Department of Information Science and Technology

Nomadic Fog Storage

A Dissertation presented in a partial fulfillment of the Requirements for the Degree of Master

in Telecommunications and Information Science

by

Ruben Oliveira Vales

Supervisor:

Professor Rui Neto Marinheiro,
ISCTE-IUL

Co-Supervisor:

Professor José André Moura,
ISCTE-IUL

October 2017

Abstract

Mobile services incrementally demand for further processing and storage. However, mobile devices are known for their constrains in terms of processing, storage, and energy. Early proposals have addressed these aspects; by having mobile devices access remote clouds. But these proposals suffer from long latencies and backhaul bandwidth limitations in retrieving data. To mitigate these issues, edge clouds have been proposed. Using this paradigm, intermediate nodes are placed between the mobile devices and the remote cloud. These intermediate nodes should fulfill the end users' resource requests, namely data and processing capability, and reduce the energy consumption on the mobile devices' batteries.

But then again, mobile traffic demand is increasing exponentially and there is a greater than ever evolution of mobile device's available resources. This urges the use of mobile nodes' extra capabilities for fulfilling the requisites imposed by new mobile applications. In this new scenario, the mobile devices should become both consumers and providers of the emerging services. The current work researches on this possibility by designing, implementing and testing a novel nomadic fog storage system that uses fog and mobile nodes to support the upcoming applications. In addition, a novel resource allocation algorithm has been developed that considers the available energy on mobile devices and the network topology. It also includes a replica management module based on data popularity. The comprehensive evaluation of the fog proposal has evidenced that it is responsive, offloads traffic from the backhaul links, and enables a fair energy depletion among mobiles nodes by storing content in neighbor nodes with higher battery autonomy.

***Keywords*:** Mobile cloud computing; Fog storage; Mobile devices; Node localization; Energy awareness; HDFS

Resumo

Os serviços móveis requerem cada vez mais poder de processamento e armazenamento. Contudo, os dispositivos móveis são conhecidos por serem limitados em termos de armazenamento, processamento e energia. Como solução, os dispositivos móveis começaram a aceder a estes recursos através de nuvens distantes. No entanto, estas sofrem de longas latências e limitações na largura de banda da rede, ao aceder aos recursos. Para resolver estas questões, foram propostas soluções de *edge computing*. Estas, colocam nós intermediários entre os dispositivos móveis e a nuvem remota, que são responsáveis por responder aos pedidos de recursos por parte dos utilizadores finais.

Dados os avanços na tecnologia dos dispositivos móveis e o aumento da sua utilização, torna-se cada mais pertinente a utilização destes próprios dispositivos para fornecer os serviços da nuvem. Desta forma, o dispositivo móvel torna-se consumidor e fornecedor do serviço nuvem. O trabalho atual investiga esta vertente, implementado e testando um sistema que utiliza dispositivos móveis e nós no "*fog*", para suportar os serviços móveis emergentes. Foi ainda implementado um algoritmo de alocação de recursos que considera os níveis de energia e a topologia da rede, bem como um módulo que gere a replicação de dados no sistema de acordo com a sua popularidade. Os resultados obtidos provam que o sistema é responsivo, alivia o tráfego nas ligações no core, e demonstra uma distribuição justa do consumo de energia no sistema através de uma disseminação eficaz de conteúdo nos nós da periferia da rede mais próximos dos nós consumidores.

***Palavras-chave*:** *Mobile cloud computing;* Armazenamento no *fog*; Dispositivos móveis; Localização dos nós; Perceção energética; HDFS

Acknowledgements

I would like to thank my supervisor, Professor Rui Neto Marinheiro and co-supervisor, Professor José Moura, for lending their time and knowledge, and guiding me through every step of this thesis. I wouldn't have been able to do it without you.

I'd also like to thank my parents for trying their best at keeping me motivated. I'd like to thank my sister for always annoying me, as well.

In last place, I'd like to thank my colleague Diogo Sequeira, for all the time he spent discussing with me aspects of the current work, or just listening to my monologues.

Figures

Tables

Equations

List of Acronyms

AP	Access Point
API	Application Programming Interface
BS	Base Station
CACTSE	Cloudlet Aided Cooperative Terminals Service Environment
CAN	Content Addressable Network
CI	Confidence Interval
CPU	Core Processing Unit
DFS	Distributed File System
DHT	Distributed Hash Table
DSS	Distributed Sharing System
E-DRM	Extended Database State Machine
EC	Edge Caching
EECRS	Energy Efficient Content Retrieval Scheme
ERMS	Elastic Replication Management System
ETSI	European Telecommunications Standards Institute
FUSE	Filesystem in Userspace
GPL	General Public License
GPS	Global Positioning System
HDFS	Hadoop Distributed File System
HTTP	Hypertext Transfer Protocol
ICN	Information Centric Network
JMX	Java Management Extensions
JVM	Java Virtual Machine
LCC	Local Content Cache
LMH	Large Mobile Host
MANET	Mobile Ad hoc Network
MCC	Mobile Cloud Computing
MDS	Metadata Server
MEC	Multi-Access Edge Computing
MOMCC	Market Oriented Mobile Cloud Computing
NAS	Network Attached storage
NC	Network Coordinates

NCS	Network Coordinates System
OS	Operating System
OSD	Object Storage Device
RAM	Random Access Memory
RAN	Radio Access Network
RD	Redirect
RE	Relative Error
REST	Representational State Transfer
RM	Replication Management
RPC	Remote Procedure Call
RRD	Round Robin Database
RT	Resting Time
RTT	Round Trip Time
SC	Service Coordinator
SM	Service Manager
SMH	Small Mobile Host
SNM	Shot Noise Model
SOA	Service Oriented Architecture
SPOF	Single Point of Failure
TC	Traffic Control
TCP	Transmission Control Protocol
TIV	Triangle Inequality Violation
USB	Universal Serial Bus
VM	Virtual Machine
WAN	Wide Area Network
WD	Weighted Distance
WLAN	Wireless Local Area Network

Contents

Chapter 1. Introduction

1.1 Motivation and Context

Mobile devices are inherently resource constrained in terms of storage, processing and energy. These resource constraints can present themselves as a limitation for running applications on mobile devices [1] [2] [3]**.** The mobile cloud computing paradigm is a solution to mobile devices' resource constraints. It provides storage and processing resources, as a service to mobile end users [1]. Typically, mobile cloud computing refers to an infrastructure where both the processing power and data storage happen outside of the mobile device [4].

At first, the mobile cloud computing (MCC) paradigm has regarded mobile devices as consumers of cloud services, through applications, that use corporate servers as a meeting point. However, these central servers are prune to single points of failure [5], and are typically located far away from mobile users. With advancements in technology and user demand, applications have started to require near real-time processing of data. These applications include high quality multimedia, distributed interactive games, streaming demands and large data transfer rates with low latency [6]. The high latency experienced by mobile users when accessing the remote cloud, associated to the distance between the two, started to be a problem in the mobile cloud paradigm [7]. As such, MCC evolved, to include the edge computing paradigm [8]. Edge computing tries to overcome the latency issue by introducing an intermediate layer between the cloud and end-users, responsible for fulfilling the emerging mobile users' resource requests. Progress has been made in this regard, in the form of cloudlets [3] [9] [10], which are stationary machines in the vicinity of mobile devices. The cloudlets act as resource providers. However, they may not always be available at the already operating network infrastructure. Other progresses in edge computing, include mobile edge computing (MEC) [11] [8] and fog computing [12] [8] [9]. MEC deploys servers that offer cloud computing capabilities inside the Radio Access Network (RAN). But the resources are accessed via cellular networks, with high cost for users. In the fog computing solution, storage and processing needs are addressed by nearby computing nodes that either fulfil the resources or forward them to nodes that can. These nodes are typically access routers and machine to machine (M2M) gateways [8].

With the proliferation of smartphones [13], and the continuous evolution regarding the available resources of these devices, it becomes ever more viable to use their extra capacity in a distributed and coordinated manner, to provide for cloud computing services. By doing this, we

envision that it is possible to bring the cloud service even closer to the consumer, by having consumers also playing the role of providers.

The current work follows this line of research and aims to provide cloud storage services. A system that uses mobile device's resources to provide for a cloud service, needs first and foremost a way to manage resource allocation in the system. The resource allocation manager has the goal of making data available within the system, and keeping data close to the end users that request data. This is hard to achieve in a mobile scenario, due to the following issues: mobile devices' resource constraints, mobility, and ephemeral presence in the network. However, these problems can be overcome, by using networking and node information in the resource allocating decisions. Metrics such as data's popularity, node's energy levels, available storage capacity, connectivity level, and node's localizations, could be used. It is important that allocated resources show a fair distribution of energy depletion, among nodes. Additionally, the system should scalable, fault tolerant and transparent [14].

1.2 Research Questions and Goals

The main question that the current work seeks to answer is:

- Given mobile devices' energy constraints and nomadicity, can an open access data storage cloud system be designed and implemented, that has mobile devices act as both consumers and providers for the cloud service, that satisfies mobile applications regarding responsiveness, while providing a fair distribution of energy depletion in the network?

Recent proposed solutions haven't quite completely addressed all the requisites discussed in the last paragraph; such is the case of Hyrax [15]. Their solution has two limitations. The first limitation is that data are not stored close to end users. The second limitation is due to the fact that extra copies of data are stored on servers to enhance data availability, overloading those servers.

In [16], the mobile device only acts as a consumer of the storage system, as data get stored on the users' personal devices. No efforts have been made in keeping data close to the end user. In addition, this proposal has not addressed the requisite of data availability.

Work done in [17] shows a cloud storage service implementation that has mobile devices in vicinity, pool their available storage resources to provide for the cloud service. This proposal is

very local, as nodes are connected in ad hoc mode. The current work has some similarities, but that is more flexible and generic than [17].

According to the author's best current knowledge of the literature, no previous work fulfills adequately all the challenges that the current work aims to address. This work aims to contribute to the area of mobile cloud computing, in particular, fog storage, by designing and implementing a data sharing system, that uses mobile devices as both the consumers and providers of cloud storage services. Our proposal includes the use of two node types: mobile nodes and fog nodes, enabling a wide-spread solution. A novel resource allocation manager has been implemented that uses networking nodes' energy level and position information. Additionally, the resource manager includes a replica management module based on data popularity, that uses both the mentioned metrics. The comprehensive evaluation of our fog proposal has evidenced that it is responsive, offloads traffic from the backhaul links due to its resource allocation design, and enables a fair distribution of energy consumption by effectively distributing content.

The current work has the following structure:

- Chapter 2 presents a review of the state-of-the-art. It's divided into four sections. The first section studies different mobile cloud architectures regarding how the cloud resources are provided to mobile devices. The second section studies data sharing systems. The third section studies data placement schemes, which refer to the resource allocation mechanism used in the data sharing system. The fourth section studies network coordinate systems, which determine the networking nodes' positions, to then be used in the data placement scheme.

- Chapter 3 describes our solution proposal for the current research question. It is divided into three sections. They are: architecture and infrastructure; data sharing system; and data placement scheme.

- Chapter 4 presents the steps taken in implementing the proposed solution. It is divided into four main parts. These include installing the data sharing system on the proposed infrastructure nodes; implementing the network coordinate system on the nodes; integrating the battery level and position metrics in the system to be used by the data sharing systems' data placement scheme; and implementing the data placement scheme.

- Chapter 5 presents a series of functional and performance tests, with several scenarios, that evaluate the system's design and implementation. A scenario has been created that

tests the network coordinates system implementation. Three other scenarios have also been created, which assess the data placement scheme. The evaluation results show that the system: reduces the latency experienced by the users when they try to access the data; diminishes the bandwidth utilization at the more constrained backhaul links; shows a fair consumption of energy when distributing content in the system.

- Chapter 6 concludes our contribution and points out some relevant future work directions aligned with the current proposal.

1.3 Research Method

The current work has been developed based on the design science research methodology process. The different phases of research have been the following ones:

Identifying a problem – Mobile cloud computing solutions haven't yet tried to take full advantage on the considerable amount of resources available on the smartphones.

Defining a goal - Implementing a cloud storage system that has mobile devices act as both provider and consumer of the cloud service, while using their "energy level", and networking position, to guarantee data availability, data proximity to end users and fair distribution in energy consumption.

Developing a design – An architecture that supports the content management needs of our goal must be developed. Also, the architecture should allow for content replication management according to the networking mobile devices' "energy level" and position.

Implementing the design – A prototype shall be implemented on a testbed with mobile nodes.

Testing the design – To reach the final prototype, the previous system design shall be tested to find its optimal configuration.

Evaluation – Final prototype shall be evaluated.

The last three phases have been repeated in a cyclical way.

Chapter 2. State-of-the-Art

This chapter starts by studying different mobile cloud architectures regarding how the cloud service provides resources to mobile device users. The current work's goal is to have mobile devices participate more actively for providing services of cloud storage. In this way, the devices should also store the clouds' data. In order to manage the data in the cloud, a few data-sharing systems have been reviewed. We look for architectural and flexibility advantages that can help the system's performance in very demanding mobile cloud scenarios. We plan to overcome the mobile cloud issues by deploying a data placement scheme, that should take into account several aspects, namely nodes' battery levels and position. In this way, different data placement schemes have been reviewed that aim for energy efficiency, system responsiveness (in accessing data), and overall system efficiency. As mentioned, the data sharing system should be location aware, in order to keep data close to end-users. This chapter finishes off by reviewing work done in network coordinate systems. These systems map the network topology, which can confer location awareness to the data sharing system, thus enabling the data placement scheme to use this information.

2.1 Mobile Cloud Architectures

Mobile cloud computing is a computing paradigm where mobile devices satisfy their resource deficit by using resources from a cloud service. At first, the cloud resources were provided by the traditional remote cloud. With the development of technology, faster response times became a necessity for some mobile applications [6]. Consequently, mobile cloud computing has been extended to include the edge computing paradigm. The idea behind edge computing is to alleviate the long wide area network (WAN) latencies associated with retrieving resources from the remote cloud. Edge computing achieves this by enabling data storage in nodes closer to the final consumers [8]. Within the edge computing paradigm, we find three different implementations. They are: fog computing, multi-access edge computing (MEC), and cloudlets. Lately, the mobile cloud computing paradigm has extended itself further with the creation of 'mobile clouds'. The resources that these clouds offer are pooled together among mobile devices.

