler’’, which executes a specific procedure, or to a callback function. Thus, if a specific procedure is related to an event, it registers a callback function that will be invoked after the occurrence this event, at which time a specific piece of code is run. A callback function is executed indivisibly until it hits a blocking operation, at which point it registers a new callback and then returns.

This rationale makes asynchronous programming quite different from traditional programming, where all procedures are executed sequentially except when they are interrupted under a pre-emptive scheduling model. A conceptual depiction of this model is presented in Figure 3-1.

Events better manage I/O concurrency in the server software and can help avoid bugs caused by unnecessary CPU concurrency introduced by threads. The adoption of an event-driven architecture has different scalability that thread/process-based architectures. The fact that just a single thread is associated with all requests leads to less memory being required - since all connections are served by a single thread. At the same time CPU performance increases since there is no overhead for context switching between multiple threads [1].

In many applications, the non-blocking implementation leads to a more stable performance under heavy load than threaded programs. Additionally, a non-blocking implementation is more convenient that the traditional blocking programming where all code is executed sequentially when one processor is used, a process can not be executed if another process is running. Thus, events can provide all the benefits of threads, with substantially less complexity, resulting to robust software implementations [64].

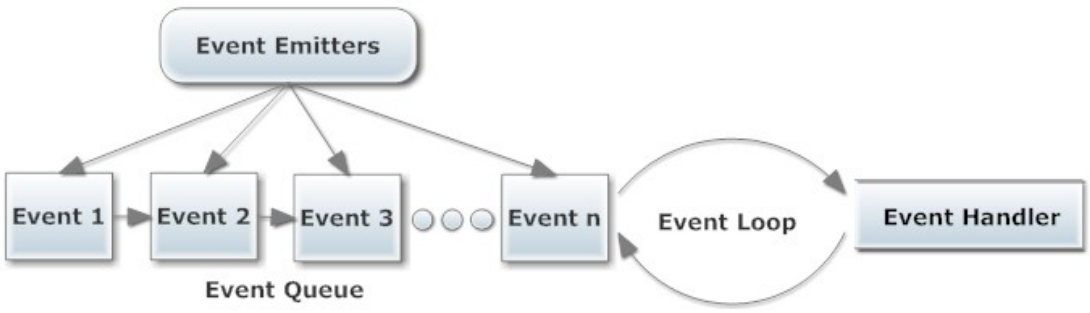

Figure 3-1: A conceptual depiction of the event-driven architecture model

3.3 Selected Method

In our case the benefits of the ‘‘push’’ technique and the event-driven approach will be used in order to address the low scalability problems that occur when a large number of users are connected to the system a the same time. Additionally, this approach will lead to lower resource use when a large number of requests arrive to the system at the same time. The IKAROS Framework’s Elastic Transfer (eT) module will be modified by adopting a solution that combines the ‘‘push’’ approach and event-driven logic. Specifically, the module will be updated by replacing the existing IMAP client-server architecture with an asynchronous, event-driven one.

The existing implementation utilizes the ‘‘Inbox’’ [65] module which creates the IMAP connection to the Gmail servers. This module follows the traditional approach where all clients continually ask the server about the arrival of a new e-mail. However, this leads to poor behavior of the system when there is a large number of incoming requests. Therefore we will replace this traditional client-server IMAP (implemented by the current ‘‘Inbox’’ module) with a new asynchronous ‘‘Imap’’ module. This new module will be adjusted in order to be integrated into the eT framework.

Regarding the meta-data scheme of the eT, the current meta-data format makes the module incompatible with other systems. It is obvious that an efficient way to transfer the meta-data information between separate programs or distributed systems is needed. Therefore, eT′s meta-data scheme should be modified to be a language independent format, but at the same time should use a format that is familiar to software developers.

JSON [7] is a lightweight data-interchange format that stores information in a well-organized and logical manner. JSON can be easily read and written by humans, and can be efficiently generated and parsed by computers. JSON was originally derived from the JavaScript programming language and it was initially used as an alternative to XML for transmitting data between web applications and servers. Nowadays, JSON defines a language independent format, with parsing and generating functions available in many programming languages. This has made JSON an ideal open standard for transmitting data objects consisting of attribute–value pairs between JavaScript-interpreting web browsers and JSON-aware web applications[7].

Consequently, to achieve efficient information exchange between different systems a change in the eT′s meta-data scheme is needed. The meta-data format will be changed to be easily parsed as JSON objects, increasing the framework’s user friendliness and interoperability.

4 Updating the eT Framework

This chapter presents the changes that were made in eT′s source code for implementing the asynchronous event-driven logic. Additionally, the new meta-data scheme of the system is described. The framework's meta-data information can now be parsed as JSON object and streamed in text files on the hard disk of each client. The difference between the meta-data notation between the older and the updated version of the eT is shown through a simple data transfer scenario. For better understanding this part of the report and for familiarizing with the eT framework it is recommended that the reader read Appendix A – as this gives a basic usage scenario for the framework. The latest version of the eT′s source code is presented in Appendix B.

4.1 Design Model of the Asynchronous, Non-Blocking IMAP Client-Server Implementation

Since Google's Gmail is eT′s basic meta-data management utility, eT utilizes Google's API for establishing a connection with Google's Gmail servers. The framework deals with the information that is required for the authentication process including the client's ID, secret, token, and a random string that is created and used during the authentication process. Afterwards, the eT creates the ''imap'' variable which contains an instance of the ''Imap'' variable which is the IMAP module that implements the event-driven connection utilizing the IMAP protocol. This specific instance contains the token that is received after the authentication process has completed successfully, the TCP port number of the connection, etc. This information is required to connect to one of Google's e-mail service servers.

Figure 4-1 shows how the asynchronous event-driven logic of the IMAP client-server is implemented and how the e-mails that contain the meta-data information are managed . The new architecture of the system is presented and the different steps that are followed by the clients are depicted.

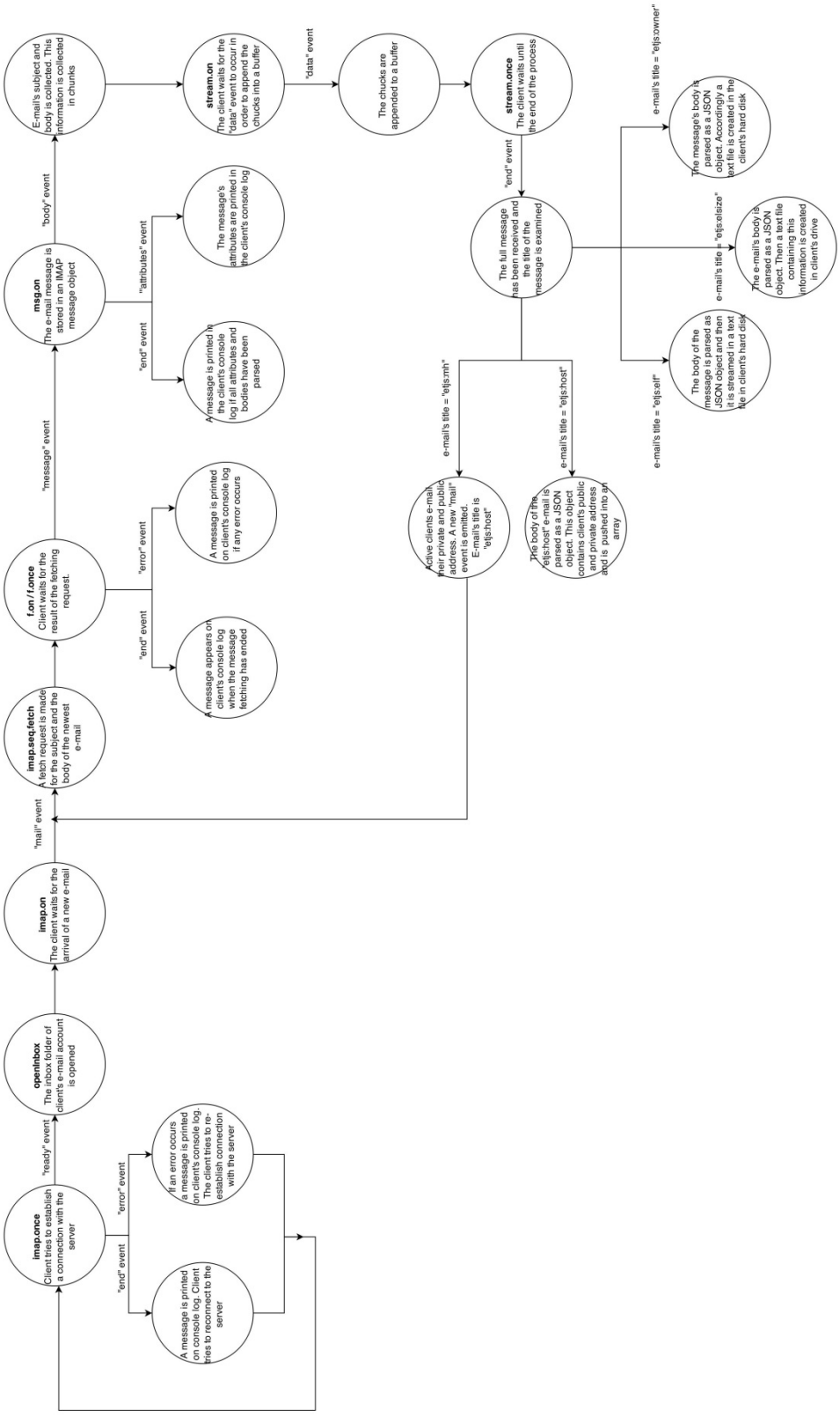

Figure 4-1: The asynchronous event-driven logic of the IMAP client-server and the meta-data management

As it is shown in Figure 4-1 the client initially tries to establish a connection with the IMAP server. Depending on the connection's status different events may be emitted. The ''ready'' event is emitted if the connection is established without errors *and* the authentication process was successful, the ''error'' event is emitted if any error occurs, and the ''end'' event is emitted when the connection has ended. In the last two cases the relevant message is printed on client's console log and the client tries to reconnect to the IMAP server.

Assuming that the connection is successfully established, the client opens the inbox folder of his/her account and then waits for the arrival of a new e-mail. If a new e-mail message arrives the "mail" event is emitted and a fetch request is made for the subject and the body of the newest message.

Next the client waits for the result of the fetch and depending on this result different events may be emitted. If there is an error during the fetch request, then an "error" event is emitted while the "end" event is emitted when the message fetching completes. Similarly, the "message" event is emitted when the fetching result is a message.

Assuming that the fetch request result is a message, then the newest e-mail is stored in an IMAP message object which in turn may react differently according to the events that may be emitted. More specifically, if all the message attributes have been collected, then the "attributes" event is created and the attributes are printed on client's console log. The "end" event is emitted if all attributes and bodies have been parsed. Lastly the ''body'' event is emitted for the requested message body and the client collects the e-mail's subject and body in chunks.

Assuming that the "body" event is emitted, then the client then waits for the "data" event to be emitted in order to store the chunks into a buffer. When the ''data'' event arrives these data chunks are appended to a buffer and once the full message is received the title of the message is examined. Depending upon the e-mail subject different processes are executed. Note that all of the different mail subjects (relevant to this thesis project) are shown in Table 4-1 along with a description of their purpose.

Subject	Purpose
etjs:mh	This specific e-mail is sent when a client requests all active clients.
etjs:host	In turn, each active client sends this e-mail which contains his private and public address.
etjs:elf	When a data transfer is made this e-mail is sent containing the name of the transferred file, the number of data chunks that the original file was broken into and the public address(es) of the destination clients(s)
etjs:elsize	Additionally, this e-mail includes the name of the transferred file as well as its size in bytes.
etjs:owner	This e-mail contains information the name of the transferred file and its owner.

Table 4-1: List of e-mail subjects used by eT

To understand when each specific e-mail is sent and how the new design handles the messages according to their subject, a simple use of the framework is presented. Assume that a number of clients are connected to the system, when the IP addresses of these active clients are requested an e-mail is automatically sent to the Gmail account. This e-mail has the title: "etjs:mh". The arrival of this new e-mail emits the event "mail" that was presented before and all active eT clients follow the same procedure that was described above in order to receive the message. Once the e-mail is received its title is examined.

In our case, since the title of the newest message is “etjs:mh”, thus the condition e-mail's title = ‘‘etjs:mh’’ is met and a specific line of code is executed. Specifically, all clients e-mail their private IP and fully qualified domain name (FQDN) public address. This e-mail has the title “etjs:host” and as before the arrival of these new e-mails causes the emission of ‘‘mail’’ events, and the procedure depicted in Figure 4-1 is followed. Once the message is received successfully its title is examined, the depending upon its title specific procedures will be followed. In this case the topic of the e-mail is “etjs:host” so the title = ‘‘etjs:host’’ condition is met. Thus, the body of of the e-mail, which contains the client’s IP address is parsed as a JSON object and then stored in an array.

Accordingly, when a data transfer is performed three new e-mails are automatically sent to the Gmail account. As before, the arrival of these new mails causes the emission of three new ‘‘mail’’ events. These new e-mails have the title ‘‘etjs:elf’’, ‘‘etjs:elsize’’, and ‘‘etjs:owner’’ respectively and include the meta-data related to this specific data transfer. If the clients successfully receive these e-mails, then the e-mails’ title is examined. Initially for the e-mail with "etjs:elf" title the e-mail's title = ‘‘etjs:elf’’ condition is met and all of the clients parse the body of the e-mail message as a JSON object. This same information is streamed to a text file on the client’s hard disk in the “~/eTshare/elfs/mdata/” directory. The same procedure is followed for the next e-mail which has the title "etjs:elsize". The condition title = ‘‘ etjs:elsize’’ is met and the e-mail’s body is parsed as a JSON object. Then the size is written to a text file in the “~/eTshare/elfs/mdata/” path of all clients. Lastly for the e-mail with title "etjs:owner" the condition title = ‘‘ etjs:owner ’’ is met and the e-mail’s body which contains information relevant to the file owner is parsed as JSON object. A text file with these information is created in the ‘‘~/eTshare/elfs/mdata/’’ directory of all clients.

4.2 The new meta-data scheme of the eT framework

The changes in the meta-data scheme will become easily comprehensible if we compare two similar data transfer scenarios. The first data transfer was made using the old version of the eT module, while the second transfer uses the updated version of the eT framework.

In both data transfers the source host, the destination host, and the transferred file remained the same. The source has the IP address http://10.0.1.171:61949, https://host1.localtunnel.me, while the destination host has the IP address http://10.0.1.173:61949, https://host2.localtunnel.me. The filed transferred is ‘‘Example.pdf’’.

Table 4-2 indicates the meta-data notation in the previous version of the eT while the Table 4-3 presents the new meta-data the scheme. It is obvious that the updated meta-data scheme is more detailed compared to the previous version of the eT. The new meta-data scheme is easily human understandable and this specific format makes the meta-data easily parsed as JSON objects.

Table 4-2: The eT's meta-data scheme before the changes

Subject	Body
etjs:mh	etjsbody:mh
etjs:host	etjsbody:http://10.0.1.171:61949,https://host1.localtunnel.me,
etjs:host	etjsbody:http://10.0.1.173:61949,https://host2.localtunnel.me,
etjs:elf	etjsbody:Example.pdf-:1-:1-:https://host2.localtunnel.me-:
etjs:elsize	etjsbody:Example.pdf-:197822-:
etjs:owner	etjsbody:Example.pdf-:owner-:gkikasn@gmail.com-:

Subject	Body
etjs:mh	{"etjsbody":"mh"}

Table 4-3: The eT's updated meta-data scheme in JSON notation

etjs:host	{"etjsbody":{"privateaddress":"http://10.0.1.171:61949","publicaddress":"https://host1.localtunnel.me"}}
etjs:host	{"etjsbody":{"privateaddress":"http://10.0.1.173:61949","publicaddress":"https://host2.localtunnel.me"}}
etjs:elf	{"etjsbody":{"outfile":"Example.pdf","sn":1,"dl":1,"destination":["https://host2.localtunnel.me"]}}
etjs:elsize	{"etjsbody":{"filename":"Example.pdf","filesize":197822}}
etjs:owner	{"etjsbody":{"filename":"Example.pdf","fileowner":"gkikasn@gmail.com"}}

5 Analysis

This chapter presents the experimental procedure that was followed to evaluate the new version of the IKAROS module. More specifically, a number of data transfer experiments were made to measure the performance of the system, in order to ensure that the changes did not affect the stability of IKAROS. The chapter is divided in two section with the first one describing the testbed used for the experiment and the second one presenting the results of the experiment.

5.1 Experimental Procedure

A set of data transfers were performed to test the behavior of the updated version of the IKAROS system. Initially the data transfer performance of the two versions of the IKAROS framework was measured, to confirm that the changes did not affect the smooth operation of the module. Additionally, the same experiments were conducted using GridFTP (a widely used protocol for data transferring in WAN environments). A comparison between the original version of the IKAROS framework and GridFTP was made earlier [22]; however, in this thesis project additional data regarding the the behavior of the two systems were gathered and a comparison between the updated version of the IKAROS framework is made with the GridFTP protocol.

The experimental process was divided in two phases and all data transfers were performed in a WAN environment, starting from the ZEUS Cluster located at the Institute of Nuclear and Particle Physics of NCSR ‘‘Demokritos’’ ending at specific sites of the EGI.

The Zeus Cluster consists of:

- Ten 2100 MHz AMD Opteron CPU based systems, each with 8 CPU cores and 16 GB of RAM,
- Four 800 MHz CPU based SOHO-NAS devices, each with 256 MB of RAM, 1000 Mbps Ethernet controller, and 3 TB of storage capacity, and
- Ten 200 MHz CPU based SOHO-NAS devices, each with 32 MB of RAM, 100 Mbps Ethernet controller, and 2 TB of storage capacity.

Additionally, a 1 Gbps full duplex link interconnects the nodes and a 2.5 Gbps WAN provided connectivity between the sites.

The cluster itself provides services to the LHC CERN CMS experiment and to the KM3Net consortium. ZEUS belongs to the NCSR ‘‘Demokritos’’ Mobile Grid Infrastructure, with the IKAROS framework being a building block of the architecture. Nevertheless, to achieve as precise results as possible, during the experiment the processors of the system and the network infrastructure were isolated from any other activity.

During the first phase of the experiment data files of the same size (100 MB, 1024 MB, and 5120 MB) were transmitted via one, four, and eight parallel channels from the ZEUS cluster to the following HellasGrid sites:

- To HG-01-GRNET - located at NCSR ‘‘Demokritos’’ campus in Athens, Greece but it is connected to the ZEUS cluster through a WAN,
- To HG-06-EKT - based in Athens, Greece, and
- To HG-03-AUTH - located in Thessaloniki, Greece.

All file transfers to each site were repeated 10 times to achieve as accurate results as possible, these results were averaged. The experiment could have been performed only one time, however to reach a high confidence level and because the network is a shared with other (rather than specifically dedicated to this specific experiment), the procedure was repeated 10 times to ensure that the data from a given run of the experiment was consistent with the data from the other runs (the variance

between the different runs' results were small). All measured results between the ZEUS cluster and the three sites were averaged into a single value indicating the average performance between the ZEUS cluster and the Hellas Grid.

During the second phase of the experiment the data transfers were performed between more geographically dispersed areas. A file of 1 GB was transferred 10 times from the ZEUS clusters to sites located in Italy, Turkey, Hungary, UK, and China. The transfers were made through four parallel channels. As before, the same data transfers from the ZEUS cluster to the corresponding sites were performed 10 times and the results averaged into a single value for each site.

The grids that took part in the procedure were the following:

- INFN-PADOVA in Italy,
- ULAKBIM in Turkey,
- NIIF-HU in Hungary,
- UKI-LIV in the United Kingdom (UK),
- UKI-ECDF in the UK,
- BEIJING-LCG2 in China.

And a conceptual image of the complete test bed is presented in Figure 5-1.

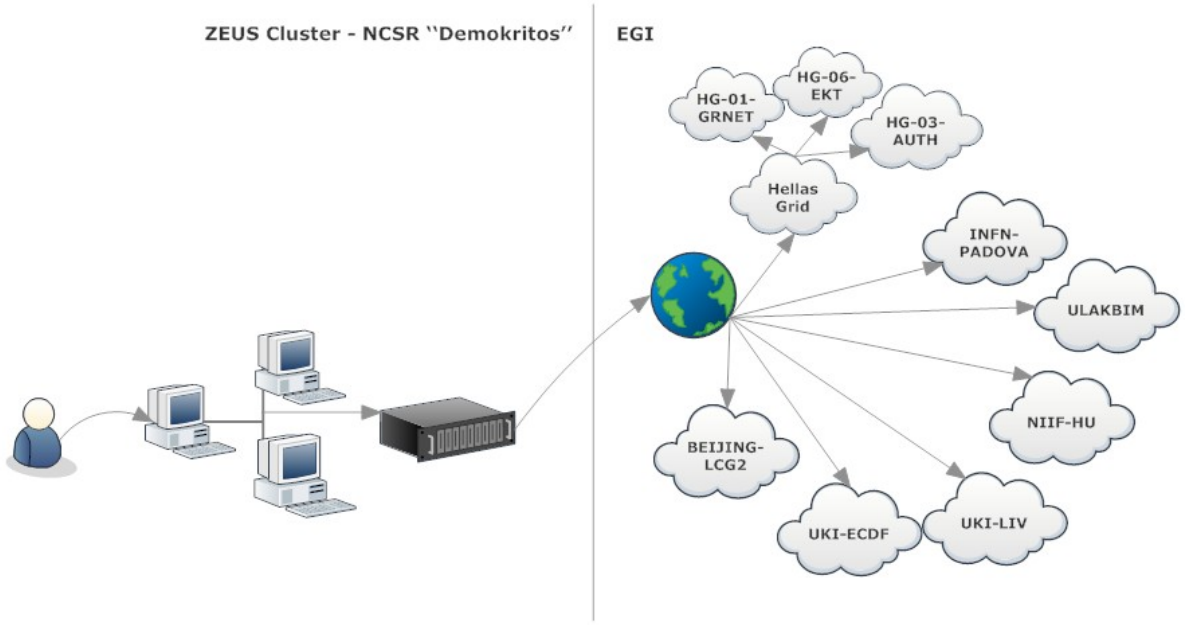

Figure 5-1: The testbed of the experimental procedure

5.3 Results

The following sections present the data transfer experimental results together with figures summarizing these results.

5.4 First phase of experiment

In this section the results from the first phase of the experiment are report. Keep in mind that all of these sites are within Greece. The average round trip time in ms between the ZEUS cluster and the three sites are following ones:

- 1 ms to HG-01-GRNET,
- 1.72 ms to HG-06-EKT, and
- 8 ms to HG-03-AUTH.

The monitoring service from https://mon.grnet.gr/ was used to find the above RTTs.