We follow-up by studying each of the previously referred architectures for providing cloud resources to mobile devices. Figure 1 shows a high-level depiction of the different architectures on how cloud service resources are provided to mobile devices. The current work focuses on providing storage resources as a cloud service.

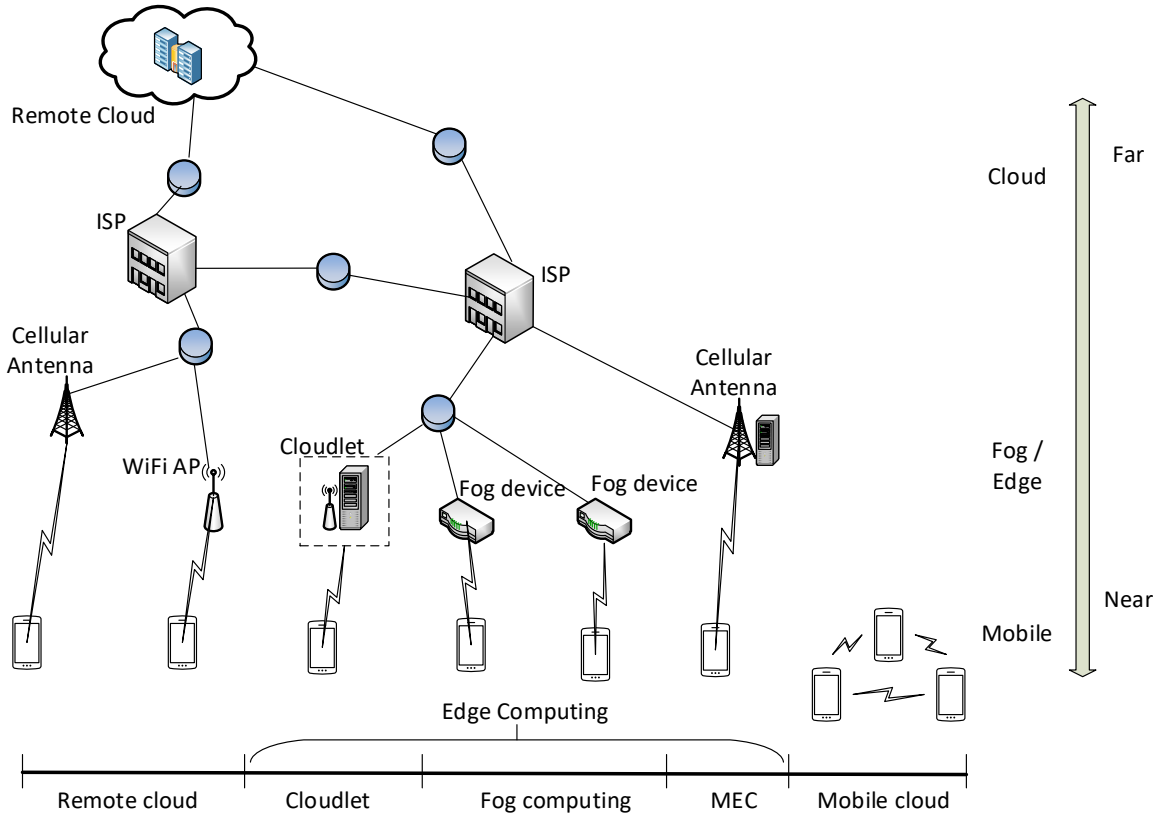


Figure 1 - Cloud architectures providing services for mobile devices [2]

Each architecture in Figure 1 reflects a different way of providing cloud resources to the mobile devices. The diverse architectures shown in Figure 1 reflect distinct characteristics in retrieving the resources. We have categorized these architectures by comparing the distance from the resource provider to the end-users' mobile device. In a later stage, a review is done regarding which architectures are more suitable to support mobile devices offering their resources for the cloud service. The following sections compare these diverse architectures and present some relevant work that has been done.

2.1.1 Remote Cloud

In this architecture, mobile devices communicate directly with the traditional resource-rich remote cloud. The connection is usually made via Wi-Fi through access points (APs) or 3G/4G radio access networks as can be seen in Figure 1. These clouds can provide unlimited resources for mobile applications. This is done by offloading parts of the mobile device's workload or data to the cloud. Long latency, due to WAN delays, are experienced in retrieving resources to users, considering the distance between users and the cloud.

2.1.2 Edge Computing

As discussed, there are three different implementations of edge computing. They're all similar in purpose – bringing computing resources closer to the end-users to overcome the limitations

of remote cloud computing [2]. Doing so, alleviates long WAN latencies, thus offering a more responsive cloud service. This is accomplished by putting intermediate nodes between the cloud and the end user, which are responsible for fulfilling the end users' resource requests. We follow up detailing how each of the three implementations work, and at the end a comparison is made among them.

Cloudlet

The cloudlet is known as a 'data center in a box' (see Figure 1). Resource-rich servers connect to mobile devices via Wi-Fi, and have a good connection to the Internet. Latency between the mobile devices and the cloudlet are very low due to the one hop wireless local area network (WLAN) [10]. These resource-rich cloudlets provide the cloud resources, and are much closer-by than the remote cloud. The downside to this implementation is that the resources are only made available to mobile devices that are in a one hop network distance to the resource-rich cloudlet infrastructure. So, the access to the resources is very local.

In [18], the authors have implemented a cloudlet and show that it reduces the power consumption of mobile devices as well as communication latency, when compared to accessing the remote cloud.

Fog Computing

This term was coined by Cisco Systems [12]. Their rationale for coining this term is that a fog is nothing more than a cloud that is closer to the ground. Fog computing's main feature is that the fog system is deployed close to end users in a widely distributed manner, in the form of fog nodes [6]. These are located between the cloud and the end devices and can be placed on a variety of network elements, such as routers, AP's, laptop, computers and IoT gateways. The idea is for a data producer to place data in the fog, and the system should store or process the data as locally as possible, considering the networking fog nodes' capabilities. The nodes are limited to a set of functions related to resource service management. Resource allocation management is also important, because different types of fog nodes have different resources to offer. Therefore, appropriate techniques should be developed in deciding which fog node should handle a specific end node's request. This can be achieved by identifying important metrics and using them as parameters in the resource allocation - context awareness [9].

A distributed storage service has been developed in [16] that allows end users on mobile devices to store their data on their personal devices, which act like the fog. It includes a data indexing system that is stored on a remote cloud for data availability. Data are retrieved directly from the users' personal devices (the fog). A resource allocation algorithm has been implemented that

uses fog node's available storage capacity. A prototype has been implemented in Android. Main concerns with this implementation is assuring data availability, because a user may power on and off his devices at different times. Additionally, there haven't been any efforts made in keeping data close to the end users' terminals. This aspect should be addressed by our current work.

In [19], another fog computing solution has been proposed, more focused on Internet of Thing (IoT) computation. IoT sensors provide data and send it to the fog to be computed. The fog has two layers: a fog layer and an edge layer. The fog layer is formed by routing nodes such as routers and switches and the edge layer is formed by resource-rich nodes such as laptops, desktops, tablets, etc. At least one of the node types is meant to be in a one hop distance from the data providers. Fog nodes typically offload data to capable edge nodes. They do this by running a resource allocation algorithm locally (although an updated snapshot of the system's state is needed), which works on both node types. The algorithm uses information on networking cost and computational cost. This has been implemented in a simulator using Python. As such, the decentralized use of the resource allocation algorithm, that took into consideration other nodes' information, doesn't include collecting the metrics. Using the terms that have been used in [19], our system could eventually have mobile devices producing data, similarly to the IoT sensors. The mobile nodes would also work as edge nodes (resource providers). In this way, a mechanism would have to be implemented for fog nodes to determine information on other fog and edge nodes. This information obtained would then be used to feed the resource allocation algorithm on the fog node, thus selecting where to store the data.

Multi-Access Edge Computing

MEC is another implementation of edge computing, used when the Internet is accessed via mobile network. In MEC, servers are placed at the RAN base stations (BS), and are responsible for storing and computing users' data [20]. The European Telecommunication Standards Institute (ETSI) is currently working on a standard called Multi-Access Edge Computing [21] that facilitates interoperability advancements in MEC deployment [22]. This will allow users to develop their applications by using MEC, thus benefiting from its use.

The MEC servers are context aware, as they manage information on end devices, such as their location and network information. The MEC server's capacity is limited, therefor deciding which data should be stored on the MEC servers is a very important feature to be supported by this system. This data selection technique refers to edge caching (EC) [23]. Some edge caching techniques are studied in section 2.3.

Comparison Among Distinct Edge Computing Implementations

Table 1 compares the different edge computing implementations that have been discussed.

Table 1 - Comparison of edge computing implementations in relation to end user [8][3] [11]

	Cloudlet	**Fog Computing**	**MEC**
Node devices	Servers	Router, AP, etc	Servers running at RAN
Node locations	Local/outdoor installation	Between end device and cloud	BS
Physical proximity	High	High	Low
Logical proximity	Ensured	Not Ensured	Not Ensured
Latency	Low	Low	Medium
Power consumed	Low	Medium	High
Context awareness	Low	Medium	High
Access mechanism	Wi-Fi	Wi-Fi, bluetooth, mobile networks	Mobile networks
Internode communication	Not supported	Supported (P2P)	Not supported
Access	Local	Open	Open
Proposed by	Prof. Satyanarayanan	CISCO	ETSI
Ownership	Local business	Decentralized fog node owners	Mobile operators

Table 1 shows the major differences among the previously discussed edge computing implementations. Cloudlets provide their resources through resource-rich servers with good connections to remote clouds, and one hop connections to mobile end users. The physical and logical proximity to end users is high, and since the end users are connected via Wi-Fi, the power consumption is low [18]. Context awareness is not an issue, since nodes get their resources from the close-by server. Cloudlets can be deployed by anybody.

Fog computing relies on fog nodes that are located at users' premises, to deal with end users' resource requests. Fog nodes are widely spread, so physical proximity is high. Logical proximity is determined by the number of hops between the end device and the resource provider. In fog computing, it isn't guaranteed that the closest fog node to the end-user is resourceful enough to run the fog framework, making the closest fog node multiple hops away. Thus, logical proximity is not ensured. Since fog nodes fulfil resource requests, this implementation supports internode communication in P2P. Fog computing uses resource allocation strategies to decide which node should fulfil a resource request. These strategies use context awareness, that can be used to collect different metrics considered important [9]. Fog computing solutions can be deployed by decentralized fog node owners.

MEC implementation shows the lowest physical proximity to end users, as the resources are provided by servers at RAN BS's, and thus internode communication is not supported. Power consumption is the highest due to resources being accessed via mobile network. The RAN keeps information on the connecting nodes' location, so context awareness is high. Although MEC's infrastructures belong to the mobile operators, open standards are being worked on to allow for deployment of applications using MEC infrastructures [21].

Comparing the three implementations regarding their accessibility: cloudlets are very local. Fog computing and MEC are the only open networks. Meaning a user can access the cloud service from anywhere, granted that it has an internet connection. As mentioned, the current work aims for an open network. Regarding implementation possibilities: MEC computing is not yet standardized, so developing a MEC solution is not yet an option. In addition, fog computing is the only solution that supports internode communication, which is a requirement in the current work. Additionally, it supports context awareness. Out of the three implementations, fog computing is the most compatible with our approach.

2.1.3 Mobile Clouds

These refer to mobile cloud computing architectures where mobile devices use their extra resources in a coordinated manner, to support the cloud service. This contrasts with the previously studied edge implementations, where the mobile device's role in the cloud was that of solely a consumer. Specifically, in the storage resource domain, content distribution happens between the mobile nodes, in P2P fashion. In this type of architecture, resource allocation management, or control management, can be centralized or decentralized. The following sub sections discuss the two types of management control architectures and some relevant work that has been done. At the end, Table 2 is presented, summarising the characteristics of the proposals described below.

2.1.3.1 Centralized Control

In these networks, a central node is used to convey tasks to the rest of the nodes in the network. It keeps track of the metadata of the files, and lets user nodes know which node to write or read from; they do not provide their resources to carry out cloud service requests. This central machine can either be located close to the mobile devices [24], or far away from them [15]. Either way, access to the central node must be maintained by all the nodes in the network.

In the following paragraphs, relevant work regarding mobile P2P centralized controlled systems are presented. The work we have reviewed is Hyrax [15], Market Oriented Mobile Cloud

Computing (MOMCC) [25], Cloudlet Aided Cooperative Terminals Service Environment (CACTSE) [26] and FemtoClouds [27].

Some differences between the work presented, are the type of resources being pooled, be it computing power or storage. Most of the works reviewed, have not only mobile devices, but also resource-rich servers acting as resource providers.

Hyrax

Marinelli [15] presented a P2P centralized approach, "Hyrax" – a platform based on the Hadoop Apache framework, which has been ported to Android, enabling cloud computing support on mobile devices. Hadoop is an open source software used for distributed computing, that can be used to query a large set of data and get the results faster using a reliable and scalable architecture. There are two main components in Hadoop. They are the Hadoop Distributed File System (HDFS), which takes care of the data storage, and the MapReduce framework, that takes care of the parallel distributed processing needs of the users. They've ported Hadoop to Android, allowing for storage and computational resources to be shared amongst mobile devices. The current work focuses on sharing storage resources, so further on, in section 2.2.1, HDFS is detailed.

The authors of [15] have implemented the Hadoop processing and storage components as can be seen in Figure 2.

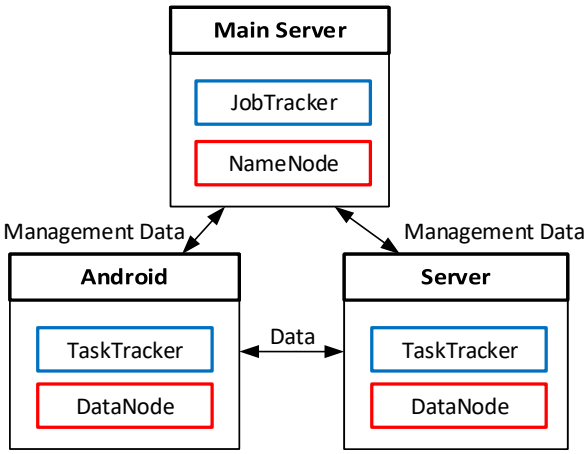


Figure 2 - Cluster components and infrastructure used in Hyrax

In Figure 2, the top box on each node represents the processing related component, and the bottom box represents the storage related component. In Hyrax, they use a fixed main server to run the centralized management components – jobtracker and namenode; and run the "working" storage and processing components on Android devices and servers.