Table 5-1: Data Transfer from Zeus Cluster via a single channel (all rates in MB/Sec)

First Phase of the Experiment			
File System	Average to HellasGrid		
File Size (MB)	100	1024	5120
IKAROS	14.3	20.4	17.0
IKAROS (updated)	14.3	20.4	17.3
GridFTP	8.1	15.6	17.7

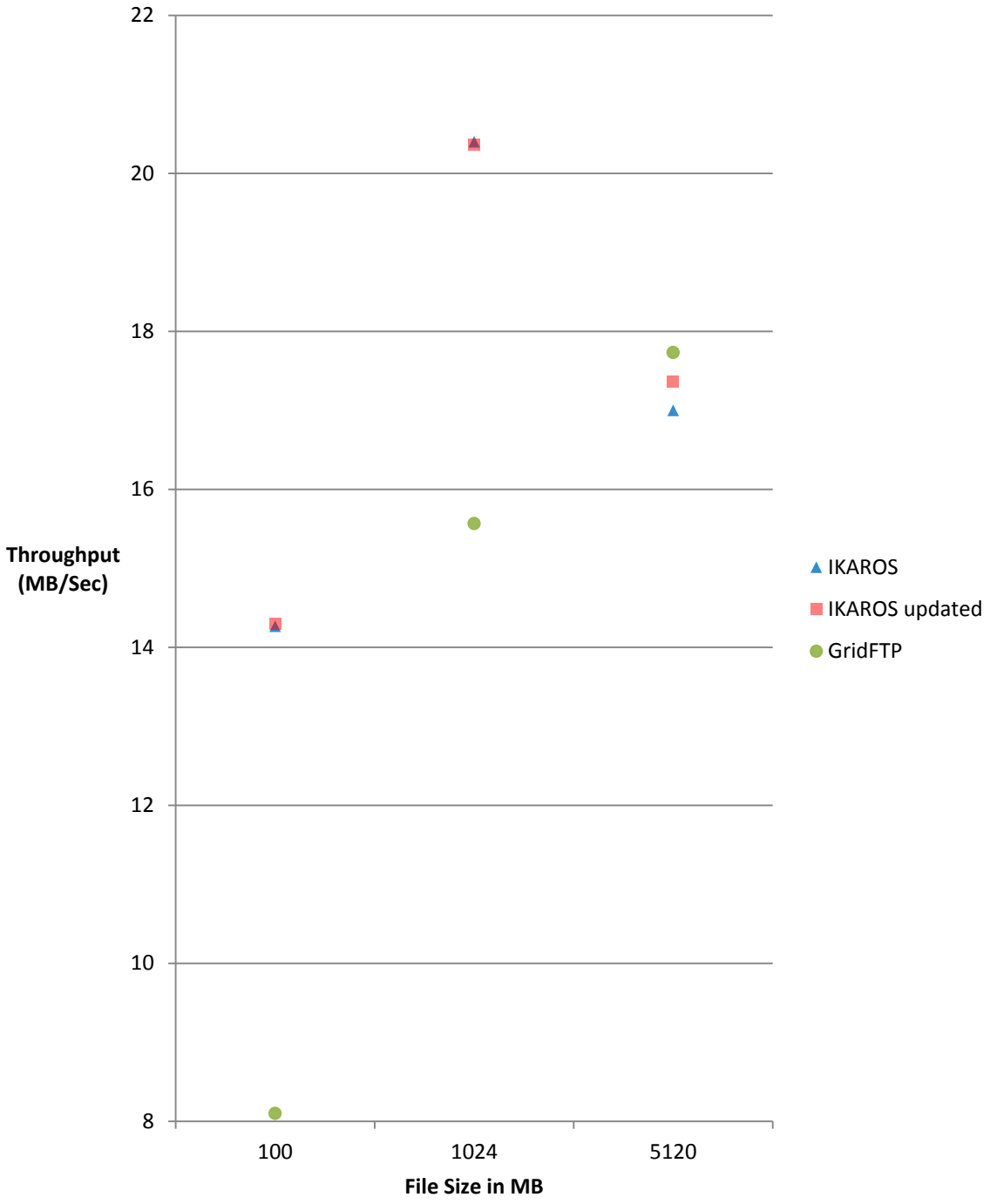

Figure 5-2: **Data transfer results through 1 parallel channel using the two versions of IKAROS and GridFTP**

First Phase of the Experiment

Table 5-2: Data Transfer from Zeus Cluster with 4 channels (all transfer rates in MB/sec)

File System	Average to HellasGrid		
File Size (MB)	100	1024	5120
IKAROS	23.9	41.6	33.5
IKAROS (updated)	23.5	41.9	33.4
GridFTP	21.3	39.1	32.2

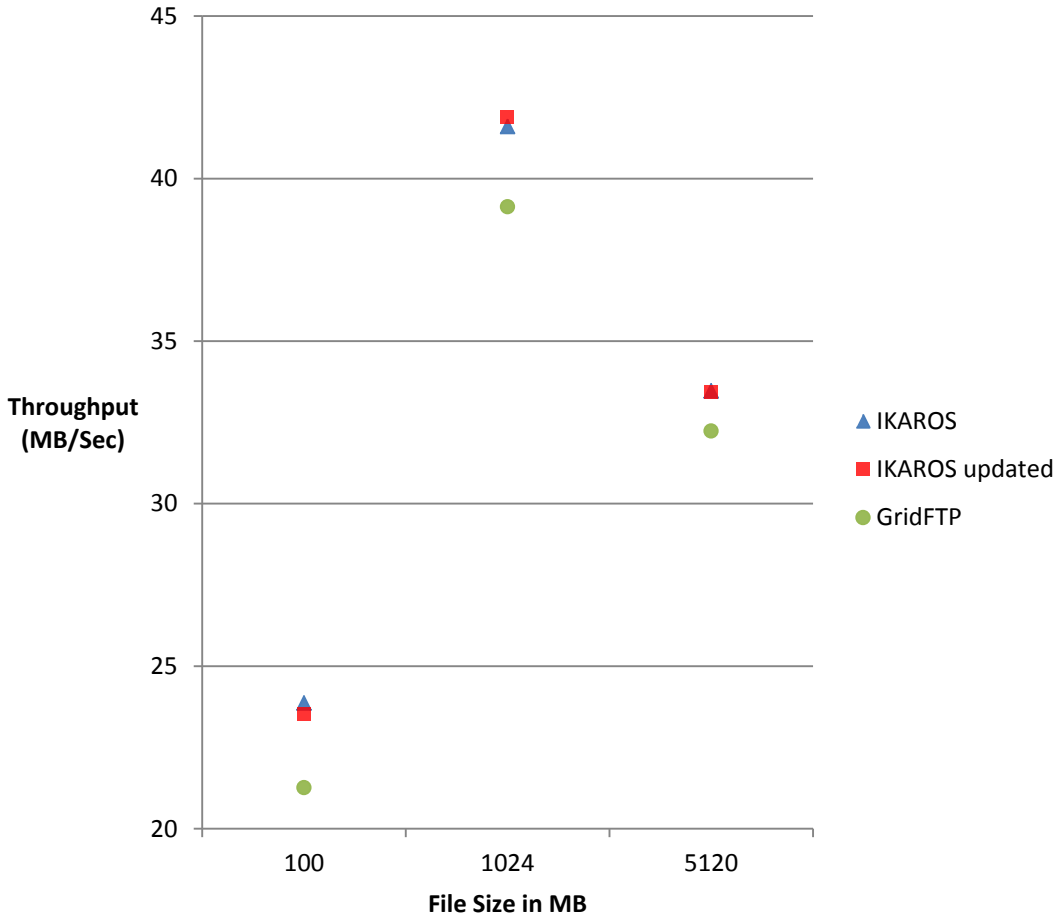

Figure 5-3: Data transfer results through 4 parallel channels using the two versions of IKAROS and GridFTP

Table 5-3: Data Transfer from Zeus Cluster with 8 channels (all transfer rates in MB/sec)

First Phase of the Experiment			
File System	Average to HellasGrid		
File Size (MB)	100	1024	5120
IKAROS	22.7	48.3	40.3
IKAROS (updated)	23	48.1	40.2
GridFTP	18.1	42.7	41.9

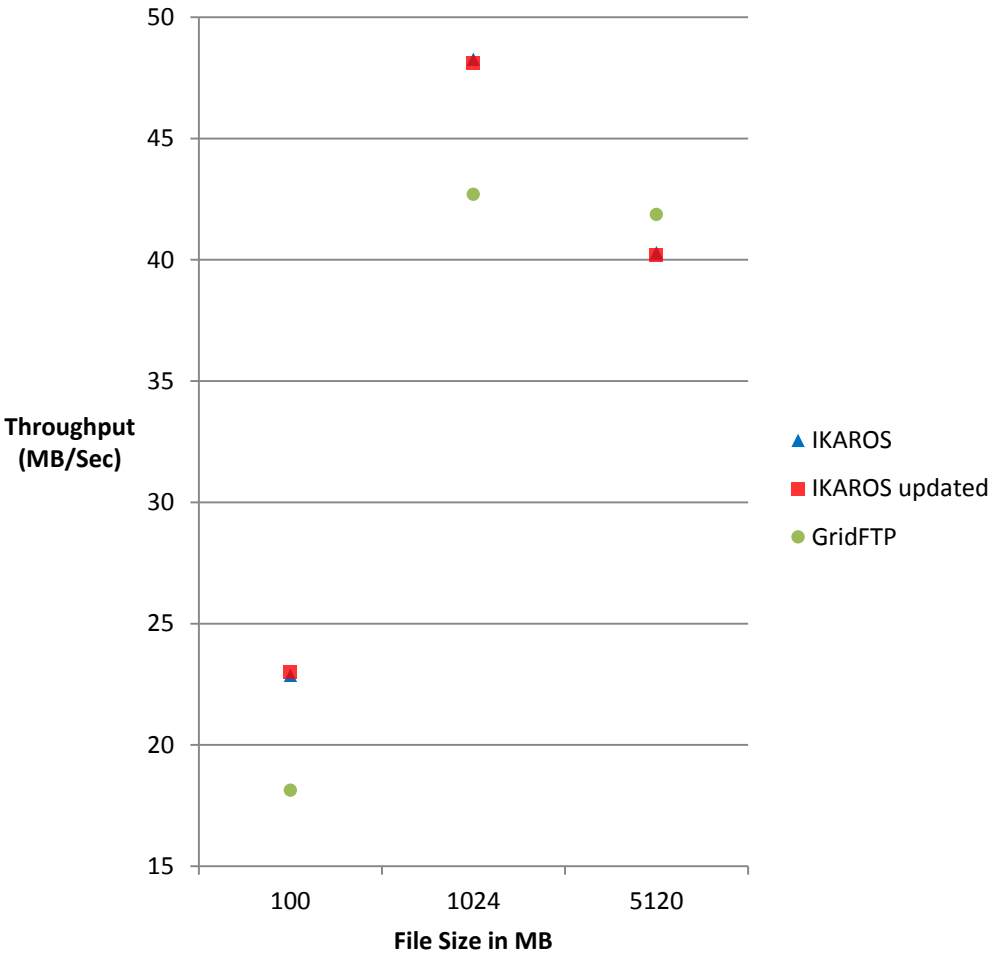

Figure 5-4: Data transfer results through 8 parallel channels using the two versions of IKAROS and GridFTP

5.5 Second phase of experiment

In this section the results from the second phase of the experiment are reported. In this phase the sites were in a number of different countries. The average round trip time from Zeus to each of these sites is shown in Table 5-4.