Note that every working/storing node must have access to the main server.

Hyrax uses an adapted version of HDFS' default location awareness, which is named rack awareness, to suit the mobile-cloud scenario. As mentioned ahead in section 2.2.1, the HDFS arranges nodes into racks. Figure 3 shows Hyrax's location awareness implementation.

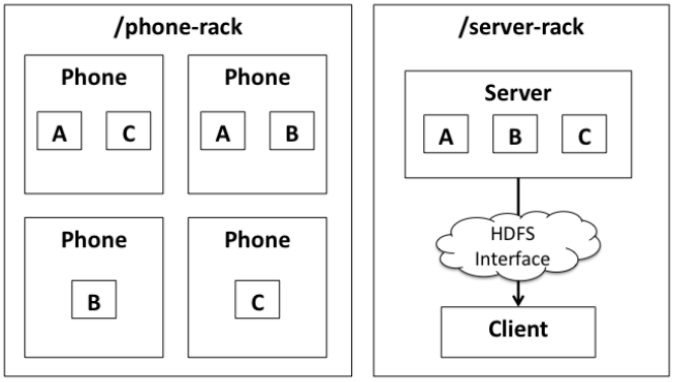


Figure 3 - Overview of location awareness in Hyrax

The authors of [15] tweak the rack awareness map to work with the HDFS' default replication algorithm. Phones get logically placed on one rack, while servers are placed on another. In Hyrax, the default number of replicas in the system for each block is one. According to the default placement strategy (or replication algorithm), when a phone user replicates content, it first replicates to a server on the server rack, and then replicates it to another server on the same rack (in case the number of replicas is three). This solution provides content availability, as data get stored on diverse servers, creating a more robust system against failures.

The authors of [15] have not addressed mobile devices' battery levels or replica number management in their system's architecture. In Hyrax, the user is who initiates the replication of a file. This can eventually lead to abuse usage of network resources. In addition, data proximity to the end user has not been considered in Hyrax.

MOMCC

In MOMCC [25] a mobile device hosting service platform is proposed, where processing is done by other mobile devices, coordinated by a trusted party-central node. Figure 4 shows MOMCC's architecture.

The system requires a central supervisory entity which acts as a trusted party (governor), a service programmer, a mobile host, and mobile node requesters. Service programmers develop services that are uploaded to a central database to be used by mobile clients. Mobile users willing to share their smartphone's resources with other mobile nodes can register at the governor. Mobile users who want to host services, have to be authenticated and authorized by the governor in order to become service hosts.

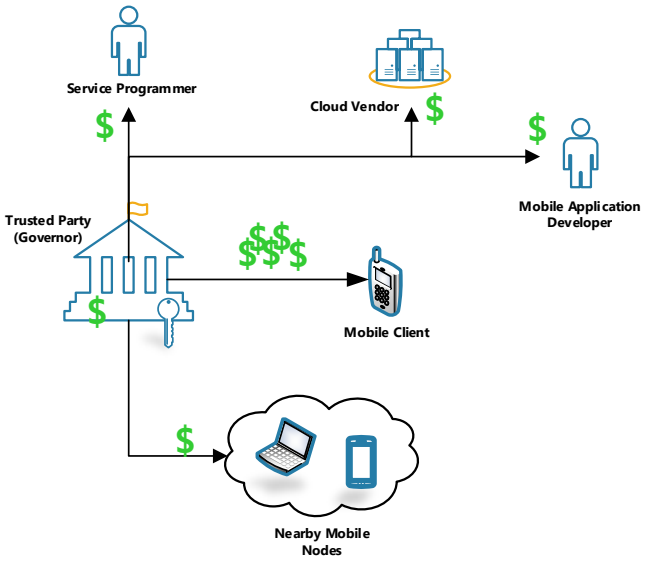


Figure 4 - MOMCC architecture

The governor finds the mobile nodes close to the service host in order to help with the necessary processing. As a larger number of users offer to cooperate, the more processing power is available, and thus, the bigger the content distribution potential of the system. The accessibility to distant stationary clouds is provisioned in case the nearby smartphone resources are not sufficient. The necessity for fine-grained services in this proposal is problematic. Participating nodes position, battery levels and storage levels are concerns of the service developer. This proposal hasn't been implemented in a real testbed.

CACTSE

Qing et al [26] have proposed a mobile cloud computing architecture for content delivery at the edge of the mobile Internet. Figure 5 illustrates how the system works.

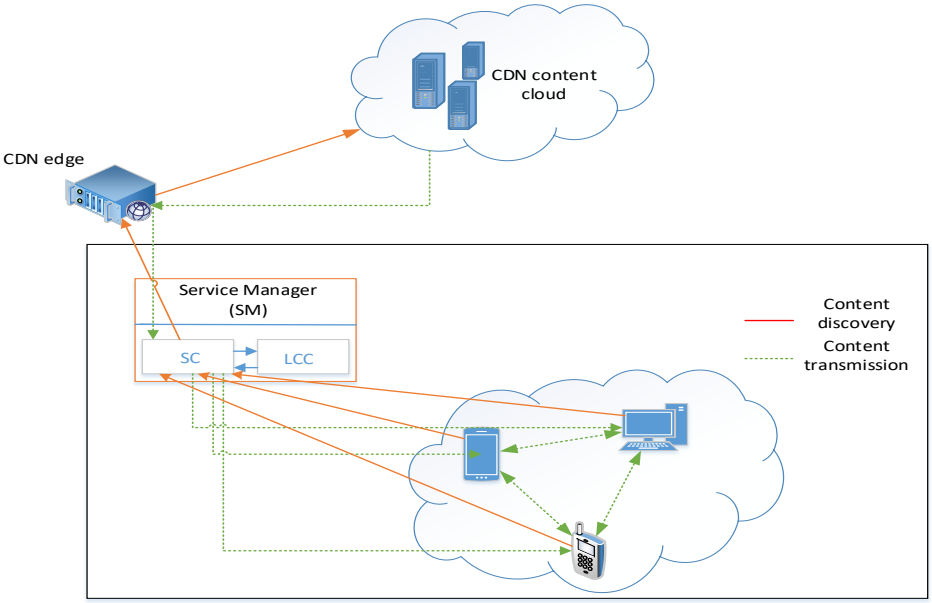


Figure 5 - CACTSE architecture

CACTSE relies on a Service Manager (SM) node to take care of coordinating the local network. The SM node has two components: Service Coordinator (SC), which manages the network content flow; and the Local Content Cache (LCC), which manages the location of the content in the cloud. Content is exchanged through a device-to-device (D2D) mechanism, with SM coordination. Data availability is assured by having the SM retrieve the data that is not stored locally, from a remote cloud. Support for node mobility is not considered, as such, updates to the LCC content table are not performed. In a highly mobile environment, where nodes enter and leave the network, this can bring very high overhead to the system.

CACTSE works as a hybrid between cloudlet and mobile cloud implementations, as both rely on a fixed machine placed between end users and the cloud, to fulfil the users' resource requests. What differs, is that in CACTSE, the fixed machine is responsible for managing the cloud storage system, which contrasts with using its own resources to fulfil requests. Content replication strategies haven't been addressed in CACTSE. It has also not been implemented.

FemtoClouds

The authors of [27], have proposed a system that has nearby mobile devices sharing their available computational resources. The resource allocation is managed by a central controller node and uses networking node's metrics, such as: available computational capacity and bandwidth estimation to other mobile nodes. Their work has been implemented in Android. Node's energy concerns have been partially addressed by configuring a minimum battery level requirement for nodes to remain in the cloud. Their task assignment algorithm can be further improved if it will consider mobile device's energy consumption. This proposal considers small networks, which differs from the goal in the current work.

2.1.3.2 Decentralized Control

In these architectures, nodes keep track of their own resources and periodically notify other nodes in the cloud of their available resources and current situation in regard to job execution. The work reviewed was proposed by, or is named: Lacuesta et al [28], Huerta-Canepa and Lee [29], Phoenix [30], Eager replication extended Database State Machine (E-DRM) [31], Energy Efficient Content Retrieval Scheme (EECRS) [32], Monteiro et al [17] and Pätris Halapuu [33].

Proposal of Lacuesta et al [28]

A spontaneous ad hoc mobile cloud computing network has been proposed. This network is formed by independent mobile users at a certain place and time. Users are free to enter and leave the network. They have developed algorithms for managing the nodes that join and leave the network. Users can then choose which network nodes to send their storage/processing needs

to. Each requested node may then accept or decline the invitation. Mobile devices' energy level has not been directly accounted for in their work. They have tested their proposal by running simulations on Castalia 2, a wireless sensor network simulator based on the OMNeT++ 3 platform [34].

Proposal of Huerta-Canepa and Lee [29]

Guidelines for a framework to create a virtual mobile cloud computing provider have been worked on. It mimics a traditional cloud provider by using mobile devices close to each other. Mobile devices in this system, when considered stable, are chosen to offload the applications to, thus creating a virtual provider for cloud services among mobile devices. The mobile devices in this system are connected by an ad hoc network, and cooperate amongst each other by a win-win incentive of a common processing load. They have implemented a prototype based on Hadoop, where they swap the file system for direct downloads from the source mobile. The Map Reduce framework was replaced by RPC methods with Jabber RPC extension. Devices' energy levels haven't been considered in this work.

Phoenix [30]

A system for providing ephemeral transient storage service by using a self-organized one hop network of mobile devices has been proposed. A data distributing protocol called Phoenix has been developed, that ensures data availability in the cloud. The protocol's main goal is to maintain K copies of every block in the network. They've implemented Phoenix in TinyOS and have evaluated its performance with TelosB sensor devices.

E-DRM [31]

This work proposes a data sharing system for mobile devices. The main goals are maintaining content availability and consistency through replication and synchronized broadcasts. They use atomic broadcasts to ensure synchronization. E-DRM considers two types of mobile nodes: Small Mobile Hosts (SMH) and Large Mobile Hosts (LMH). SMH's have limited resources and as such have only a part of the database to accomplish query tasks. LMH's are rich in resource capacity and as such have complete databases. They are responsible for broadcasting, certifying and updating tasks. The system is optimized to send a low number of broadcasts, which saves the battery level of mobile devices. However, the design of this proposal does not account for nodes' battery levels directly.

EECRS [32]

This work has proposed a data sharing system that addresses scalability and energy efficiency. It uses Information-Centric Network (ICN) [35] for content search and retrieval. In these

networks, users only need to know the name of the content they are looking for. Nodes use Global Position System (GPS) to identify their network coordinates. Users send out broadcasts, indicating their interest in a given content file. Energy consumption is saved by using a direction-selective forwarding scheme for the content requesting phase, which eliminates duplicate request packets, diminishing traffic load. Testing proves the concept to not be "applicable" in real case scenarios, as the system needs a high number of replicas to guarantee an efficient content retrieval.

Proposal of Monteiro et al [17]

A decentralized storage system created by nearby mobile devices has been developed. It relies on a distributed hash Table (DHT) structure to know where data is stored. Each node has a partial knowledge of the network, and knows where the remaining content is placed. A prototype in Android has been implemented, using TomP2P – a java implementation of DHT. Data availability has been addressed by making each node responsible for maintaining a target number of replicas, for a specific file, in the system. A replication strategy based on popularity has been implemented. The system addresses energy concerns by having an Energy Manager on every node, that turns the node invisible to the network when its energy level goes below a configurable threshold.

In the above-mentioned work, the P2P communicating nodes are all in a local ad hoc network. The work that follows relies on a more open network protocol, which allows for nodes to be far away from each other, while maintaining a decentralized management approach.

Proposal of Pätris Halapuu [33]

A distributed file system that has been developed by Pätris Halapuu [33]; it has mobile devices work in P2P fashion, by porting the libtorrent library to android. The work involves service discovery between nodes in P2P and WAN domains and developing a BitTorrent client that discovers and publishes .torrent files. This approach hasn't considered node proximity or their energy levels, in deciding where the torrent files get published. The author of [33] argues that content transmission consumption is distributed, since content is retrieved in parallel. Although the energy consumption is more distributed, which makes the system fairer; this also drains more energy in retrieving content, making it less energy efficient.

Having studied a considerable amount of different mobile cloud approaches, with centralized and decentralized management, we follow up by showing a comparison between these P2P mobile cloud approaches. A comparison among the several proposals is available in Table 2.

Table 2 - Comparison between centralized and decentralized mobile cloud approaches

	Phoenix	Proposal of Pätris	Proposal of Lacuesta	E-DRM	EECRS	MOMCC	Hyrax	CACTSE	Proposal of Huerta	Proposal of Monteiro	FemtoCells
Network Management	D	D	D	D	D	C	C	C	D	D	C
Network Size	S	B	S	S	S	B	B	S	S	S	S
Data Proximity	H	L	H	H	H	L	L	H	H	H	H
Management Latency	L	L	L	L	L	H	H	L	L	L	L
Resource Retrieval Latency	L	H	L	L	L	H	H	L	L	L	L
Context Awareness	L	L	L	L	H	M	M	L	L	L	H
Resource Type	S	S	P/S	S	S	P/S	P/S	S	P	S	P
Implementation	TO	BT	O	N	TB	N	Had	N	Had	TP2P	And

Legend:			
	D – Decentralized	L – Low	TB – Testbed
	C – Centralized	P/S – Processing/Storage	Had - Hadoop
	S – Small	TO – TinyOS simulator	TP2P – TomP2P
	B – Big	O – OMNet simulador	And – Android
	H – High	N – No	Bit - BitTorrent

We will now compare the decentralized management approaches with the centralized management approaches and try to unravel the best course of action for the current work. Regarding context awareness: in the decentralized approaches reviewed, the majority did not show context awareness support in allocating resources within the network. The exceptions being EECRS, which uses location awareness via GPS, and Monteiro et al's proposal that runs an agent that monitors the node's energy levels, turning it invisible to the network when under a configured threshold. This approach uses nodes' energy levels in deciding where not to store data; we want to use it to decide where to store data. The centralized approaches, also do not present work done in this regard. Except for FemtoCells, that collects available computational capacity and bandwidth estimation on its networking nodes. This solution focuses on computational resources though. This analysis leaves for now, our options open, regarding the choice of a control management architecture to be used in the current work.