Table 5-4: Average round trip time in ms between Zeus and the indicated site

Second Phase of the Experiment	
Site	Round trip time (ms)
INFN-PADOVA in Italy	
ULAKBIM in Turkey	
NIIF-HU in Hungary	
UKI-LIV in the United Kingdom (UK)	
UKI-ECDF in the UK	
BEIJING-LCG2 in China	

Table 5-5: Data Transfer of 1024 MB from Zeus Cluster with 4 channels

Second Phase of the Experiment						
File System	to INFN-PADOVA in Italy	to ULAKBIM in Turkey	to NIIF-HU in Hungary	to UKI-LIV in the United Kingdom	to UKI-ECDF in the United Kingdom	to BEIJING-LCG2 in China
IKAROS	15.2	34.3	40.1	11.6	17.7	6.1
IKAROS (updated)	15.1	34.4	39.9	11.8	17.7	6.3
GridFTP	15.7	29.1	36.3	10.1	15.9	3.1

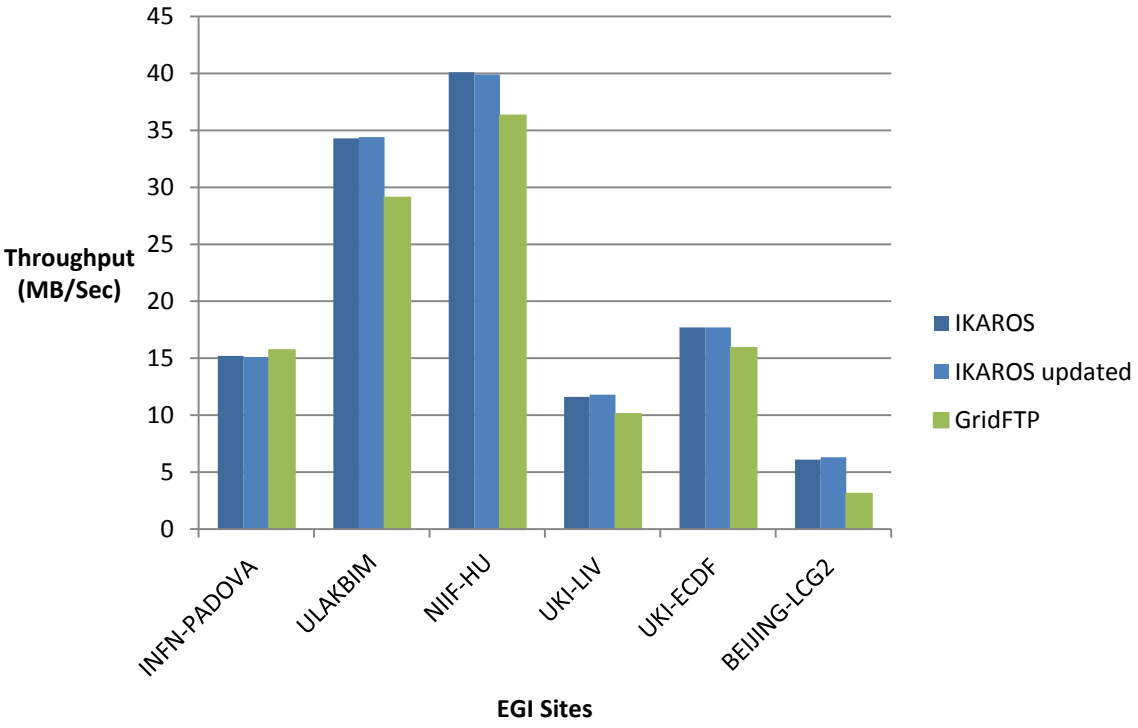

Figure 5-5: **Data transfer results of 1GB file through 4 parallel channels using the two versions of IKAROS and GridFTP**

5.6 Discussion of the experimental results and analysis

After observing the data transfer rates of all of the experiments, it is evident that the changes made to the original version of IKAROS system did not affect its data performance, i.e., there was neither a positive nor negative change in transfer rate. The new version of the system showed the same performance as its predecessor, and operated as smoothly as before and without the occurrence of any errors. It is crucial to mention that the purpose of the thesis project was not to increase the performance of the system, but rather to ensure that its stable operation is maintained. The experimental results were as expected, because the changes were only related to the IMAP client-server implementation and not to the protocols that play an active role in the data transfer performance (such as TCP) as these were the same in both versions of the IKAROS platform.

However, after comparing the two versions of IKAROS system with the GridFTP protocol the following become apparent:

- During the first phase of the experiment, in the data transfer through one parallel channel, both versions of the IKAROS system had better performance than GridFTP when files of 100 MB and 1024 MB were transmitted. IKAROS and GridFTP had almost the same performance, with GridFTP being a slightly better when a 5120 MB file was transferred.
- During the first phase of the experiment, in the data transfer through four parallel channels, IKAROS achieved a slight better performance in all data transfers when compared to GridFTP.
- During the first phase of the experiment, in the data transfer through eight parallel channels, IKAROS had a better performance when compared to GridFTP in 100MB and 1024 MB file transfers, however GridsFTP's performance reached and slightly exceeded the performance of IKAROS when the 5120 MB file was transmitted.

- During the second phase of the experiment IKAROS exceeded GridFTP's performance in the five out of the six cases where a 1 GB file was transmitted over four parallel channels to geographically dispersed areas.

In summary, the experimental results proved that the modification to the framework's was successful, since its performance was not affected, while at the same time the operation of the system was as smooth and stable as before.

6 Conclusions and Future work

This chapter gives the conclusions that were reached after analyzing the results of the experimental procedure. Additionally some future work is suggested to further enhance the IKAROS framework's capabilities.

6.1 Conclusions

The new architecture did not change the fundamental behavior or performance of the IKAROS framework. However, the new architecture offers a series of advantages which are not so apparent, but nevertheless they should be mentioned:

- The ‘‘Imap’’ module is more widespread than the ‘‘Inbox’’ module. The fact that a large community uses and maintains this specific module increases its reliability. ‘‘Imap’’ is frequently updated which in turn increases the value of the IKAROS framework.
- During the installation process of the ‘‘Imap’’ module, several libraries that are necessary for eT to run are automatically installed. In contrast, these libraries were not automatically installed during the installation process of the ‘‘Inbox’’ module, but had to be manually installed by the system's users. By changing the architecture to use the “Imap” module, the installation process of IKAROS has been facilitated, thus increasing the module's flexibility and potentially increasing the size and diversity of the pool of users.
- The ‘‘Imap’’ module has a more complete API than the ‘‘Inbox’’ one. Of course only some of its commands have been used by eT; however, a more complete API could offer additional features and capabilities for future improvements.

Additionally, as mentioned earlier:

- The new asynchronous event-driven IMAP client-server implementation is more suitable to be used in real world cases when compared to the previous IMAP client-server implementation which was used as in the prototype version of the eT.
- The modifications in the meta-data scheme of the system to the JSON standard format which is a very common Web 2.0 technology offers greater interoperability and compatibility with other systems.

6.2 Future work

In addition to the modifications that were made for the purpose of this project, some future work is needed to enhance the module's capabilities. This future work will not focus on improving the system's data transfer performance, but will provide applications and users the ability to use the system in the best possible way, i.e., on demand, adapted to the specific infrastructure that they are equipped with. Two changes are proposed, with the second requiring greater expertise and in depth technical knowledge.

In its current version of eT, if a new host connects to the framework, it has no meta-data concerning the file transfer processes that have previously been done. This means that whenever a new client connect it is unaware of the meta-data files that have been created until now, hence a potential update could be made to automatically fetch & store to the new client's hard disk drive all the meta-data that has been generated prior to this moment.

A second future change that would increase the module's value is to create a POSIX module that will mount and unify the distributed storage space of all hosts into one virtual disk. Creating a virtual file system will enable users to access this distributed storage space through their specific operating system *without* having to explicitly run the framework in their specific environment.

6.3 Required reflections

IKAROS is a framework that based on tools and utilities that can fulfill the demands of the next generation international collaborative experiments. The eT module was directly derived from the IKAROS framework and shares the same philosophy with IKAROS when it comes to organizing, managing, and transferring data. The goal of this thesis project was to bring the eT IMAP client-server implementation from a prototype version to a more functional form, and to create a more visible meta-data scheme to enable eT able to communicate with external systems. Obviously the work that has been done in this thesis project is just a small step toward making eT a bit more efficient and functional. Additional work is needed to further enhance the module's capabilities. However, making the eT framework fully operational would offer a number of advantages from social, economic, and ethical aspects. The framework offers users the ability to efficiently organize access, manage, and transfer their files by unifying their distributed storage space into a single virtual disk. By using their own infrastructures and without needing to pay for cloud storage space they could build their own personalized cloud that suits their needs. This approach increases data privacy since users' information will be stored within their own trustworthy systems.

References

[1] B. Erb, “Concurrent Programming for Scalable Web Architectures, Institute of Distributed Systems,” Institute of Distributed Systems, Ulm University, 2012 [Online]. Available: http://www.benjamin-erb.de/thesis. [Accessed: 05-Mar-2015]

[2] J. P. Martin-Flatin, “Push vs. pull in Web-based network management,” 1999, pp. 3–18 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=770671. [Accessed: 17-Apr-2015]

[3] S. Acharya, M. Franklin, and S. Zdonik, “Balancing push and pull for data broadcast,” *ACM SIGMOD Rec.*, vol. 26, no. 2, pp. 183–194, Jun. 1997.

[4] “EGI in numbers.” [Online]. Available: https://www.egi.eu/infrastructure/operations/egi_in_numbers/. [Accessed: 11-May-2015]

[5] “imap.” [Online]. Available: https://www.npmjs.com/package/imap. [Accessed: 17-Apr-2015]

[6] “npm.” [Online]. Available: https://www.npmjs.com/. [Accessed: 17-Apr-2015]

[7] “JSON.” [Online]. Available: http://json.org/. [Accessed: 17-Apr-2015]

[8] Christos Filippidis, Yiannis Cotronis, and Christos Markou, “Forming an ad-hoc nearby storage, based on IKAROS and social networking services,” *J. Phys. Conf. Ser.*, vol. 513, no. 4, p. 042018, Jun. 2014.

[9] Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, “Synergistic Challenges in Data-Intensive Science and Exascale Computing: Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee,” U. S. Department of Energy, Office of Science, ASCAC Data Subcommittee Report, Mar. 2013 [Online]. Available: http://science.energy.gov/~/media/40749FD92B58438594256267425C4AD1.ashx

[10] Tony Hey, Stewart Tansley, and Kristin Tolle, “The Fourth Paradigm: Data-Intensive Scientific Discovery,” University of North Texas [Online]. Available: http://digital.library.unt.edu/ark:/67531/metadc31516/

[11] John D. White and Earl C. Joseph, “Proceedings of the 1975 annual conference,” in *Proceedings of the 1975 annual conference*, New York, NY, USA, 1975, p. 371.

[12] Data-Intensive Distributed Systems Laboratory,Illinois Institute of Technology,Department of Computer Science, “DataSys: Projects,” 04-Jun-2014. [Online]. Available: http://datasys.cs.iit.edu/projects/FusionFS/. [Accessed: 12-Oct-2014]

[13] Larry Kaplan and Dave Hensele, “Exascale Nearby Storage,” Cray, Inc., Washinton, DC, USA, Submission to Exascale Operating Systems and Runtime Software Collaboration Space, Oct. 2012 [Online]. Available: https://collab.mcs.anl.gov/download/attachments/3801153/exaosr12_submission_17.pdf?version=2&modificationDate=1342893681000

[14] “CMS Public | CMS Experiment.” [Online]. Available: http://cms.web.cern.ch/. [Accessed: 17-Apr-2015]

[15] “The Large Hadron Collider | CERN.” [Online]. Available: http://home.web.cern.ch/topics/large-hadron-collider. [Accessed: 17-Apr-2015]

[16] “CERN | Accelerating science.” [Online]. Available: http://home.web.cern.ch/. [Accessed: 17-Apr-2015]

[17] “KM3NeT- Opens a new window on our universe.” [Online]. Available: http://www.km3net.org/home.php. [Accessed: 17-Apr-2015]

[18] National Research Council (U.S.), *The potential impact of high-end capability computing on four illustrative fields of science and engineering*. Washington, DC: National Academies Press, 2008 [Online]. Available: http://www.nap.edu/catalog.php?record_id=12451

[19] Paul E. Ceruzzi, *A history of modern computing*, 2nd ed. London, Eng. ; Cambridge, Mass: MIT Press, 2003.