However, looking at Table 2, it can be seen that decentralized network management solutions tend to manage small networks, with the exception of Patris Halapu's proposal. In opposition, the centralized managed systems, are more versatile. The central node can be close, or far away

from the rest of the nodes, which also reflects on the networks' size. To take full advantage of the fact that there are a lot of smartphones attached to the network, we envision a system for the current work, that works on a big (open) network. This shifts our decision on an architecture for control management in the cloud, to a centralized approach. The analysis of Table 2 continues in the following text.

Focusing in on approaches with big network, Table 2 shows that these, tend to have lower data proximity. This is because the systems are not context aware. Looking back at the fog computing architecture, its goal is to have nodes close by (in the fog) cooperating with other resource-constrained devices (e.g. IoT sensors). They also offer the opportunity of working on an open network. This is where the current work fits in making a contribution to the mobile cloud computing paradigm. Developing a widely accessible storage system, where data are placed on other mobile devices, close to the end user, by making the system location aware. In addition, the system should also be energy aware, and use this awareness in deciding where to place or retrieve data within the network. This can be seen as a fog computing implementation, where mobile devices are both at the edge (requesting to store data) and in the fog (storing data); or a mobile cloud centralized approach, that uses location awareness to keep data close to the end users while taking into account their energy level.

2.2 Data Sharing Systems

This section reviews data sharing systems that can be implemented having in mind our P2P mobile cloud approach. The main choices here are to either create a customized data sharing system from scratch which can either have centralized [26], [25] or decentralized management [28], [10], [30], [31], [32], [33], or to implement a distributed file system (DFS) which already comes with the underlying cloud storage mechanics such as read and write operations [15].

A DFS is a file system that allows clients to access files from multiple machines via the network. We will be focusing on mechanisms that DFSs have that can counteract the mobile cloud computing common issues.

Common issues in mobile cloud computing [3] derive from mobile devices' energy constraints and limited storage. We expect to counteract these issues by having the data sharing system use energy level and position metrics in deciding where to store and retrieve data from. Therefore, a DFS that supports the needs in the current work, should be flexible in terms of adding custom metrics to be used in its data placement scheme.

We will now have a look at some important characteristics that data sharing systems should fulfil, and how DFSs achieve these characteristics by resorting to different mechanisms. The characteristics are transparency, fault tolerance, and scalability. We explain each one in the following text.

The **Transparency** in a filesystem means that users are oblivious to how the internals of the system works. They just store and retrieve data from the system.

The **Fault tolerance** refers to the system's robustness against important failures. For example, network and server issues could originate situations of data being inaccessible. Fault tolerance mechanisms try to overcome these flaws. To improve fault tolerance in a system, data can be replicated and distributed amongst the nodes according to a data placement strategy, which incorporates a content replication algorithm [15],[30].

The **Scalability** in a filesystem means that a high number of nodes incorporated in that system should not be a problem for its performance. DFSs are designed having in mind containing large amounts of data, servers and clients.

These are characteristics all DFSs should have and that are employed in different ways. The different architectures in how DFSs can be implemented, are as follows:

- **Client-Server Model**, where metadata and data are stored on a central server. Users access this server to retrieve data.
- **Cluster-based Distributed Model**, where metadata and data are decoupled. Data are stored on several servers. Metadata can be stored on one metadata server, making it a **centralized** system; or several metadata servers, making it a **decentralized** system**.** In both approaches, clients access metadata servers to find out which server has the data they are requesting, or which server they should write their data to.

In this thesis, mobile devices should pool their extra storage resources together to provide for a distributed storage network. This scenario calls for a DFS with a cluster-based distributed architecture, where a DFS client runs on the data-holding mobile devices, storing the clouds' data, and lets users retrieve data from the cloud (from other mobile users). The security aspects of this proposal are out of the scope of the current work (e.g. data privacy). What may differ is the management aspect of this architecture: be it centralized or decentralized. We now continue our discussion by studying a few cluster-based distributed DFSs [14].

2.2.1 HDFS

HDFS is the file system used by Hadoop, it is open source and implemented in java.

Architecture - It uses a cluster-based architecture with a centralized management, where metadata are managed by a central node called Namenode. Data are split into blocks and distributed in a redundant way throughout server nodes (Datanodes). The namenode is responsible for managing the content flow within the cluster. Figure 6 shows how the HDFS components interact.

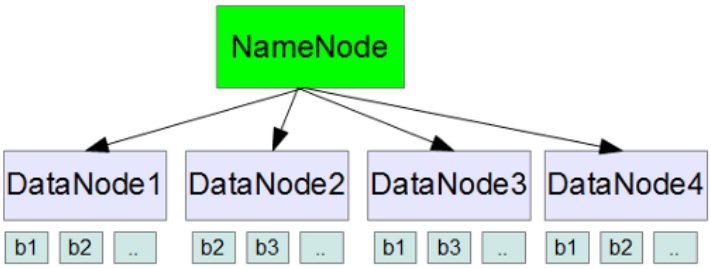


Figure 6 - High-level architecture schematic of HDFS

The namenode may be a single point of failure (SPOF). But HDFS presents additional features that help diminish the chance of cluster failure. It allows for a second namenode to take over the management of the cluster, in case the main one fails. This doesn't work as a failover mechanism. Although, HDFS does support a feature which enables a failover mechanism on the namenode. HDFS also supports larger deployments formed by several namenodes operating as a federation system, aggregating several clusters.

API and client access - HDFS provides an API in java and an HTTP REST API called WebHDFS to allow users to read, write, and delete files or directories. Users access a file by referring to its path in the namespace and contact the namenode to know where the blocks of this file have been distributed to. Data are then retrieved by accessing the datanode directly and requesting for the blocks stored on that datanode that contain the data being searched.

Replication - HDFS allows for setting a target number of replicas for every block in the file system; the default number is three. HDFS splits data into blocks and distributes them across the datanodes according to a replication strategy, configured in the namenode. The namenode then ensures that each block has the intended number of replicas in the system. If a block is over-replicated, or under-replicated, the namenode will accordingly, delete or add a replica to the cluster.

We will now have a look at how the operations of writing and reading files work in HDFS. In Figure 7 we can visualize how the write operation is carried out.

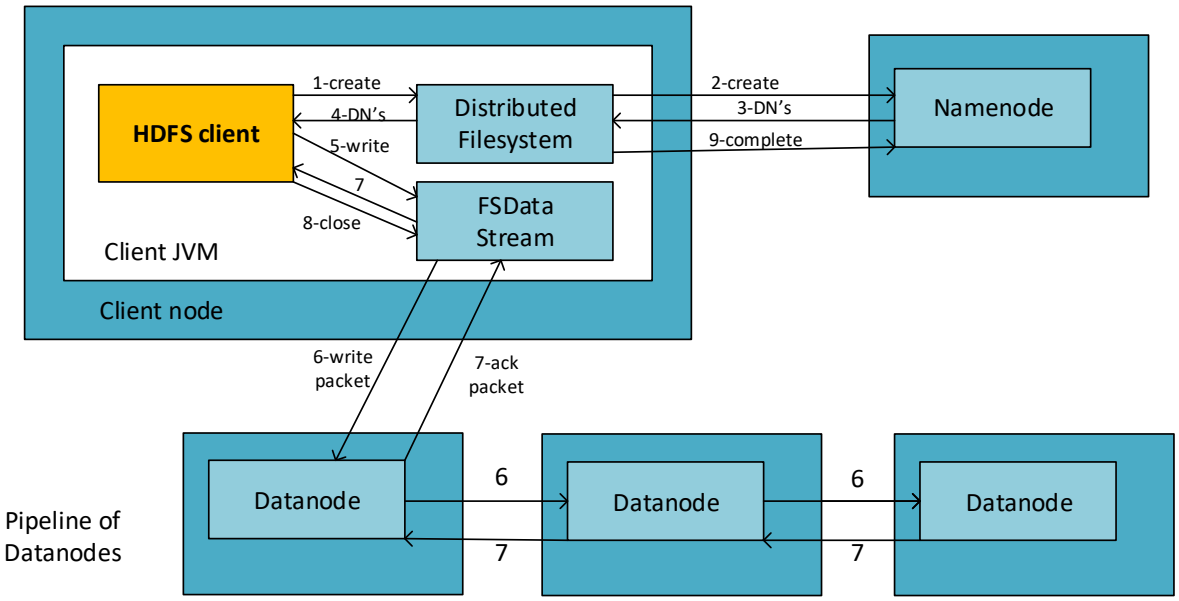


Figure 7 - HDFS write operation [36]

The process seen in Figure 7 is triggered when a user tries to add a file to the file system. The user interfaces through the HDFS client that he requires to write to a file. The datanode on the client contacts the namenode and informs the latter on the size of the file. The namenode splits the file into blocks of a predetermined size, and then checks the **replication strategy**, to determine which datanodes should store the blocks.

HDFS implements location awareness with the help of Hadoop Rack Awareness. The cluster administrator is in charge of deciding which rack each datanode belongs to. The default replication strategy uses this rack awareness to select where blocks are stored. It dictates that the first replica is placed on the file-writing node, the second replica is placed on a remote rack, and the third replica is placed on a different node on the remote rack. In our system, we have changed the context awareness model to fit our mobile node needs (by changing the location awareness model HDFS uses and adding energy awareness) and also adapted the replication strategy to use the new context awareness model.

The namenode then answers with a list of the chosen datanodes (list's size is equal to number of default replicas). In Figure 7 we can see that after the datanodes have been selected, the file gets written in a pipeline throughout these nodes. This detail is interesting in the mobile cloud scenario, as a mobile device writing a file would only have to write to one node. This way, the mobile devices' energy is saved, and overall, this increases the system's lifetime performance.

In order to minimize both bandwidth consumption and read latency, the namenode tries to satisfy read requests by indicating nodes with requested replicas' closest to the reader. Replicas

on the same rack are preferred to satisfy read requests. This feature is something we want to keep in our system, but it would have to be adapted to fit our context awareness model.

Load Balancing - HDFS decides if a cluster is balanced by checking every datanode and verifying if its' usage is different from the usage of the cluster by no more than a threshold value. If a node is unbalanced, replicas are moved from that node to another, according to the replication strategy.

In our scenario, load balancing could be triggered to run periodically, and move data around the cluster, with the intent of keeping data close to the mobile end user and enhancing the data's availability.

Features - Hadoop supports the metrics2 framework which is designed to collect and dispatch Hadoop component metrics to monitor the overall status of the Hadoop system [37]. This enables collecting cluster nodes' metrics, so that they can then be visualized on a server machine. The metrics2 design is depicted in Figure 8.

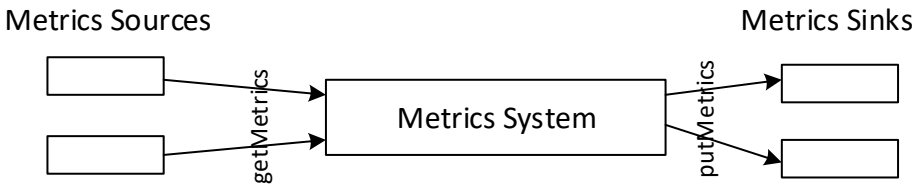


Figure 8 - Metrics2 framework [37]

The metrics system is configured to retrieve metrics from the metric sources, and forward them to the metric sinks. The metric sources retrieve their metrics from the Hadoop components.

This could be used as a way of retrieving metrics from datanodes on mobile devices, and sending them to the namenode, so that they can be used in its replication strategy.

2.2.2 MooseFS

MooseFS is an open source (GPL) DFS developed by Core Technology.

Architecture - Very similar to HDFS. It has a failover mechanism for the metadata server. When a metadata server fails, it switches to the metalogger server, which has a copy of the metadata servers' status.

API and client access - Clients access the file system by mounting the name space (mfs command) in their local file system. User operations are directed to the metadata server which

allow for direct transmission of data between the correct servers. This is supported by the Filesystem in Userspace (FUSE) mechanism.

Replication - As in HDFS, each file has a target number of replicas in the cluster. When a client writes data, the master server sends a list of nodes to the client, telling it where to store the data. The client then sends the data to those nodes.

Load Balancing - Load balancing is provided in the sense that the target number of replicas for a file is assured, by either adding or removing replicas from the cluster when needed.

2.2.3 iRODS

Integrated Rule-Oriented Data System (iRODS) is an open source data management software that is highly customizable.

Architecture - iRODS has a centralized management system. It has two main components: the iCat, which stores the metadata and manages queries to the metadata; and several iRODS servers, which store the data. An iCat server along with several iRODS servers make up a zone. Files of one zone can be reached by other zones, creating a federated system.

API and client access - iRODS provides a client interface in command line, a FUSE module and API (in PHP and Java) to process queries and allow for I/O operations. Clients communicate with iCat to query metadata, and directly with iRODS servers for data transfer.

Replication - iRODS allows for creating rules after certain operations have taken place. For example, every time a file is created it could be replicated to a certain node, according to an implemented replication rule. The data placement strategy is the user's responsibility.

Load balancing - can also be customized. Storing nodes are monitored to measure server activity (CPU load, disk space). A rule can be periodically run to determine if a certain node is being overloaded. It's also possible to add rules to, for example, force data to be placed in a certain storage or swap data between storages.

2.2.4 Ceph

Ceph is yet another open source distributed file system.

Architecture - Ceph has a decentralized management, meaning there are several metadata servers. The system is composed by two main components: the metadata server (MDS), which provides dynamic distributed metadata management; and the Object Storage Devices (OSD), which store data. MDSs manage the namepace, while data are passed along the OSDs.