[20] Alexis Madrigal, “Exascale Computing Requires Chips, Power and Money,” *Wired*, 22-Feb-2008 [Online]. Available: http://www.wired.com/science/discoveries/news/2008/02/exascale_computing
[21] Ioan Raicu, Ian T. Foster, and Pete Beckman, “Making a case for distributed file systems at Exascale,” 2011, p. 11 [Online]. Available: http://portal.acm.org/citation.cfm?doid=1996029.1996034. [Accessed: 13-Oct-2014]
[22] Christos Filippidis, Yiannis Cotronis, and Christos Markou, “IKAROS: An HTTP-Based Distributed File System, for Low Consumption & Low Specification Devices,” *J. Grid Comput.*, vol. 11, no. 4, pp. 681–698, Dec. 2013.
[23] “Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update 2014–2019 White Paper,” *Cisco*. [Online]. Available: http://cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html. [Accessed: 17-Apr-2015]
[24] Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, “The Opportunities and Challenges of Exascale Computing,” U. S. Department of Energy, Office of Science, Fall 2010 [Online]. Available: http://science.energy.gov/~/media/ascr/ascac/pdf/reports/exascale_subcommittee_report.pdf
[25] T. White, *Hadoop: the definitive guide*, Third edition. Beijing: O’Reilly, 2012.
[26] “Computing | CERN.” [Online]. Available: http://home.web.cern.ch/about/computing. [Accessed: 17-Apr-2015]
[27] Steve Scott, “Titan supercomputer points the way to Exascale,” *Nvidia Blogs*, 12-Oct-2011. [Online]. Available: http://blogs.nvidia.com/blog/2011/10/12/titan-supercomputer-points-the-way-to-exascale/
[28] “Advanced Scientific Computing Advisory Committee (ASCAC) Homepage | U.S. DOE Office of Science (SC).” [Online]. Available: http://science.energy.gov/ascr/ascac. [Accessed: 17-Apr-2015]
[29] Collaborative Research into Exascale Systemware, Tools, and Applications (CRESTA), “CRESTA - Developing techniques and solutions which address the most difficult challenges that computing at the exascale can provide.,” *CRESTA - Developing techniques and solutions which address the most difficult challenges that computing at the exascale can provide.* [Online]. Available: http://cresta-project.eu/
[30] International Exascale Software Project, “Main Page - IESP,” 30-Apr-2012. [Online]. Available: http://www.exascale.org/iesp/Main_Page
[31] Big Data and Extreme-scale Computing (BDEC), “Home | BDEC,” *Home | BDEC*. [Online]. Available: http://www.exascale.org/bdec/
[32] European Exascale Software Initiative, “EESI project - The European Exascale Software Initiative: - Homepage,” *EESI project - The European Exascale Software Initiative: - Homepage*. [Online]. Available: http://www.eesi-project.eu/pages/menu/homepage.php
[33] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 0.80 ed. Arpaci-Dusseau Books, 2014.
[34] H. Stern, *Managing NFS and NIS*. Sebastopol, CA: O’Reilly & Associates, 1991.
[35] Oracle, “NFS File Servers,” *NFS Server Performance and Tuning Guide for Sun Hardware,*. [Online]. Available: http://docs.oracle.com/cd/E19620-01/805-4448/6j47cnj0g/index.html. [Accessed: 17-Apr-2015]
[36] Shane Kerr, “Use of NFS Considered Harmful,” 14-Nov-2000. [Online]. Available: http://www.time-travellers.org/shane/papers/NFS_considered_harmful.html
[37] Amina Saify, Garima Kochharr, Jenwei Hsieh, and Onur Celebioglu, “Enhancing High-Performance Computing Clusters with Parallel File Systems,” *Dell Power Solutions Online Extra*, p. 3, May-2005.
[38] Samuel Lang, “Parallel File Systems,” US Department of Energy, Argonne National Library, 20-Sep-2010 [Online]. Available: http://www.cs.iit.edu/~iraicu/teaching/CS554-F13/lecture17-pfs-sam-lang.pdf

[39] V. Meshram, X. Besseron, X. Ouyang, R. Rajachandrasekar, R. P. Darbha, and D. K. Panda, “Can a Decentralized Metadata Service Layer Benefit Parallel Filesystems?,” 2011, pp. 484–493 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6061137. [Accessed: 11-May-2015]

[40] Dean Hildebrand and Peter Honeyman, “Exporting Storage Systems in a Scalable Manner with pNFS,” presented at the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST ’05), 2005, pp. 18–27 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1410720. [Accessed: 13-Oct-2014]

[41] Dean Hildebrand and Peter Honeyman, “Direct-pNFS: scalable, transparent, and versatile access to parallel file systems,” in *Proceedings of the 16th international symposium on High performance distributed computing HPDC ’07*, 2007, p. 199 [Online]. Available: http://portal.acm.org/citation.cfm?doid=1272366.1272392. [Accessed: 13-Oct-2014]

[42] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur, “PVFS: A Parallel File System for Linux Clusters,” in *Extreme Linux Track: 4th Annual Linux Showcase and Conference*, Atlanta, GA, USA, 2000, pp. 317–327 [Online]. Available: http://www.mcs.anl.gov/papers/P804.pdf

[43] SNIA Ethernet Storage Forum, “An Overview of NFSv4: NFSv4.0, NFSv4.1, pNFS, and proposed NFSv4.2 features,” SNIA Ethernet Storage Forum, White paper, Jun. 2012 [Online]. Available: http://www.snia.org/sites/default/files/SNIA_An_Overview_of_NFSv4-3_0.pdf

[44] Alex McDonald, “NFSv4,” *;login: The Magazine of UYSENIX*, vol. 37, no. 1, pp. 28–35, Feb-2012.

[45] Alexandra Glagoleva and Archana Sathaye, “Load Balancing Distributed File System Servers: A Rule-based Approach,” in *Web-enabled Systems Integration*, Ajantha Dahanayake and Waltraud Gerhardt, Eds. Hershey, PA, USA: IGI Global, 2003, pp. 274–297 [Online]. Available: http://dl.acm.org/citation.cfm?id=762558.762574

[46] R. Esposito, P. Mastroserio, G. Tortone, and F.M. Taurino, “Standard FTP and GridFTP protocols for internationa l data transfer in Pamela Satellite Space Experiment,” presented at the 2003 Conference for Computing in High Energy and Nuclear Physics (CHEP 2003), La Jolla, California, USA, 2003, p. 3 [Online]. Available: http://arxiv.org/pdf/hep-ex/0305084.pdf

[47] J. Postel and J. Reynolds, “File Transfer Protocol,” *Internet Req. Comments*, vol. RFC 959 (INTERNET STANDARD), Oct. 1985 [Online]. Available: http://www.rfc-editor.org/rfc/rfc959.txt

[48] W. Allcock, J. Bresnahan, R. Kettimuthu, and M. Link, “The Globus Striped GridFTP Framework and Server,” presented at the 2005 ACM/IEEE conference on Supercomputing (SC ’05), 2005, pp. 54–54 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1560006

[49] Christos Filippidis, Christos Markou, and Cotronis Yiannis, “IKAROS with brand new, high tech wings is ready to confront the upcoming data challenges in scientific computing infrastructures and exascale environments.”

[50] Ioan Raicu, Zhao Zhang, Mike Wilde, Ian Foster, Pete Beckman, Kamil Iskra, and Ben Clifford, “Toward Loosely Coupled Programming on Petascale Systems,” in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, Piscataway, NJ, USA, 2008, pp. 22:1–22:12 [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413393

[51] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, Xuebin Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Zhong Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel,

H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. van der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The International Exascale Software Project roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011.

[52] Christos Filippidis, Yiannis Cotronis, and Christos Markou, "The IKAROS Metadata service as a Utility," presented at the 7th IEEE/ACM International Conference on Utility and Cloud Computing, 2014.

[53] Christos Filippidis, "Using IKAROS to unify remote and local access in the overall data flow."

[54] Joseph L. Naps, Mohamed F. Mokbel, and David H. C. Du, "Pantheon: Exascale File System Search for Scientific Computing," in *Scientific and Statistical Database Management*, vol. 6809, Judith Bayard Cushing, James French, and Shawn Bowers, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 461–469 [Online]. Available: http://link.springer.com/10.1007/978-3-642-22351-8_29

[55] The Apache Software Foundation, "Dynamic Shared Object (DSO) Support - Apache HTTP Server Version 2.2," 21-May-2014. [Online]. Available: http://httpd.apache.org/docs/2.2/dso.html. [Accessed: 14-Oct-2014]

[56] eλastic | Transfer, "Elastic | Transfer (Innovating File Management)," *Elastic | Transfer (Innovating File Management)*. [Online]. Available: http://www.et-js.org/

[57] Guillermo Rauch, *Smashing Node.js: JavaScript everywhere*. Chichester, West Sussex: John Wiley & Sons, Ltd, 2012.

[58] Joyent, Inc., "node.js," 17-Sep-2014. [Online]. Available: http://www.nodejs.org/

[59] Pedro Dennis Teixeira, *Professional node.js: building javascript based scalable software*, 1st ed. Indianapolis, IN: Wiley Pub., Inc, 2012.

[60] Shelley Powers, *Learning Node*. Sebastopol, CA: O'Reilly Media, 2012.

[61] E. Bozdag, A. Mesbah, and A. van Deursen, "A Comparison of Push and Pull Techniques for AJAX," 2007, pp. 15–22 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4380239. [Accessed: 17-Apr-2015]

[62] M. Franklin and S. Zdonik, "'Data in your face': push technology in perspective," *ACM SIGMOD Rec.*, vol. 27, no. 2, pp. 516–519, Jun. 1998.

[63] M. Welsh, D. Culler, and E. Brewer, "SEDA: an architecture for well-conditioned, scalable internet services," 2001, p. 230 [Online]. Available: http://portal.acm.org/citation.cfm?doid=502034.502057. [Accessed: 17-Apr-2015]

[64] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris, "Event-driven programming for robust software," 2002, p. 186 [Online]. Available: http://portal.acm.org/citation.cfm?doid=1133373.1133410. [Accessed: 17-Apr-2015]

[65] "inbox." [Online]. Available: https://www.npmjs.com/package/inbox. [Accessed: 17-Apr-2015]

Appendix A: A Basic Usage Scenario of the eT Framework

In this section a typical data transfer scenario through the elastic-transfer module will be presented. A data transfer from a host computer to multiple destination hosts will be performed. Additionally, the file sharing capability of the module will be presented.

Initially, the framework has to be installed in all systems, so in the command line of all of these computers the ‘‘npm install elastic-transfer’’ command has to be executed. Moreover, to run the framework the following commands have to be executed on all systems:

cd ~\node_modules\elastic-transfer

node eT.js

The default browser opens and all hosts connect to the loopback address 127.0.0.1 on port 61949. More specifically, they are directed to link 127.0.0.1:61949/gui which corresponds to the starting page of the framework. This index page has been developed in HTML and javascript within the file index5.ejs, located in the folder ~\node_modules \elastic-transfer\views\index5.ejs. Figure A depicts the starting page of the eT module.

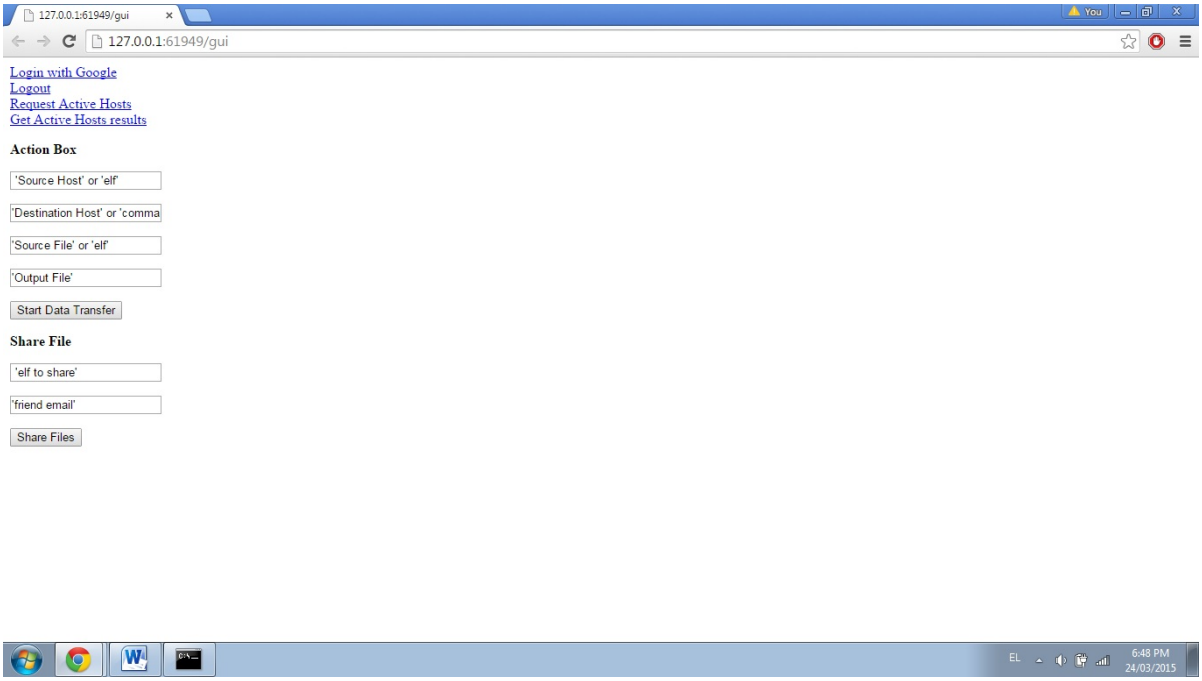

Figure A: eT’s starting page

It should be obvious that eT provides two main operations: data transfers between a source host and a destination or multiple destination hosts, and data sharing services. To use these services all host computers should login into their Gmail account. Hence, in the starting page the ‘‘Login with Google’’ link should be clicked and in the page that follows, each eT host should provide their Gmail credentials, such as username and password. The following screen appears as shown in Figure B.

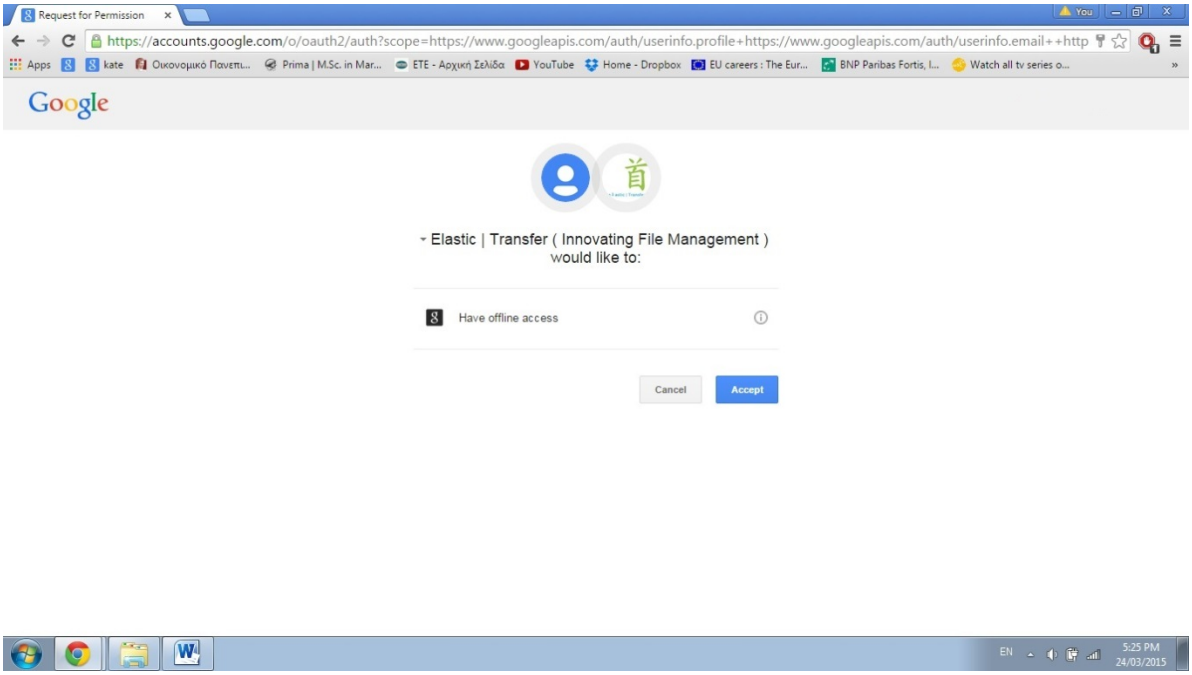

Figure B: eT's connection with Google's Gmail services.

The "Accept" button should be selected and the user is automatically re-directed to the starting page. Afterwards, the source host computer should be informed about the IP addresses of the other connected hosts. This is done by following the "Request Active Hosts" link. The source device is directed to page http://127.0.0.1:61949/mh through the express module that runs in the eT. The following message is displayed to the browser: "please wait while we are fetching Active Hosts". At this time the source computer sends an e-mail to the Gmail account that has been logged into. The title of this e-mail is "etjs:mh" and its body is "etjsbody:mh". Hence the arrival these new e-mails from the host computer consists an event for all connected hosts, which triggers them to send their IP address in an e-mail. Specifically each machine sends an e-mail which has the title "etjs:host", and its body includes its private IP address and port number alongside with the public name that eT.js has assigned to it. This information is also pushed in an array.

As mentioned earlier the implementation of this event-driven logic has been done through the "Imap module" which has been integrated to the eT framework. The previous implementation of eT was based to the conventional way to serve multiple connections at the same time, by serving each connection with a different process or thread. However, this approach could lead to a trivial operation when a large number of requests was arrived to the system at the same time. This asynchronous approach changes the rationale of the system, and the typical way to serve multiple connections at the same time by adopting multiple threads is rejected. Detailed information related to the source code will be provided to the next appendix.

If the user wants to be informed about the IP addresses of all active users through the browser, rather than through e-mail, the "Get Active Hosts Results" link has to be followed. The "Express" module that runs within the eT directs the user to the http://127.0.0.1:61949/mh_result link, which includes the IP addresses of all connected hosts through the array that was mentioned before. For each host its private IP address appears alongside its public FQDN that eT has assigned to it. For example:

http://10.0.1.173:61949,https://kanvmzztqh.localtunnel.me,
http://10.0.1.171:61949,https://yuzpjdzdnm.localtunnel.me,

…

Notice: If a user clicked on the ‘‘Get Active Hosts Results’’ link before selecting the ‘‘Request Active Hosts’’ link, an error occurs since the array with the IP addresses would have been empty and the following message appears: ‘‘Go to STEP 1 and Click: Request "Active Hosts" or just wait... ’’.

A simple example of a data transfer will follow. After connecting N computers to the framework, a file transfer from the source host PC1 to the destination hosts PC2, PC3, ... , PC(N-1) will be made. The original file will be broken into N-2 equal sized data chunks and transmitted to computers PC2, PC3, ... , PC(N-1).

Initially, the IP address of the host computer has to be entered into the first box ('Source Host' or 'elf') and then IP addresses of the destination hosts, PC2, PC3, ... , PC(N-1), have to be entered into the next box ('Destination Host' or 'comma separated Hosts'), separated with comma. The path and the name of the file to be transmitted has be entered into the third box ('Source File' or 'elf'). The file to be transmitted should be stored at the root folder of the source host, in the `~\node_modules\elastic_transfer\eTshare\root\ directory`. For example if the video file `test.avi` is to be transmitted, the third field should be filled as follows: `root\test.avi`. Additionally, in the same root folder of the source host a text file specifying the owner of the file should exist and the word ‘‘owner”, (without the quotation marks) should be written in it. The file’s name should comply to the following format: `name_of_the_file_to_be_transferred.owner’s-gmail-account.txt`

Thus in our example the text file should have the following name: `test.avi.gkikasn@gmail.com.txt` and the word `owner` should be written within the text file.

The fourth field ('Output File') will intentionally be left empty, since it is desired that the file be transferred and stored at the destination hosts with the same name.

Next the user clicks on the ‘‘Start Data Transfer’’ button. The user is directed to a page `http://127.0.0.1:61949/hr` through the module ‘‘Express’’ which runs within the eT and the following message appears: “the transfer just started, please go back to GUI: http://localhost:61949/gui” .

The file is transferred after being spit into N-2 equal sized parts over the destination hosts PC2, PC3, ... , PC(N-1) and is stored in the `~\node_modules\elastic_transfer\eTshare\elfs\` directory of each destination host. At the same time, three new e-mails arrive at the Gmail account of the user, including information regarding this specific data transfer. To be more precise the following messages arrive:

- An e-mail with subject ‘‘etjs:owner’’ which contains the file owner’s name.
- An e-mail with subject ‘‘etjs:elsize’’. Its body includes the size of the transferred file.
- An e-mail with title ‘‘etjs:elf’’. The e-mail’s body contains information related to this specific data transfer, containing the name of the transferred file, the number of data chunks that the initial file was broken into and IP addresses of the destination hosts that the file was delivered to.

It is obvious that these e-mails contain the meta-data files of the transfer. As highlighted the meta-data scheme was changed for the purpose of this thesis, making them ease to parse as JSON objects. The older version of the eT did not included this ability, undermined eT′s interoperability, since as mentioned earlier the JSON format is one of the most popular and wide spread standards to transmit data objects by web applications and distributed systems.

At the same time these e-mails are stored as text files in their hard drive of all hosts. More specifically, the information that is contained in the first e-mail, related to the owner of the file, is downloaded as a text file, the ‘‘`test.avi.gkikasn@gmail.com.txt`’’ to the `~\node_modules\elastic_transfer\eTshare\elfs\` directory of all hosts. Accordingly, two more

text files are created containing the information that is included in the other two e-mails. These text files are the ‘‘test.avi.elfsize.txt’’ and ‘‘test.avi.elf.txt’’ and they are saved into the ~\node_modules\elastic_transfer\eTshare\elfs\mdata directory of all hosts.

Now the same file ‘‘test.avi’’ will be transferred in parallel to destination host PC(N). The word ‘‘elf’’ should be typed in the first field ('Source Host' or 'elf'), and the IP address of the PC(N) in the second field ('Destination Host' or 'comma separated Hosts') of the starting page. Next the name of the file (without the ‘‘\root’’) should be typed in the third field ('Source File' or 'elf'), i.e. ‘‘test.avi’’ . The fourth field ('Output File') is left empty again on purpose, since it is not needed to change the name of the transferred file. The button ‘‘Start Data Transfer’’ is selected and the data transfer is begins. Since the destination host PC(N) has access to all of the meta-data files that include information related to previously transmitted ‘‘test.avi’’ file, and to its data chunks, this time the data transfer is made in parallel from PC2, PC3, ... , PC(N-1) to PC(N). Each computer of PC2, PC3, ... , PC(N-1) sends each the data chunk of the previously transmitted ‘‘test.avi’’ and PC(N) pieces them together creating the original file.

Regarding the share operation, if an already transferred file has to be shared with a third user, the file’s name has to be written in the ‘‘elf to share’’ field and the user's e-mail has to be filled in the ‘‘friend’s e-mail’’ field. Then the ‘‘Share Files’’ button is selected and the third user receives an e-mail with the meta-data files that are related with the to be transferred file. Thus the third user is able to receive the relevant file from the host source(s).

After using eT, the hosts should disconnect from the system by selecting the ‘‘Logout’’ link from the starting page of the framework.