API and client access - In Ceph, clients themselves can figure out which nodes have the necessary blocks to retrieve the file they have requested. Data are written to OSD nodes according to a function called CRUSH (Controlled, Scalable, Decentralized Placement of Replicated Data) [38]. When a client requests content, it contacts a node in the MDSs cluster, which sends information that with the help of CRUSH, allows for calculating how many blocks comprise the file, and in which OSDs they are stored. Transparency is provided through a REST API or through FUSE mechanism.

Replication - Ceph's replication resorts to a hierarchical cluster map, where the system administrator attributes weights to nodes, according to the amount of data that they can store. The CRUSH function determines the placement strategy according to free disk space and the devices' weights. The replication strategy can be customized, by choosing a number of replicas to be placed on nodes that belong to a certain hierarchy level (which is based on their weights).

Load Balancing - Loading balancing is done at two levels: metadata and data. It has a counter for metadata that allows it to know each files access frequency. So Ceph allows for the replication of 'popular' metadata from a MDS to another.

Table 3 shows a comparison of the studied cluster-based DFS approaches.

Table 3 - Comparison of cluster-based DFS approaches

	HDFS	**MooseFS**	**iRODS**	**Ceph**
Architecture	Centralized	Centralized	Centralized	Decentralized
Written in	Java	C	C++	C++
Access API and Client Access	Java, HTTP	FUSE	FUSE, PHP, Java	Librados (C, C++, Python, Ruby)
Context Awareness	Racks	Racks	Zones	Weights
Replication according to	Placement strategy	Placement strategy	Custom Rules	CRUSH function
Load Balancing Triggers	Node usage compared to cluster usage	Maintaining target number of replicas per block	Custom Rules	Allows to move popular data/metadata

The current work requires a data sharing system that is supported on mobile devices. In addition, the DFS should have a replication algorithm that is flexible to change. In fact, the context awareness should also be flexible, in the sense that new metrics can be added to be accounted for. We essentially look for a DFS that has these traits, or that has been previously modified in other works to achieve similar goals.

Table 3 shows that all the reviewed DFS's are context aware. HDFS and MooseFS rely on the administrator to manually configure which rack nodes belongs to. The replication strategy then chooses where to store data based on this information.

Ceph also relies on the administrator to configure the data nodes' weights', which is then used as the only metric [38] by the CRUSH function, in deciding where to store or retrieve data. This system doesn't seem very flexible in allowing to add custom metrics, to be accounted for, in data placement decisions.

iRODS's context awareness is managed in zones. Each zone has several data servers and one metadata server. The main issue with this implementation, is porting it to work on a mobile device OS. Since it is written in C++, and mobile devices do not support this language. In addition, and more importantly, the code has 250k+ lines. To our best knowledge, no attempts have been made in porting an iRODS server instance to a mobile device.

Looking at Table 3, another hint related to the language they are written in, may help in deciding on a course of action. HDFS is the only DFS developed in java in Table 3. Java is a multi-platform language. One only needs to have a java virtual machine (JVM) installed, and java applications can run. In addition, Android supports java. In Hyrax [15], described in 2.1.3.1, they have managed to port HDFS to Android. HDFS also shows a file write operation mechanism based on pipelines, where a writer node stores a file on one node, and that node stores it on the next. This detail saves battery on mobile devices. In addition, HDFS supports the metric2 framework, which allows for collecting metrics on HDFS components. As such, HDFS presents itself as the best option for us to pursue in the current work.

2.3 Data Replication

Resource allocation management directly influences the lifetime and performance of cloud service systems [7]. In this way, the performance metrics that the current work reaches for are system responsiveness, and fair energy consumption. Before trying to achieve these system qualities, data's availability needs to be addressed. This can be done by maintaining a certain number of replicas for each file in the filesystem. Knowing how many replicas a file should have in the system, as well as deciding where to place them, aligns with our interest in having a responsive and energy efficient system. However, replicating content in a mobile cloud scenario wastes devices' energy, occupies storage, and bandwidth connection. So, it should not be taken lightly.

We follow-up by reviewing work where content popularity acts as a measure for the quantity of replicas that should be in the system. After that, some work has been reviewed that gives insight on where popular content could be placed, as well as some energy-efficient storage systems.

2.3.1 Data Replication according to Popularity

Evaluating content popularity can be done using static models or dynamic models. Static popularity models assume content popularity to be static, which is not true in data sharing systems [39]. Thus, dynamic models have been proposed, which track the changes in content popularity. Such a model, is Shot Noise Model (SNM) [40]. The model is based on video-on-demand traffic, and accounts for temporal locality in requests for content. This means that it manages the past access frequency of content.

Z. Cheng et al [41] have developed ERMS, an Elastic Replication Management System for HDFS. A dynamic popularity estimating module has been added to HDFS. Content is sorted according to its popularity, into hot or cold data. The system dynamically adds or deletes extra replicas depending on the data type. A storage model has been developed, that keeps both, active and standby datanodes in different racks. The replication strategy has been extended place hot data on standby nodes. The hot data can be accessed from the standby nodes when the active nodes are being heavily used. They have used a testbed of eighteen commodity computers working as datanodes. Their experiments showed that ERMS effectively improved the reliability and performances of HDFS.

Adding a replication management module based on file popularity is something that could work well in the current work.

In order to ensure that popular content is distributed to the network edge where it is required, an accurate popularity estimation model is necessary. In [42], the authors argue that the popularity estimation models used in big data suffer from poor accuracy due to the sparse population at the network edge. They have developed an edge popularity prediction model based on social-driven propagation dynamics. This could be used in our work, in identifying the areas demanding for popular data; and replicating the data to those areas accordingly.

In Mostafa Dehgan et al [43], the authors study optimal content placement at the edge. Their proposal has the goal of minimizing average data access delay over the requests of all users for all files. They have achieved this by considering the transmission delay for each link.

2.3.2 Data Replication according to Energy

A way of bringing energy efficiency to a network design is to use edge caching. This approach is being studied in 5G networks in the mobile network domain [20], [44]. The idea is to replicate popular data to the edge of the wireless network. The edge devices do not share the resource constraints that mobile devices suffer from. This technique brings system energy gains and fault tolerance, as well as alleviate long WAN latencies and backhaul offloading [20], [44].

In [23], the authors take another approach in determining caching placement in heterogenous wireless networks. They argue that caching systems are usually designed to maximize hit rate, by caching popular data. They develop a more energy-efficient caching solution, that minimizes two fundamental metrics: expected backhaul rate and energy consumption. Energy consumption is given as a trade-off for energy used to store data at edge devices, and energy used in transporting data over the backhaul link.

Ting Yang et al. [45] have developed an energy-efficient storage strategy for cloud datacenters using HDFS by verifying which data are more often used, and which is less used, and calculating a minimum set of datanodes required to run in the datacenter. This way, nodes that have less accessed data can be turned off, saving energy at the datacentre.

Liao et al. [46] have proposed an energy-efficient algorithm for distributing data in HDFS by reconfiguring the data storage structure with a block storage structure reconfiguration algorithm. They divide racks into two storage areas: active and sleep zones. Nodes in the sleep zone are turned to sleep mode while the workload is low. Their experiments showed the algorithm could improve energy efficiency in the distributed storage system.

Kaushik and Bhandarkar [47] have worked on GreenHDFS, an energy-conserving, logical multi-zoned variant of HDFS. They trace data access patterns and divide the servers into Hot and Cold zones according to their access frequency. Nodes placed in the Cold zone are changed to inactive power modes, saving a lot of energy.

Mesh networks use battery level as a metric in routing decisions [48]. But these refer to local network topologies.

Most of the work above has been developed having in mind a cluster of computers, which has different limitations compared to the mobile cloud scenario. In the approaches reviewed, nodes can be active or inactive, depending on their accessed data. If a user wants to retrieve a file that is placed in both an active and inactive node, the system will inform the user to retrieve the file

from the active node. This decision is made because the system associates a lower energy cost in retrieving the file from the active node.

This could be adapted to work in a mobile cloud scenario. Instead of selecting a node only based on the data access patterns it has, the selection could also be based on the energy levels it has.

We give now an example of this aspect. Two mobile nodes with 10% and 100% battery levels, respectively, have a file that is being requested by a user. The system could associate a lower energy cost in retrieving the file from the node with 100%. The idea of introducing a cost in deciding where to read from or write to, according to nodes' energy level, should be implemented in the current work.

2.4 Network Coordinates

In order to reduce the latency to data resources, the data-sharing system should store data as close as possible to the end-user. Consequently, it must be location-aware. The networking nodes must find a way to pinpoint their position in relation to one another, so that the data sharing system can use this information to keep data in the proximity of requesting nodes.

Network Coordinate Systems (NCS) allow mapping nodes to a position in a geometric space. The most used geometric spaces are pure Euclidean spaces, and other simple geometric spaces, like the surface of a sphere. Each node has a coordinate, and distances are computed using nodes' coordinates. They accomplish this by estimating delays, without performing direct measurements to all nodes, which saves in the consumption of network resources [49]. A common problem in network coordinate systems, is that routing on the Internet doesn't always take the shortest path possible. As such, it doesn't behave as a Euclidean space, and therefore error is introduced in the system. This is known as the Triangle Inequality Violation (TIV), and it affects most network coordinate systems.

There are different approaches to find out nodes' network coordinates (NC) in a networking system. These are classified into **landmark-based** approaches, and **distributed** approaches.

2.4.1 Landmarks-based

Landmark-based approaches rely on a few fixed nodes, called landmarks, that are known to the cluster. One example of a system that uses this, is the CAN overlay network [50]. Looking at Figure 9, we see that the client measures its distance to each landmark and then computes its network position in a two-dimensional Euclidean coordinate system.

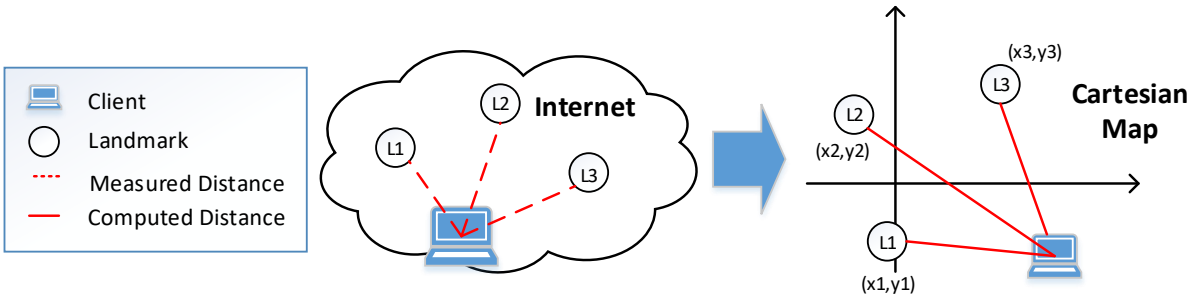


Figure 9 - Landmark-based approach [51]

The number of landmarks and their disposition affect the RTT prediction accuracy. The main drawbacks of this approach happen due to landmark failure or overload, which increase latency to the client and consequently affect the system's accuracy.

2.4.2 Distributed

Distributed approaches don't use dedicated nodes. It is distributed in the sense that nodes communicate with each other to find out their relative position. These systems have shown to be beneficial for P2P applications and overlays [25]. One of such algorithms is Vivaldi [52]. Vivaldi simulates a physical spring system, where nodes determine their coordinates by pinging a configurable number of closest nodes to them. The RTT between two nodes works as a force. The nodes' coordinates change to account for the force. The goal is to reach a state of minimal potential energy, where all forces are accounted for in the system, and all nodes have the correct network coordinates. In such a system, predicting the latency between nodes that don't ping each other is possible.

Pharos [53] is another distributed system, which evolved from Vivaldi. Each node has a pair of NC, one for global cluster, and another for local cluster. Having two groups of coordinates, allows for better RTT prediction. Both Pharos and Vivaldi, rely on the use of n-dimensional Euclidean space for their coordinates, accompanied by a height coordinate. The height vector models the latency penalty of network access links, such as queuing delay [51]. They're both iterative algorithms, as such, achieving a state of minimal potential energy is a converging process. Figure 10 shows how nodes determine their local cluster in Pharos.

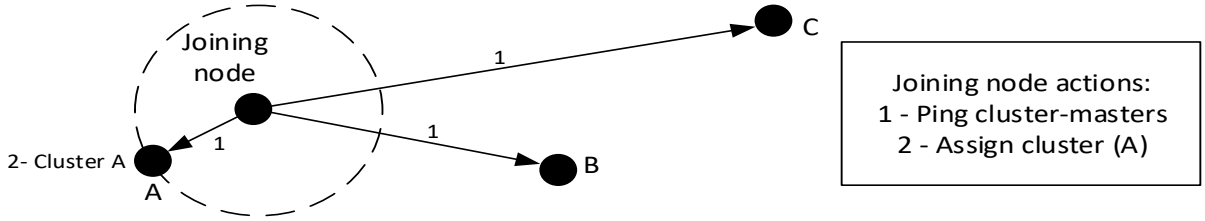


Figure 10 - Cluster selection in Pharos

In Figure 10, a joining node pings the three cluster masters, A, B and C and determines A is closest, so it places itself in the A cluster. Each client keeps tracks of two important lists, the global and local neighbour's lists. Local neighbours belong to the same cluster. The system predicts RTT latency between nodes by first looking at the nodes' clusters.

If the nodes are in the same cluster, the RTT prediction is calculated by using their local coordinates, otherwise their global coordinates are used.

Yet another distributed NCS is Phoenix [54]. It is a weight-based network coordinate system that uses matrix factorization. This approach differs from the others described above, in the sense that the coordinates do not represent a position in a Euclidean space, and as such this system does not suffer from TIV inaccuracies. Instead, it maps distances into a large distance matrix, which is then factorized into smaller matrices and stored on nodes. These matrices have embedded in them, the predicted distance between nodes. These predicted distances are then associated to a Relative Error (RE), which indicates their accuracy. Phoenix adds weight to matrices according to their RE, thus containing spreading inaccurate information. We have not found an implementation of Phoenix to work with.