Appendix B: The Latest Version of the eT's Source Code

```
1  //License
2  //Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)
3  //http://creativecommons.org/licenses/by-sa/3.0/
4  // Elastic Transfer
5  // elastic.transfer@gmail.com
6  //V0.3.95 - 2/07/2014
7  var http = require('http'),
8      https = require('https'),
9      express = require('express'),
10     events = require('events'),
11     net = require('net'),
12     util = require('util'),
13     path = require('path'),
14     localtunnel = require('localtunnel'),
15     nodemailer = require('nodemailer'),
16     passport = require('passport'),
17     util = require('util'),
18     bodyParser = require('body-parser'),
19     cookieParser = require('cookie-parser'),
20     session = require('express-session'),
21     morgan = require('morgan'),
22     directory = require('serve-index'),
23     methodOverride = require('method-override'),
24     GoogleStrategy = require('passport-google-oauth').OAuth2Strategy,
25     fs = require('fs'),
26     os = require('os');
27 var Imap = require('imap'),
28     inspect = require('util').inspect;
29 var xoauth2 = require("xoauth2"),
30     xoauth2gen;
31 var request = require('request');
32 var open = require('open');
33 var imap_server = "imap.gmail.com";
34 var imap_port = true; // true = 993, false = 143
35 var choose_mailbox = "INBOX";
36 var publicport = 61949;
37 var my_host_UUID;
38 var address = [];
39 var emailuid = [];
40 var get_mh = [];
41 var paddress;
42 var gtoken;
43 var grtoken;
44 var guser;
45 var service = 'http://127.0.0.1:' + publicport + '/gui';
46 var service_host = 'http://127.0.0.1:' + publicport;
47 var GOOGLE_CLIENT_ID = "213412485889-
poh1fu074b8kpep27bj7v97ukcg0j20u.apps.googleusercontent.com";
48 var GOOGLE_CLIENT_SECRET = "Z6HmKu8BFfttHIggadCh8buX";
49
50 var getip = function(callback) {
51     var ifaces = os.networkInterfaces();
52     for (var dev in ifaces) {
53         var alias = 0;
54         ifaces[dev].forEach(function(details) {
55             if (details.family == 'IPv4') {
56                 if (details.address != '127.0.0.1') {
57                     address.push(details.address);
```

```
                    ++alias;
                }
            }
        });
    }
}

function randomString(length) {
    var chars =
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXTZabcdefghiklmnopqrstuvwxyz'.split('');
    if (!length) {
        length = Math.floor(Math.random() * chars.length);
    }
    var str = '';
    for (var i = 0; i < length; i++) {
        str += chars[Math.floor(Math.random() * chars.length)];
    }
    return str;
}

var tweet = function(message, sub, from, to, callback) {

    if (to == '0') {
        to = from;
    }
    var transport = nodemailer.createTransport("SMTP", {
        service: 'Gmail', // use well known service
        auth: {
            XOAuth2: {
                user: from,
                clientId: GOOGLE_CLIENT_ID,
                clientSecret: GOOGLE_CLIENT_SECRET,
                refreshToken: grtoken
            }
        },
        debug: true
    });
    var message = {
        from: from, // sender address
        to: to, // list of receivers
        subject: 'etjs:' + sub, // Subject line
        text: message, // plaintext body
        //The encoding defaults to 'quoted-printable'. We specify utf8
encoding.
        encoding: 'utf8'
    }
    transport.sendMail(message, function(error, response) {
        if (error) {
            console.log(error);
        } else {
            console.log("Message sent: " + response.message);
        }
        // if you don't want to use this transport object anymore,
uncomment following line
        //transport.close(); // shut down the connection pool, no more
messages
    });
}

var ic = function(email, imap_port, callback) {
    xoauth2gen = xoauth2.createXOAuth2Generator({
```

```
115         user: guser,
116         clientId: GOOGLE_CLIENT_ID,
117         clientSecret: GOOGLE_CLIENT_SECRET,
118         refreshToken: grtoken
119     });
120     xoauth2gen.getToken(function(err, gbasetoken) {
121         if (err) {
122             return console.log(err);
123         }
124         console.log("AUTH XOAUTH2 " + gbasetoken);
125         var imap = new Imap({
126             xoauth2: gbasetoken,
127             user: guser,
128             //password: '',
129             host: imap_server,
130             port: 993,
131             tls: true,
132             // keepalive : true,
133             debug: console.log,
134             tlsOptions: {
135                 rejectUnauthorized: false
136             }
137         });
138 
139         function openInbox(cb) {
140             imap.openBox(choose_mailbox, true, cb);
141         }
142         imap.once('ready', function() {
143             openInbox(function(err, box) {
144                 if (err) throw err;
145                 imap.on('mail', function(numNewMsgs) {
146                     console.log('numNewMsgs #%d', numNewMsgs);
147                     console.log('box.messages.total #%d',
box.messages.total);
148                     var nM = box.messages.total - numNewMsgs + 1;
149                     console.log('nM #%d', nM);
150                     var f = imap.seq.fetch(nM + ':' +
box.messages.total, {
151                         bodies: ['HEADER.FIELDS (SUBJECT)', 'TEXT']
152                     });
153                     f.on('message', function(msg, seqno) {
154                         console.log('Message #%d', seqno);
155                         var prefix = '(#' + seqno + ') ';
156                         var title = '';
157                         var body = '';
158                         msg.on('body', function(stream, info) {
159                             if (info.which === 'TEXT')
160                                 console.log(prefix + 'Body [%s] found,
%d total bytes', inspect(info.which), info.size);
161                             var buffer = '',
162                                 count = 0;
163                             stream.on('data', function(chunk) {
164                                 count += chunk.length;
165                                 buffer += chunk.toString('utf8');
166                                 if (info.which === 'TEXT')
167                                     console.log(prefix + 'Body [%s]
(%d/%d)', inspect(info.which), count, info.size);
168                             });
169                             stream.once('end', function() {
170                                 if (info.which !== 'TEXT') {
```

```
171                                     console.log(prefix + 'Parsed header:
%s', inspect(Imap.parseHeader(buffer)));
172                                     title =
Imap.parseHeader(buffer).subject;
173                                     if (title == 'etjs:mh') {
174
tweet('{\"etjsbody\":{\"privateaddress\":\"http://' + address[0] + ':' +
publicport + '\",\"publicaddress\":\"' + paddress + '\"}}', 'host', email,
0, function() {});
175                                     }
176                                     if (title == 'etjs:host') { //Push
host's private and public addresses into an array
177                                         try{
178                                             var
jbodyhost=JSON.parse(body);
179                                         }catch(err){console.log('Host
message JSON parsing error: '+err)}
180
get_mh.push(jbodyhost.etjsbody.privateaddress,
jbodyhost.etjsbody.publicaddress);
181                                         //nullify object to assist
garbage collection
182                                         jbodyhost = null;
183                                     }
184                                     if
(title.toString().indexOf('etjs:elf') != -1) {
185                                         try{
186                                             var jbodyelf =
JSON.parse(body);
187                                         }catch(err){console.log('elf
message JSON parsing error: '+err)}
188                                         var jbodyelfbodystring =
JSON.stringify(jbodyelf.etjsbody);
189                                         fs.createWriteStream(__dirname +
'/eTshare/elfs/mdata/' + jbodyelf.etjsbody.outfile +
'.elf.txt').write(jbodyelfbodystring);
190                                         //nullify object to assist
garbage collection
191                                         jbodyelf = null;
192                                     }
193                                     if
(title.toString().indexOf('etjs:elsize') != -1) {
194                                         try{
195                                             var jbodyelfsize =
JSON.parse(body);
196                                         }catch(err){console.log('elsize
message JSON parsing error: '+err)}
197                                         var jelffilesizestring =
jbodyelfsize.etjsbody.filesize.toString(); //Make the
jbodyelfsize.etjsbody.filesize nunmber value into a string in order to pass
it to fs.createWriteStream().write()
198                                         fs.createWriteStream(__dirname +
'/eTshare/elfs/mdata/' + jbodyelfsize.etjsbody.filename +
'.elfsize.txt').write(jelffilesizestring);
199                                         //nullify object to assist
garbage collection
200                                         jbodyelfsize = null;
201                                     }
202                                     if
(title.toString().indexOf('etjs:owner') != -1) {
203                                         try{
```

```
204                                             var jbodyowner =
JSON.parse(body);
205                                         }catch(err){console.log('owner
message JSON parsing error: '+err)}
206                                         fs.createWriteStream(__dirname +
'/eTshare/elfs/' + jbodyowner.etjsbody.filename + '.' +
jbodyowner.etjsbody.fileowner + '.txt').write("owner");
207                                         //nullify object to assist
garbage collection
208                                         jbodyowner = null;
209                                     }
210                                 } else {
211                                     console.log(prefix + 'Body [%s]
Finished --> [%s]', inspect(info.which), inspect(buffer));
212                                     body = buffer;
213                                 }
214                             });
215                         });
216                         msg.once('attributes', function(attrs) {
217                             console.log(prefix + 'Attributes: %s',
inspect(attrs, false, 8));
218                         });
219                         msg.once('end', function() {
220                             console.log(prefix + 'Finished ');
221                         });
222                     });
223                     f.once('error', function(err) {
224                         console.log('Fetch error: ' + err);
225                     });
226                     f.once('end', function() {
227                         console.log('Done fetching new message!');
228                     });
229                 });
230             });
231         });
232         imap.once('error', function(err) {
233             console.log('imap error: ' + err);
234             ic(email, imap_port, function() {});
235             console.log('imap reconnect');
236         });
237         imap.once('end', function() {
238             console.log('client on END');
239             ic(email, imap_port, function() {});
240             console.log('imap reconnect');
241         });
242         imap.connect();
243     }); //xoauth2 generator
244};
245
246getip(function() {});
247// Passport session setup.
248//   To support persistFent login sessions, Passport needs to be able to
249//   serialize users into and deserialize users out of the session.
Typically,
250//   this will be as simple as storing the user ID when serializing, and
finding
251//   the user by ID when deserializing.  However, since this
implementation does not
252//   have a database of user records, the complete Google profile is
253//   serialized and deserialized.
254passport.serializeUser(function(user, done) {
```

```
255     done(null, user);
256});
257
258passport.deserializeUser(function(obj, done) {
259     done(null, obj);
260});
261// Use the GoogleStrategy within Passport.
262//   Strategies in Passport require a `verify` function, which accept
263//   credentials (in this case, an accessToken, refreshToken, and Google
264//   profile), and invoke a callback with a user object.
265passport.use(new GoogleStrategy({
266         clientID: GOOGLE_CLIENT_ID,
267         clientSecret: GOOGLE_CLIENT_SECRET,
268         callbackURL: service_host + "/auth/google/callback"
269             //passReqToCallback: true
270     },
271     function(accessToken, refreshToken, profile, done) {
272         gtoken = accessToken;
273         grtoken = refreshToken;
274         guser = profile.emails[0].value;
275         ic(guser, imap_port, function() {});
276         // asynchronous verification, for effect...
277         process.nextTick(function() {
278             // To keep the this simple, the user's Google profile is
returned to
279             // represent the logged-in user.  In a typical application,
you would want
280             // to associate the Google account with a user record in
your database,
281             // and return that user instead.
282             return done(null, profile);
283         });
284     }
285));
286
287var app = express();
288app.use(bodyParser());
289app.use(morgan());
290app.use(cookieParser());
291app.use(methodOverride());
292app.use(session({
293     secret: randomString()
294}));
295// Initialize Passport!  Also use passport.session() middleware, to
support
296// persistent login sessions (recommended).
297app.use(passport.initialize());
298app.use(passport.session());
299//app.use(directory(__dirname + '/eTshare'));
300//app.use(express.static(__dirname + '/eTshare'));
301app.get('/ic', ensureAuthenticated, function(req, res) {
302     ic(guser, imap_port, function() {});
303     res.send('trying to reconnect to IMAP server...');
304});
305app.get('/r', function(req, res) {
306     var snumber = req.query["snumber"];
307     var ntotal = req.query["ntotal"];
308     var infile = req.query["infile"];
309     var outfile = req.query["outfile"];
310     var own = req.query["own"];
311     var elfv = req.query["elfv"];
```

```
312     var outfr = outfile.replace("elfs/", "");
313     var gtk = req.query["gtk"];
314     var host = req.headers.host;
315     var host_plain = host.split(':');
316
request('https://www.googleapis.com/oauth2/v1/tokeninfo?access_token=' +
gtk, function(error, response, body) {
317             if (!error && response.statusCode == 200) {
318                 fs.exists(__dirname + '/eTshare/' + infile + '.' + own +
'.txt', function(exists) {