2.4.3 GPS

Another popular approach for localizing nodes is with GPS. This differs from network coordinate systems in the sense that GPS doesn't attempt to globally model Internet hosts positions using absolute coordinates. It is used more often to contribute to specific application needs. A GPS receiver discovers its localization by computing the distance separating itself, from several satellites. The receiver uses the arrival time of each satellite to calculate the distance between them, and then intersects this information. The resulting coordinates are usually converted to latitude and longitude. GPS can work with three or more satellites. Nowadays, any smartphone comes with GPS incorporated.

GPS fails to meet our requirements because geographical coordinates do not map the latency between nodes on the Internet properly. In addition, GPS consumes a lot of energy on mobile devices [55]. Also, using GPS would limit our infrastructure proposal to nodes that support GPS – most network elements, used in fog computing, do not support GPS.

Our best candidates for achieving location awareness in a mobile cloud scenario, are using a landmark based approach (like CAN topology) or a distributed approach, such as Vivaldi, Pharos or Phoenix. In our thesis, we will explore the best solution, after performing some evaluation tests (in section 5.2.1).

Chapter 3. Proposed Solution

The current work aims to provide for a data-sharing-cloud system that has mobile devices cooperate and share their extra storage resources. This section specifies our proposal for that system. We first design a mobile cloud computing architecture and detail how the cloud resources are provided to the end-user (section 3.1). We then propose a data-sharing system to manage the cloud's data storage (section 3.2). We focus our efforts on keeping the data as close as possible to the end-user, while being energy-aware. As such, we propose an efficient data placement scheme that takes into consideration nodes' battery levels as well as their position (section 3.3). The data placement scheme will include a data replication module, that has the goal of replicating popular data as close as possible to the interested consumers. The data placement scheme could also consider using other networking node metrics, such as: available storage capacity, connectivity level, or mobile speed. These have been left for future work.

3.1 Architecture and Infrastructure

As seen in the previous chapter, different architectural approaches could be selected for designing a storage system which has its resources pooled with the help of mobile devices. We have opted to use a hybrid design solution between the fog computing and mobile cloud architectures, as addressed in sections 2.1.2 and 2.1.3, respectively. Mobile users, located at the network edge, shall put data in the fog nodes, with the goal of keeping data close to the final data consumers. We propose two types of nodes. The first represents the mobile devices. The second are fixed fog nodes at the edge of the network that are in the vicinity of mobile nodes. These fixed fog devices are commonly installed at users' premises, such as: switch, router, tvbox, set-top box, network-attached storage (NAS) or a computer. We argue that eventual storage scarcity on these nodes can be surmounted by connecting hard-disks via USB. These nodes do not have energy limitations and since their connectivity is good, they can be seen as a stable networking node. As such, storing data on them ensures data availability, and at the same time brings the content closer to the final data consumers.

Regarding the control management architecture, we have selected the centralized control approach. We have chosen this option because it is easier to collect mobile node metrics and use them in a flexible and intelligent data placement scheme, to decide where to store or retrieve data from. Especially considering the size of the network we are aiming for.

The Figure 11 visualizes our architecture and infrastructure proposal.

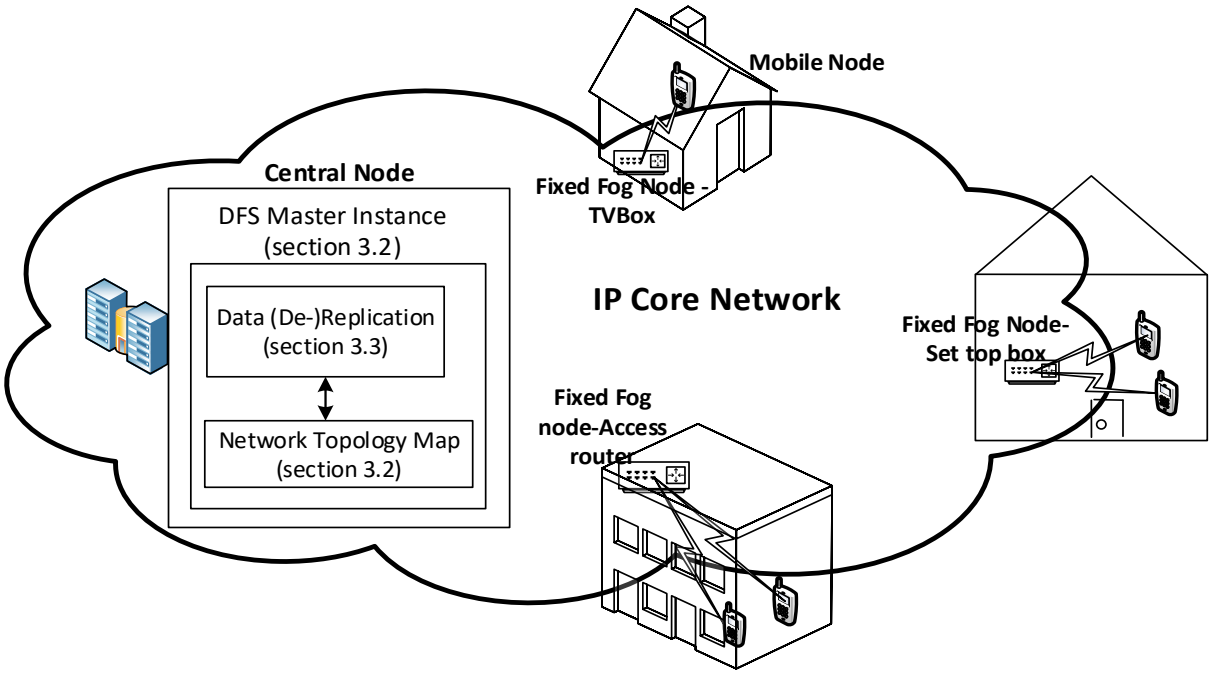


Figure 11 - Architecture and infrastructure proposal

The central node shall receive all the user requests and keep metadata information for the files stored within the system. The central node controls the data flow between the data-storing nodes, by using the data placement scheme. When an end-user puts a file in the system, the central node decides which node in the fog is close enough to store the file. It can be saved in either fixed fog units, or other mobile devices.

All nodes must have access to this central node. This central node shall run on a fixed device. This option enables a more robust final solution in terms of both connectivity and energy availability. Nevertheless, it can potentially have a limitation due to its centralized control design. To mitigate this limitation, secondary central nodes could exist, in standby mode, ready to act in case of a failure on the central node.

3.2 Data Sharing System

Section 2.2 reviewed some options for a data sharing system. A system could be built from scratch, or an already available implementation of a distributed file system could be used. We have opted to use a DFS, as they already come with core file system functionalities, like reading, writing and indexing files. They also have replication and load balancing mechanisms implemented, which we can tweak to satisfy our system performance requisites.

Considering our centralized control decision, we are naturally driven to use also a centralized data sharing architecture, within the cluster-based approach, as dicussed in section 2.2.

We extend our proposed architecture in Figure 11, by adding a centralized data sharing system on top. The central node supports the DFS master component. It stores metadata and has access to the network topology map. The network topology map is responsible for the system's context awareness. It should keep track of the cluster nodes' position and battery levels. The retrieval of these metrics on each data-holding node is further explained in chapter 4 (section 4.2). The central node uses a data placement scheme to determine the flow of data in the system. The data placement scheme uses the context awareness information stored in the network topology map.

The mobile devices as well as the fixed fog units, should have installed a DFS server instance, to allow for storing data in the system. Fixed fog units that are scarce in resources, can eventually run open-source OSs that make possible installing a DFS server instance to store data on. The mobile devices should also install a DFS client instance, to allow the end users to interact with the data sharing system. When an end user requests a file via the client instance, it is the server instance who retrieves the file (contacts the central node and retrieves the file from another server instance). Once the client instance has received the file, the end user can then access it. The node that received the file, cannot however send this file to other nodes in the system, as it never got stored on the server instance.

3.3 Data Placement Scheme

This section presents our proposal for a data placement scheme. It includes how context awareness should be managed and how it interacts with the data placement scheme. A replication strategy is proposed for simple write operations, and a sorting strategy is proposed for read operations. In addition, a file replica management system based on data popularity is proposed. Figure 12 shows a summary an overview of our proposed data placement scheme.

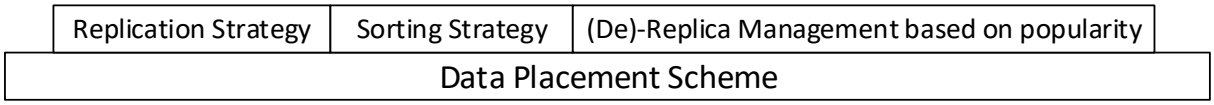


Figure 12 - Data placement scheme overview

As previously mentioned, the main goal of the system considered in this work, is to keep data close to the end user. The general idea is to store and retrieve data from fixed fog units. They are typically placed on the same network as mobile nodes and do not suffer from energy constraints. They also have a reliable connection. By doing this, the end-user benefits from data availability, responsiveness, and energy gains. To further increase these advantages, we propose to have a default number of replicas in the system for each file. This number should be easily configurable and will be stored in a parameter called 'DefaultNrReplicas'.

Although storing and retrieving data from mobile devices are possible, they should generally have a bigger cost associated than storing/retrieving data from fixed fog units. The data placement scheme should use an algorithm that weighs the battery level as well as the distance between nodes as a cost, and decide based on that cost where to store/retrieve data from.

Integrating the metrics in the system, to enable the replication strategy to evaluate the cost, is a two-part process. First, the networking nodes' positions and battery levels must be retrieved and sent to the central node. Second, the network topology map, on the central node, should support storing the nodes' metric information. The central node must continuously update the network topology map with received metrics. When needed, the data placement scheme accesses the network topology map and uses the nodes' metrics for evaluating the necessary costs. Figure 13 shows a high-level representation of the file system interactions that occur between data-holding nodes and the central node.

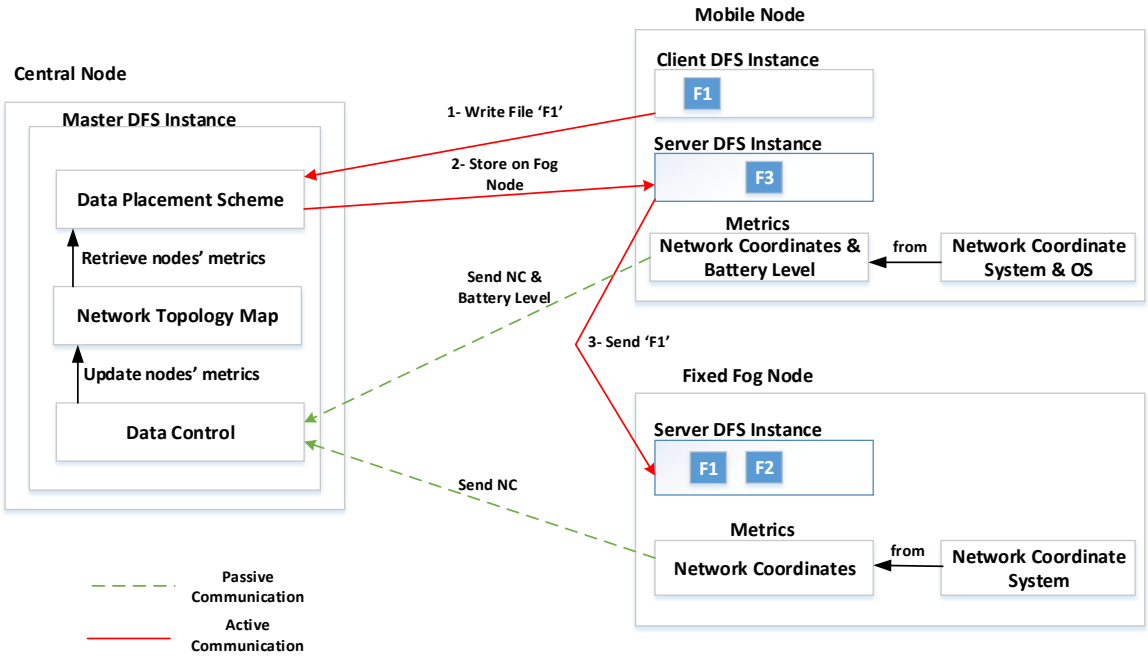


Figure 13 - Interaction between central node and data-holding nodes

From Figure 13, it is possible to identify two types of communication, that we discuss as follows. The first communication type is passive and it is responsible for updating nodes' information on the central node. It is passive because the metrics aren't explicitly requested; they are periodically retrieved and sent by the data-storing nodes, and then received and saved on the central node. Network positions are retrieved using a network coordinate system. If the data-holding node has battery, it should retrieve this information from its OS. The central node receives this information in a Data Control unit, which updates the network topology map with the metrics. The network's topology map should be kept updated, as it is used in the data

placement scheme, who is responsible, as we have already mentioned, for deciding where data gets stored/retrieved from.

The second communication type (active) is also present in Figure 13. It is associated to a filesystem write request from the mobile node. The data placement scheme weighs the cost between distance and battery level for all the network nodes, to determine the fog nodes where the data should be transferred to.

File Write Operation

We follow-up by describing how filesystem operations work in the system, as well as presenting our proposal for the replication algorithm. Figure 14 shows how the write operation is managed by our system.

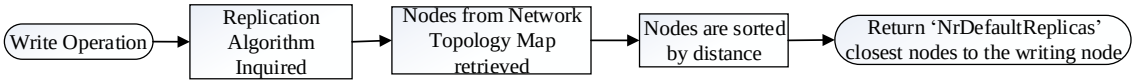


Figure 14 - Filesystem write operation flow diagram

When a file is written to the filesystem, the central node inquires the replication strategy to decide where to save the data to. The replication strategy uses an algorithm that takes into account data-holding devices' current battery levels and positions. This information is used in (1), to calculate weighted distances (WD) between nodes. The weighted distances are evaluated between the writing node and all other nodes, and sorted according to the closest rank criteria. The 'NrDefaultReplicas' closest nodes are selected by the DFS to store the file.

$$\boldsymbol{WD}_{(\boldsymbol{Node1},\boldsymbol{Node2})} = \mathbf{T}.\overline{\boldsymbol{Node1},\boldsymbol{Node2}} + \boldsymbol{\omega}.\frac{\mathbf{1}}{\boldsymbol{BatteryLvl}\,(\%)_{\boldsymbol{Node2}}}\ (\boldsymbol{s}) \qquad \textbf{(1)}$$

Where 'T' and 'ω' are constant parameters (non negative real numbers). These parameters reflect the weight of importance for storing data, according to the predicted distance and battery level, respectively. Node1 is the writing node, and Node2 is cycled for every node in the cluster, in order to determine the 'NrDefaultNodes' with the shortest WeightedDistance.

Different NCS output different NC, and NC are mostly placed in a Euclidean space. We use Euclidean distances to calculate distances between networking nodes. This is explained in section 4.2.

Expression (1) works so that the lower the battery level, the bigger the distance between two nodes. Since edge units (fixed fog units) don't have battery, they should end up being used for storage more often than battery-operated devices. Considering two mobile devices at the same distance from a writing node, the one with higher battery level is selected. This solution keeps

data close to the end-user, while simultaneously taking into consideration nodes' battery levels, in an effort to enhance the energy efficiency, data availability and the system's lifetime.

In a scenario where there are only mobile nodes nearby, storing files on other nodes can lead to low fairness and network resource abuse. In these situations, caching techniques could be developed that store data on the mobile node writer, rather than storing it on other nodes. This is a consideration that has been left for future work.

Other approaches can be taken in the future regarding how battery level is taken into account in the formula. But for now, our proposal will be suffice in showing that this will bring energy efficiency and data availability gains to our system. Moreover, our proposal could be extended to include other important metrics that we do not address in the current work, such as available storage capacity, and a few others mentioned in section 6.2.

File Read Operation

Figure 15 shows how the read operation works in our system.

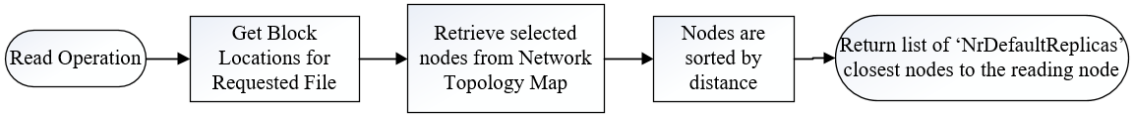


Figure 15 - Filesystem read operation flow diagram

When a file is requested for a read operation, the central node checks which devices are storing that file. It then retrieves the metric information on these nodes from the network topology map and calculates the distance to the reading node by using a sorting algorithm**.** The sorting algorithm uses (1) to evaluate the distances between the reading node and the nodes that have the requested file. The central node then sends to the reader a list of the nodes containing that file, ordered by the closest node occupying the first position. The reading node then requests the file from the first node on the list, and only goes onto the second node if the first transmission fails, and so on. When a transmission fails on a node, the reading node should report this back to central node. When it fails on every node that is in the current node list, then the central node should send the reader a new list of nodes to read from, that do not include the ones that have already failed in the previous phase.

Replica Management based on Data Popularity

We propose a replica management (RM) module based on data popularity. The more popular a file is, the more benefits in having more replicas spread throughout the system. This increases content availability, and decreases latency in accessing the data.

The replica management module should determine which data is popular enough to be replicated; and where to replicate it to. This requires a popularity estimating model, which can be static or dynamic. We will be using a dynamic model because it models the users' interest for files better than the static model, as mentioned in section 2.3.1. This module should use the context awareness information that the data placement scheme already uses.

Generally, the higher the popularity, the more replicas we want in the system. But they shouldn't be concentrated in a single area. They should be placed in diverse areas, where they are considered popular, meaning that the users located in those areas have very high expectations to consume that data.

We see two ways to support the RM module in our solution. One is to use a proactive popularity predicting model based on social-driven propagation dynamics [42]. This solution associates data's popularity to edge networks. This would allow the system to replicate content to where it is most requested. The second way, is developing a reactive module that uses a dynamic data popularity model for evaluating popularity. The module could then manage replication by factoring in the file's evaluated popularity, with location awareness information on the reading node position and where the requested data is placed.

We have used the second approach to design our RM module, because we feel that a reactive system is less likely to encounter failure, than a proactive system. We follow-up describing how we have implemented the replication module in a reactive way.

Our RM module shall run whenever a 'read' request is received on the central node. It shall calculate the files' popularity according to it's near-past access frequency.

The module then draws a searching circumference, with radius 'b', around the reading nodes' position (Fig. 17). The radius of the circumference is inversely proportional to the popularity of the requested file. The module then checks if the requested file is stored on a node within the circle area associated to the current circumference.

If it is, then the file is located close enough to the reader and it doesn't need to be replicated. If it isn't within the circle, then the system should register a 'hit' for that user/file. The term 'hit', reflects the system's intent to replicate a file to a node closer to the reader. After a configurable amount of consecutive 'hits'– 'hit threshold', the system replicates the file to the closest node to the reader using (1). This parameter confirms that a user has a high expectation to consume data (based on past requests) which is placed far away, and should be replicated closer-by.

Figure 16 portrays a flow diagram of how our replication module works.

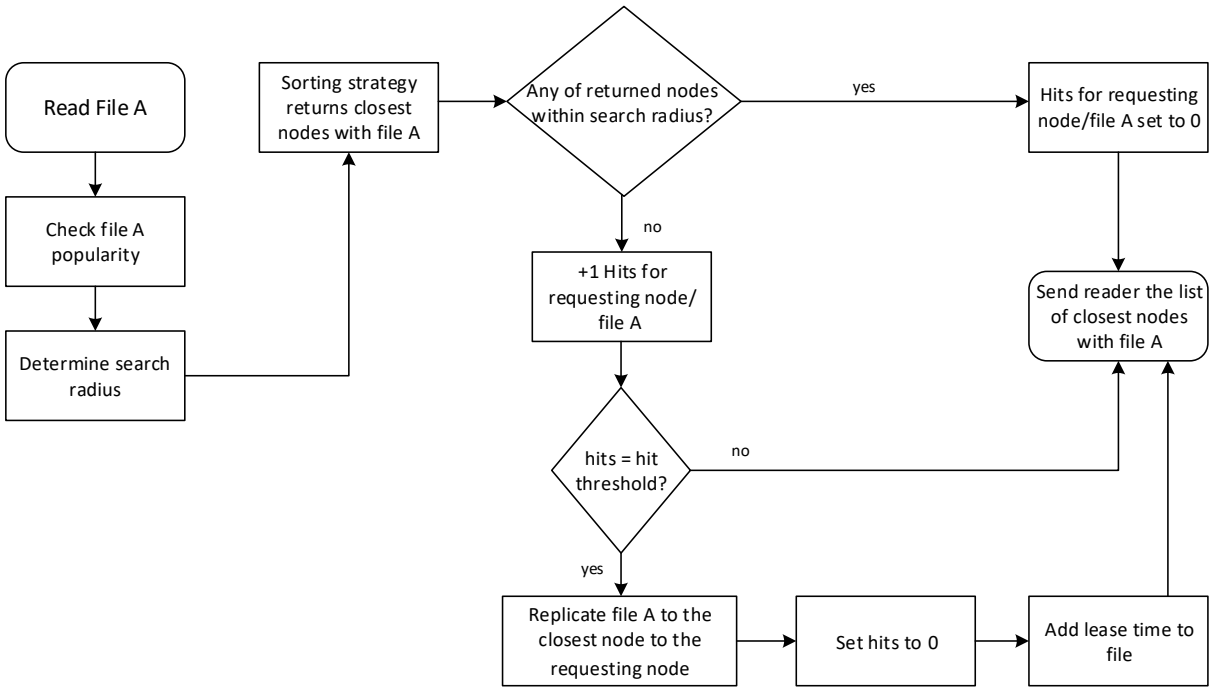


Figure 16 - Flow diagram for replication module

We follow-up by presenting our proposal for calculating the 'b' search radius (Fig. 17), as well as how to calculate files' popularity.

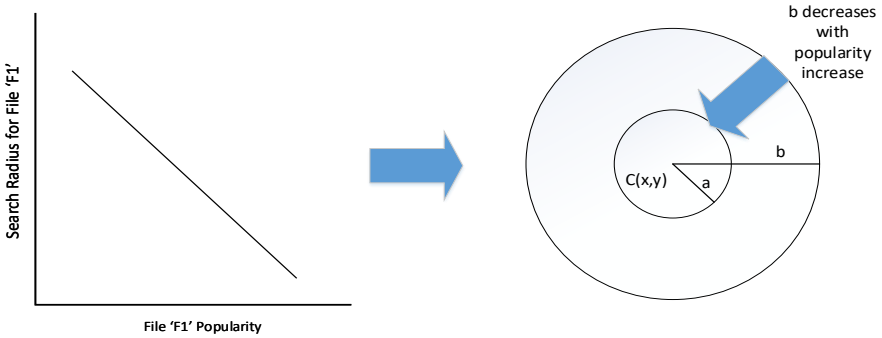


Figure 17 - 'b' searching radius decreasing with popularity increase

With our current solution, popular content gets consistently replicated to the network edges. Alternatively, unpopular content doesn't get replicated, as the search radius should be large enough to include the nodes that already store the requested file.

The algorithm behind the evaluation of the 'b' search radius is presented in (2).

$$b = k.a + \beta \frac{1}{P} \text{ (s)} \quad \textbf{(2)}$$

Where 'P' is the files' popularity for the current request, and 'a' is the RTT of the "NrDefaultReplicas"th closest node to the read requesting node. 'a' works as the minimum possible search radius, since files get written to the closest "NrDefaultReplicas" nodes to the writing node. 'a' reflects the node proximity of a requesting node.

'k' and 'β' are configurable parameters that work as constants. The higher their values, the bigger the search radius 'b', and therefore less replication in the system. Specifically, 'k' contributes to the search radius in proportion to 'a'. It works as a regulator for node proximity. While 'β' contributes to the 'b' search radius in proportion to the requested files' popularity. It works as a regulator that reflects the importance the system gives to popularity, in replicating files. Higher values mean less importance is given, and therefore it is harder to replicate – a file needs to be more popular for it to be replicated.

'hit-threshold' is another configurable parameter, for the amount of consecutive 'hits' the system must register, before replicating a file closer to a specific user.

At the end of this chapter, all parameters are discussed in more detail.

Our proposal for how files' popularity is calculated, is inspired on the temporal sliding window average [56], which is used in TCP to estimate Ack timeouts. It uses moving average calculations to smooth out short-term fluctuations and highlight long-term trends. The popularity will reflect the trend of access frequency for any given file. The file popularity is also shared amongst all the data chunks associated to that file. In (3) and (4), we can see our proposal for how to calculate files' popularity.

$$\boldsymbol{P} = \frac{\mathbf{1}}{\mathbf{Avg}\,\Delta\boldsymbol{t_n}}\ (\mathrm{s}^{-1}) \qquad \textbf{(3)}$$

Where,

$$\mathbf{Avg}\,\Delta\boldsymbol{t_n} = \Delta\boldsymbol{t_n}.\propto\ +(\mathbf{1}-\propto).\mathbf{Avg}\,\Delta\boldsymbol{t_{n-1}}\ (\mathrm{s}) \qquad \textbf{(4)}$$

As visualized in (4), Δtn is the time that has passed from the last access of the file to the present time. Parameter ∝ is a constant that determines the system's reactivity to the current Δtn. The Avg Δtn of a file ends up being equivalent to its access frequency in the near past. It relates to the popularity history. When a file is accessed more often, its Δt_n will decrease, and so will the Avg Δt_n, which will reflect a high popularity for that file, as shown in (3).

De-replicating unpopular content

As important as it is to not have under-replicated content in the cloud, it is also important to not have over-replicated data occupying unnecessary storage, essentially in nodes with scarce available resources. In Figure 18 we can see how we propose to control over-replicated content.

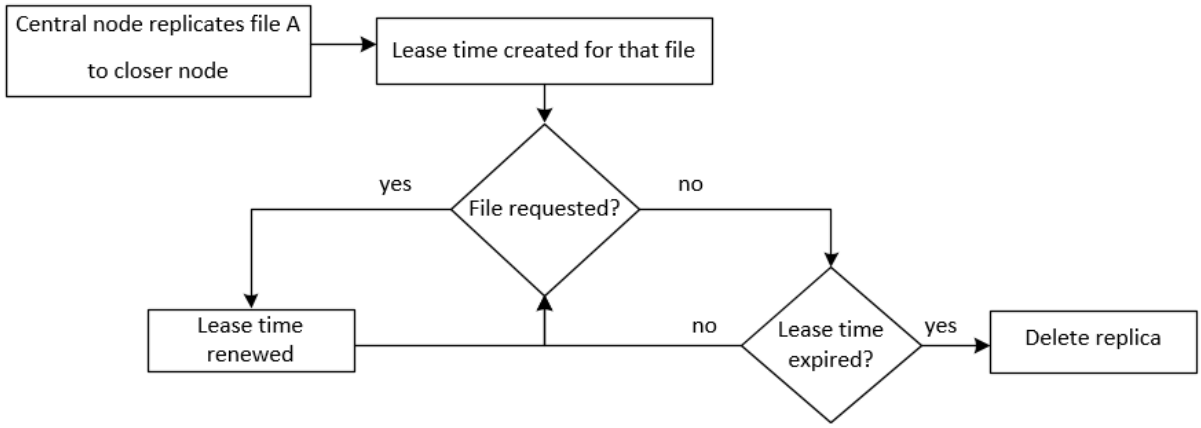


Figure 18 - Deleting replica strategy

Only when a file is over-replicated (higher than the default 'NrDefaultReplicas'), will it be given a lease time. The lease time is renewed whenever the file is requested. If the lease time expires, the file is deleted on the node furthest away from the initial writer of the file, or its last known position. This way, we are balancing out the system, keeping the replication number of a file adjusted to its popularity.

Table 4 lists the configurable parameters used in the system and their purposes. The parameters are then detailed.

Table 4 - System configurable parameters

Parameters	Used in	Purpose
NrDefaultReplicas	Replication/ Sorting Strategies	Number of default replicas for each file in the system
ω	RM (1)	Regulate fairness in energy consumption
T	RM (1)	Regulate content distribution according to distance between nodes
k	RM (2)	Regulate replication according to node proximity
β	RM (2)	Regulate replication according to popularity level
Δt_n	RM (4)	Hit interval, for calculating a file's access frequency
$\propto$	RM (4)	Regulate reactiveness to current Δt_n
LeaseTime	RM - De-Replication Strategy	Time given to over-replicated files to be accessed, before de-replicating them

In the following text, we briefly describe the system's sensitivity to changes to the values of the parameters shown in Table 4.