319                     if (exists && JSON.parse(body).email == own &&
JSON.parse(body).verified_email == true && JSON.parse(body).expires_in !=
null) {
320                         var stat = fs.statSync(__dirname + '/eTshare/' +
infile);
321                         if (!stat.isFile()) return;
322                         var t = snumber - 1;
323                         var ch_size = Math.floor(stat.size / ntotal);
324                         var remainder = stat.size % ntotal;
325                         var start = ((ch_size) * t + 1);
326                         var end = ch_size * (t + 1);
327                         if (t == 0) {
328                             start = (ch_size) * t;
329                         }
330                         if (t == ntotal - 1) {
331                             end = (ch_size * (t + 1) + remainder);
332                         }
333                         var range = req.header('Range');
334                         if (range != null) {
335                             start =
parseInt(range.slice(range.indexOf('bytes=') + 6,
336                                 range.indexOf('-')));
337                             end = parseInt(range.slice(range.indexOf('-
') + 1,
338                                 range.length));
339                         }
340                         if (isNaN(end) || end == 0) end = stat.size - 1;
341                         if (start > end) return;
342                         var date = new Date();
343                         res.writeHead(206, { // NOTE: a partial http
response
344                             // 'Date':date.toUTCString(),
345                             'Connection': 'close',
346                             // 'Cache-Control':'private',
347                             // 'Content-Type':'video/webm',
348                             // 'Content-Length':end - start,
349                             'Content-Range': 'bytes ' + start + '-' +
end + '/' + stat.size,
350                             // 'Accept-Ranges':'bytes',
351                             // 'Server':'CustomStreamer/0.0.1',
352                             'Transfer-Encoding': 'chunked'
353                         });
354                         var stream = fs.createReadStream(__dirname +
'/eTshare/' + infile, {
355                             flags: 'r',
356                             start: start,
357                             end: end
358                         });
359                         stream.pipe(res);
360                         if (snumber == '1' && elfv != '1') {
```

```
361                             tweet('{\"etjsbody\":{\"filename\":\"' +
outfr + '\",\"filesize\":' + stat.size + '}}', 'elsize:' + outfr, own, 0,
function() {});
362                             tweet('{\"etjsbody\":{\"filename\":\"' +
outfr + '\",\"fileowner\":\"' + own + '\"}}', 'owner:' + outfr + ':' + own,
own, 0, function() {});
363                         }
364                     } //end of if, in exist
365                     else {
366                         console.log('you do not have permission to
access this resource');
367                     }
368                 }); //end of  exist
369             } //end of if, in request
370         }) // end of request
371});
372app.get('/gui', function(req, res) {
373    res.render('index5.ejs', {
374        layout: false
375    });
376});
377//express GET function. It triggers in the ic function the tweet
function that emails the hosts' private and public addresses
378app.get('/mh', ensureAuthenticated, function(req, res) {
379    get_mh.length = 0;
380    tweet('{\"etjsbody\":\"mh\"}', 'mh', guser, 0, function() {});
381    res.send('please wait while we are fetching  Active Hosts');
382});
383//used to display the hosts' addresses
384app.get('/mh_result', ensureAuthenticated, function(req, res) {
385    if (get_mh == '') {
386        res.send('Go to STEP 1 and Click: Request "Active Hosts" or just
wait... ');
387    } else {
388        res.send(get_mh.toString());
389    }
390});
391app.get('/rq', function(req, res) {
392
393    var source = req.query["source"];
394    var snumber = req.query["snumber"];
395    var ntotal = req.query["ntotal"];
396    var infile = req.query["infile"];
397    var outfile = req.query["outfile"];
398    var elfsize = req.query["elfsize"];
399    var own = req.query["own"];
400    var gtk = req.query["gtk"];
401    var inelf = infile.split('/');
402
403    if (inelf[0] == 'elfs') {
404        var t = snumber - 1;
405        var ch_size = Math.floor(elfsize / ntotal);
406        var remainder = elfsize % ntotal;
407        var start = ((ch_size) * t + 1);
408        var end = ch_size * (t + 1);
409        if (t == 0) {
410            start = (ch_size) * t;
411        }
412        if (t == ntotal - 1) {
413            end = (ch_size * (t + 1) + remainder);
414        }
```

```
415         var pos = parseInt(start, 10);
416         var drq = source + '/r/?' + 'infile=' + infile + '&snumber=1' +
'&ntotal=1' + '&outfile=' + outfile + '&elfv=1' + '&own=' + own + '&gtk=' +
gtk;
417         fs.exists(__dirname + '/eTshare/' + outfile, function(exists) {
418             if (exists) {
419                 request(drq).pipe(fs.createWriteStream(__dirname +
'/eTshare/' + outfile, {
420                     'flags': 'r+',
421                     start: pos
422                 }));
423             } else {
424                 request(drq).pipe(fs.createWriteStream(__dirname +
'/eTshare/' + outfile, {
425                     'flags': 'w+',
426                     start: pos
427                 }));
428             }
429         });
430     } else {
431         var drq = source + '/r/?' + 'infile=' + infile + '&snumber=' +
snumber + '&ntotal=' + ntotal + '&outfile=' + outfile + '&own=' + own +
'&gtk=' + gtk;
432         request(drq).pipe(fs.createWriteStream(__dirname + '/eTshare/' +
outfile));
433     }
434     res.send('just moving files ');
435});
436/* share your files*/
437app.post('/share', ensureAuthenticated, function(req, res) {
438     var source = req.body.user.source;
439     var edestination = req.body.user.edestination;
440     var mdestination = req.body.user.mdestination;
441     //split the edestination string and create an array.Make array into
JSON string
442     var edestinations = edestination.split(',');
443     var jowner = JSON.stringify(edestinations);
444     //File existence check
445     fs.exists(__dirname + '/eTshare/elfs/mdata/' + source + '.elf.txt',
function(exists) {
446         if (exists) {
447             fs.readFile(__dirname + '/eTshare/elfs/mdata/' + source +
'.elf.txt', 'utf8', function(err, data) {
448                 if (err) {
449                     console.log('error :: 10070 ::' + err);
450                 }
451                 tweet('{\"etjsbody\":' + data + '}', 'elf:' + source,
guser, edestination, function() {});
452             });
453         }
454     });
455     fs.exists(__dirname + '/eTshare/elfs/mdata/' + source +
'.elfsize.txt', function(exists) {
456         if (exists) {
457             fs.readFile(__dirname + '/eTshare/elfs/mdata/' + source +
'.elfsize.txt', 'utf8', function(err, data) {
458                 if (err) {
459                     console.log('error :: 10071 ::' + err);
460                 }
461                 console.log('req.user.emails[0].value2 :: ' +
req.user.emails[0].value);
```

```
462                 tweet('{\"etjsbody\":{\"filename\":\"' + source +
'\",\"filesize\":\"' + data + '\"}}', 'elsize:' + source,
req.user.emails[0].value, edestination, function() {});
463             });
464         }
465     });
466     fs.exists(__dirname + '/eTshare/elfs/' + source + '.' +
req.user.emails[0].value + '.txt', function(exists) {
467         if (exists) {
468             fs.readFile(__dirname + '/eTshare/elfs/' + source + '.' +
req.user.emails[0].value + '.txt', 'utf8', function(err, data) {
469                 if (err) {
470                     console.log('error :: 10072 ::' + err);
471                 }
472                 console.log('req.user.emails[0].value :: ' +
req.user.emails[0].value);
473                 tweet('{\"etjsbody\":{\"filename\":\"' + source +
'\",\"fileowner\":' + jowner + '}}', 'owner:' + source + ':' +
edestination, req.user.emails[0].value, req.user.emails[0].value,
function() {});
474                 tweet('{\"etjsbody\":{\"filename\":\"' + source +
'\",\"fileowner\":' + jowner + '}}', 'owner:' + source + ':' +
edestination, req.user.emails[0].value, edestination, function() {});
475             });
476         }
477     });
478     res.send('just sharing files');
479});
480/* make your request*/
481app.post('/hr', ensureAuthenticated, function(req, res) {
482     var source = req.body.user.source;
483     var destination = req.body.user.destination;
484     var infile = req.body.user.infile;
485     var outfile = req.body.user.outfile;
486     if (outfile == "'Output File'" || outfile == '') {
487         var outf = infile.split('/');
488         outfile = outf.pop()
489     }
490     var sources = source.split(',');
491     var destinations = destination.split(',');
492     var sl = sources.length;
493     var dl = destinations.length;
494     //elf file existence check,i.e. 'Has this file been transfered
before?'
495     if (sources[0] == 'elf') {
496         fs.exists(__dirname + '/eTshare/elfs/mdata/' + infile +
'.elf.txt', function(exists) {
497             if (exists) {
498                 fs.readFile(__dirname + '/eTshare/elfs/mdata/' + infile
+ '.elf.txt', 'utf8', function(err, data) {
499                     if (err) {
500                         console.log('error :: 107 ::' + err);
501                     }
502                     fs.exists(__dirname + '/eTshare/elfs/mdata/' +
infile + '.elfsize.txt', function(exists) {
503                         if (exists) {
504                             fs.readFile(__dirname +
'/eTshare/elfs/mdata/' + infile + '.elfsize.txt', 'utf8', function(err, ds)
{
505                                 if (err) {
```

```
506                                     console.log('error :: 1077 ::' +
err);
507                                 }
508                                 try{
509                                     var jdata = JSON.parse(data);
510                                 }catch(err){console.log(' metadata JSON
parsing error: '+err)}
511                                 //The outer loop is meant for the
separate blocks that create the elf
512                                 //The inner loop is meant for the
destinations that the elf will be sent
513                                 for (var block = 0; block < jdata.dl;
block++) {
514                                     var sn = block + 1;
515                                     for (var target = 0; target < dl;
target++) {
516                                         var rq = destinations[target] +
'/rq/?source=' + jdata.destination[block] + '&infile=elfs/' + infile +
'&outfile=transfers/' + outfile + '&snumber=' + sn + '&ntotal=' + jdata.dl
+ '&elfsize=' + ds + '&gtk=' + gtoken + '&own=' + req.user.emails[0].value;
517                                         request(rq, function(error,
response, body) {
518                                             if (!error &&
response.statusCode == 200) {}
519                                         }) // end of request
520                                     }
521                                 } // end of for loop
522                                 //nullify object to assist garbage
collection
523                                 jdata = null;
524                             });
525                         }
526                     });
527                 });
528             } else {}
529         });
530     } else {
531         var dest = [];
532         for (var i = 0; i < dl; i++) {
533             var sn = i + 1;
534             var rq = destinations[i] + '/rq/?source=' + sources[0] +
'&infile=' + infile + '&outfile=elfs/' + outfile + '&snumber=' + sn +
'&ntotal=' + dl + '&gtk=' + gtoken + '&own=' + req.user.emails[0].value;
535             //We will append the destinations[i] string to the dest
array
536             dest[dest.length] = destinations[i];
537             request(rq, function(error, response, body) {
538                     if (!error && response.statusCode == 200) {}
539                 }) // end of request
540         } // end of for loop
541         var jdest = JSON.stringify(dest);
542         tweet('{\"etjsbody\":{\"outfile\":\"' + outfile + '\",\"sn\":' +
sn + ',\"dl\":' + dl + ',\"destination\":' + jdest + '}}', 'elf:' +
outfile, req.user.emails[0].value, 0, function() {});
543     }
544     res.send('the transfer just started, please go back to GUI:
http://localhost:' + publicport + '/gui');
545});
546app.get('/auth/google',
547     passport.authenticate('google', {
548         scope: ['https://www.googleapis.com/auth/userinfo.profile',
```

```
549             'https://www.googleapis.com/auth/userinfo.email', '
https://mail.google.com/'
550         ],
551         accessType: 'offline',
552         approvalPrompt: 'force'
553     }),
554     function(req, res) {
555         // The request will be redirected to Google for authentication,
so this
556         // function will not be called.
557         // console.log('passport:: ' + req.email)
558     });
559// GET /auth/google/callback
560//   Use passport.authenticate() as route middleware to authenticate the
561//   request.  If authentication fails, the user will be redirected back
to the
562//   login page.  Otherwise, the primary route function function will be
called,
563//   which, in this example, will redirect the user to the home page.
564app.get('/auth/google/callback',
565     passport.authenticate('google', {
566         failureRedirect: '/gui'
567     }),
568     function(req, res) {
569         res.redirect('/gui');
570     });
571app.get('/logout', function(req, res) {
572     req.logout();
573     res.redirect('/gui');
574});
575app.listen(publicport, function() {});
576localtunnel(publicport, function(err, tunnel) {
577     if (err) {
578         console.log('error :: 121 :: problem on localtunnel : ' + err);
579     }
580     tunnel.url;
581     paddress = tunnel.url;
582     console.log(tunnel.url);
583});
584
585open(service, function(err) {
586     if (err) {
587         console.log('error :: 120 :: please open your browser, pointing
: ' + service);
588     }
589     console.log('status :: running');
590});
591//});// end of prompt
592// Simple route middleware to ensure user is authenticated.
593//   Use this route middleware on any resource that needs to be
protected.  If
594//   the request is authenticated (typically via a persistent login
session),
595//   the request will proceed.  Otherwise, the user will be redirected
to the
596//   login page.
597function ensureAuthenticated(req, res, next) {
598     if (req.isAuthenticated()) {
599         return next();
600     }
601     res.redirect('/gui');
```

```
602}
```

TRITA-ICT-EX-2015:29