Increasing NrDefaultReplicas is a trade-off for occupying valuable storage space, and enhancing data availability to the final consumers.

Increasing ω, will increase the fairness in energy consumption in distributing content. Alternatively, lower values of ω mean less fairnes, and therefore battery level is not so much an important factor in distributing content, and so nodes with popular files can have their energy drained out in a more easy way.

There really isn't any point in increasing T because after all, when using (1), we are only adding distance adjusted to nodes' battery levels, to the calculated distance between nodes using their network coordinates. As such, we give T a value of one.

Parameters k and β, regulate the replication according to node proximity and popularity level, respectively. By varying parameter k, the intensity of replication that occurs in the system changes according to node proximity. Lower values of 'k' will make the system replicate with more intensity, because it requires requested data to be placed very close to the reading node. Higher values of 'k' reflect less replication intensity because data can be placed far away from a node, and this would not meet the replication requirements (of being outside the 'b' radius).

Similarly, β controls the impact of popularity on the system's replication intensity. Using higher values of β results in lower replication intensity, because the popularity level requirements are set to a higher value. Conversely, by using lower values of β, replication happens more often, as data needs lower levels of popularity to be replicated.

Δt_n reflects the hit interval between file accesses, for a particular file. In general, the lower the hit interval, the higher the popularity on the requested file. The higher the popularity, the higher the chance of it getting replicated.

Parameter $\propto$ works as a regulator for how reactive the system is, in calculating popularity, according to the last Δt_n. Higher values of $\propto$ mean that the system is more sensitive to the last measured Δt_n, and so less importance is given to the file's popularity history.

Increasing the LeaseTime means that the files that are given a lease time, have a bigger time frame for waiting on a node to request the file, before decreasing the replica count for that file by one. In this way, the storage capacity on the storing node is used during more time.

Chapter 4. Implementation

This chapter starts by justifying why we have opted for HDFS to be implemented in our system. Figure 19 shows a representation of the implementation steps that we have taken, per node type, and that are explained in the following text. Having chosen HDFS as our data sharing system, we then detail the implementation of HDFS on each node type (1st in Figure 19). Afterwards, we focus on how data-holding nodes retrieve their position and battery level. Retrieving the battery level is straightforward. To retrieve the nodes' position, we have implemented two network coordinate systems: one with a centralized approach; and another with a decentralized approach (2nd in Figure 19). We then describe how we have integrated these metrics in the filesystem's data placement scheme. This has been achieved by modifying HDFS' network topology map to use our context awareness metrics (3rd I. in Figure 19); implementing a way of sending the metrics on each data-holding node to the central node (3rd II. in Figure 19); and updating the newly modified topology map regularly with the received metrics (3rd III. in Figure 19). Finally, we discuss how we have implemented the data placement scheme, which includes implementing the data replication strategy, sorting strategy and the replica management module that is based on the data's popularity (4th in Figure 19).

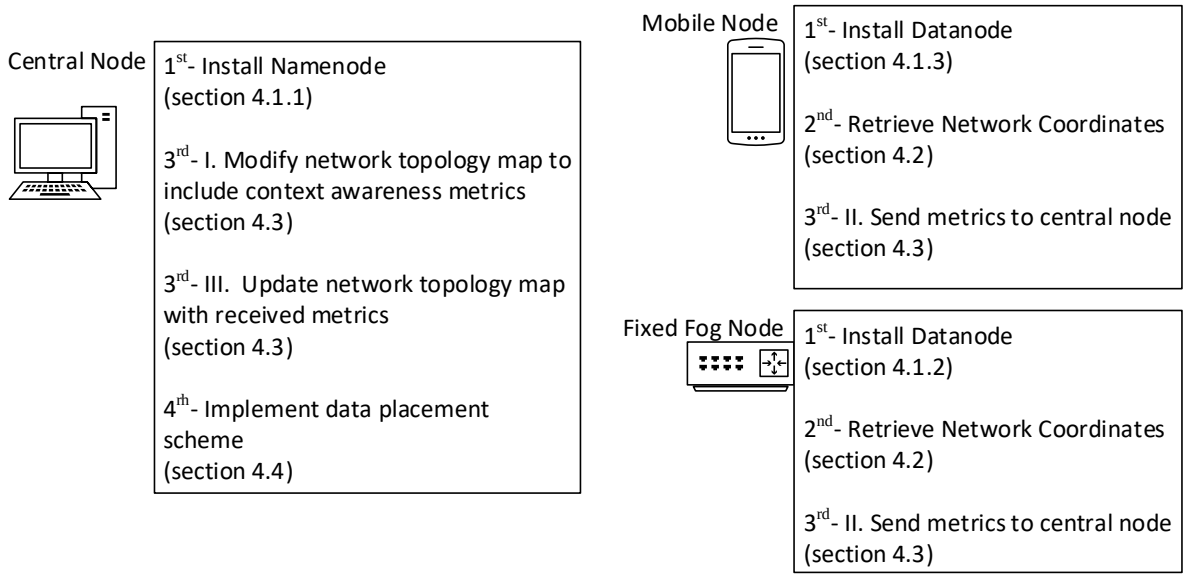


Figure 19 - Steps taken in implementing the system per node type

4.1 Data Sharing System

Our proposed infrastructure has three main components. Each runs a different OS and so choosing a DFS to work on all three platforms proved itself tricky, as it had to be cross-platform compatible. The server node has no issue installing software. Conversely, the fixed fog units usually have scarce resources, and have specialized network operating systems installed on

them, optimized for routing. This has been solved by installing an open source operating system for embedded devices on the fixed fog unit – openwrt [57]. It's based on linux and is highly customizable, allowing to port applications to the firmware, or building the firmware with support for the applications. It has a porting system that allows for cross-compiling applications. This enables for installing a DFS instance on this unit. Regarding mobile devices, the majority either run Android or iOS. Although, linux is starting to find its way into the mobile device domain [58] [59]. Android applications are developed in java, while iOS are developed in swift. With such an heterogeneous environment, the use of java is relevant.

Java is a cross platform language, which means any machine that has a java environment installed, can run a java application. With this in mind, and taking into account the work reviewed in section 2.2, we have opted for HDFS as our DFS, as it is written in java. In this way, we can install openwrt on the fixed fog unit, and port java to openwrt. Once java is installed, we can install HDFS on it. As for mobile devices, we choose to go with Android, which also supports java, although it uses a different implementation from the standard java that HDFS is developed in.

Recalling how HDFS works, the namenode is a component that runs on the central node and manages where data get stored according to its replication strategy, and where nodes read from, according to its sorting strategy. Datanodes are data-holding components that in the current work will be running on fixed fog units and mobile devices. The end user communicates with the filesystem through an available java API.

Next sub sections discuss how the diverse HDFS were implemented on each node type of our proposed system.

4.1.1 Central Node

For deploying the central node, HDFS has been installed on an Ubuntu OS. Ubuntu allows an easy installation of HDFS, by first installing java through the apt-get repository. This central node launches the namenode.

4.1.2 Fixed Fog Node

In order to install HDFS on a fixed fog node, java must first be installed on openwrt.

Nodes that run openwrt normally do not have enough RAM resources to run a Java Virtual Machine (JVM) properly. Openwrt supports a lightweight JVM called JamVM [60] that has been developed to use fewer resources when compared to other JVMs. Unfortunately, Hadoop components don't run on this JVM [61].

One way to get HDFS to work on openwrt would be to cross-compile the java code to native [62]. However the HDFS project is too big and complex to make this option worthwhile.

A openwrt associated forum [57] had a user [63] allegedly pull off installing java on openwrt by building the openwrt firmware with eglibc support. According to the forum, eglibc method enables to install in a successful waym a soft-float version of oracle embedded java on openwrt.

We end up building openwrt with eglibc support as suggested. We have used x86 for the oracle java processing architecture. Openwrt can be built with eglibc support or uclibc support. Unfortunately we have noticed that the package management system only supports uclibc. Consequently, we have been forced to build packages manually for our OS.

Fixed fog nodes may have storage limitation. We can overcome this limitation by adding extra storage via USB disks. Openwrt allows mounting additional storage via USB, on its original filesystem. In our testbed, fixed fog nodes, implemented with openwrt, were deployed on virtual machines. Unfortunately, we have encountered some difficulties regarding the USB support on our virtual machine environment used for openwrt. We overcame this by increasing the openwrt VM's partition size. This solution is similar to the increase of the fog unit's internal storage (which can only be done by the manufacturer).

Once java had been installed, installing and launching a HDFS datanode on the fixed fog node was straightforward.

4.1.3 Mobile Node

As mentioned, we have used Android for the mobile nodes in our testbed. Android works with a different java implementation than the one required for HDFS. So HDFS does not run on Android, and as such it was necessary to port HDFS into Android.

Porting Hadoop to Android has been done in Hyrax [3]. They have ported Hadoop directly to Android by re-writing and removing offending code of Hadoop and its libraries' source code, to work with Androids' dalvik virtual machine. They succeed at launching Hadoop storage and processing components as a service.

Other approaches have been taken in getting Hadoop to work on Android devices, using the chroot method [64]. With the chroot method, one changes the root environment within the mobile device. By adding a linux distribution to the mobile device, we can then install java to run a datanode. Running a linux distribution wastes more resources on the mobile node, but also gives us more freedom to incorporate non-java written software in the system. Accessing the linux distribution is done using the command line, which makes this implementation not

very practical in a real-life scenario. Due to flexibility reasons in making a prototype, we've chosen to use the chroot method to enable running datanodes on mobile nodes. In this way, we can test and obtain results without worrying about the time-consuming process of porting HDFS to Android, which has already been done [3]. We don't expect this decision to affect the obtained results, which are discussed in chapter 5, because our virtual environment has enough resources to run the two OS's (i.e. Android, Linux). Figure 20 shows how these OS's can interact via the chroot method.

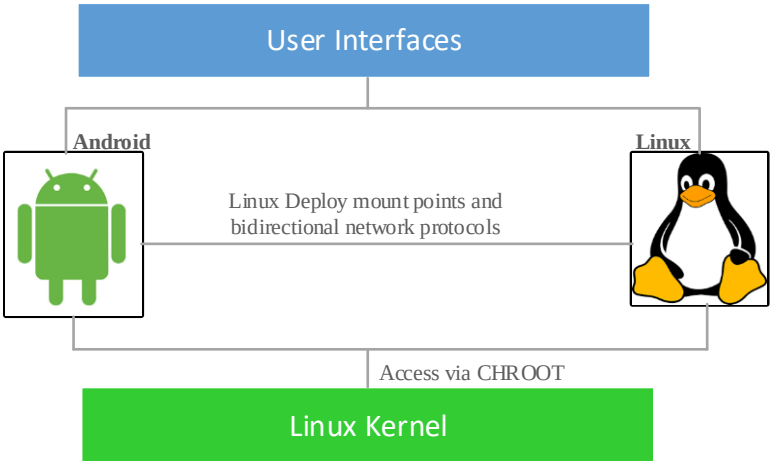


Figure 20 - Chroot method layout [65]

As we can see in Figure 20, the linux distribution is installed on top of the Androids' kernel (which is based on linux). It can communicate with the Android OS installed on the machine by configuring mount points. In this way, we have both OS's working simultaneously on the mobile device, and more importantly, they can communicate among each other. In order to launch a datanode, one has to access it through the linux shell on the mobile device. Porting the HDFS components to Android, is an option left for future work.

To perform the chroot method, we have installed an application called linux deploy [73], from google play store. This application installs the linux OS on an Android device. We then have configured the linux install environment. After that, we have installed VX connectbot [74] from google play store, which allows us to connect to our linux directory installed on the Android device. Now that linux is operational, we are capable of installing java and HDFS, like we did on the central node. We can now launch a datanode from the mobile node.

4.2 Network Coordinate System

This section discusses the implementation of two different NCS on our datanode devices. They're responsible for determining their network coordinates, so that they can be sent to the namenode. Then the network coordinates of eah datanode device can be be used in the data

placement scheme of the namenode. The first NCS is a landmark-based approach, inspired on the Content Addressable Network (CAN) topology [50], and the second NCS is a decentralized approach called Pharos [66]. We have implemented the NCS only on datanode devices, as visualized in Figure 13 of section 3.3. We follow-up by detailing the two NCS implementations and dicussing for each one, how they calculate distances between nodes. This last part is important as it is a fundamental part of our DistanceWeighted algorithm proposal, depicted in equation 1 (section 3.3). We disregard battery level in the current discussion, and solely focus on calculating distances between nodes using their network coordinates retrieved from the NCS.

4.2.1 CAN Topology

We've implemented a strategy similar to the way a CAN topology is built, to achieve location awareness. Three stationary devices are known by all nodes. They will be used as landmarks. Nodes will ping each one of these landmarks sequentially. The ping measures the RTT latency between the node and each landmark, and associates it to a specific level, according to Table 5. Each node has coordinates (lvl A, lvl B, lvl C), which are associated to the landmarks (A, B, C) latency levels. For example, a datanodes' position could be (0,2,1) meaning it has a 0-30ms RTT latency to the landmark A, a 101+ms latency to B, and a 31-100ms latency to C.

Table 5 - CAN topology level/latency mapping

Level	Latency
0	0-30ms
1	31-100ms
2	101+ms

Once a node has pinged the three landmarks, it knows its network coordinates, and is ready to send them off to the namenode. We follow-up by dicussing how the system calculates distance among nodes using CAN topology.

Distance with CAN Topology

We present in Figure 21, a node layout for a scenario where the Android node requests to write a file in the filesystem. The Figure 21 should be interpreted as a latency map, where distance equals latency. Each node calculates its NC according to the RTT latencies measured to the landmarks.

Given the network coordinates in Figure 21 (considering Android in the initial position a), we can calculate the euclidean distance between the android node and the four openwrt datanodes (Q, R, W, Z).

Nodes network coordinates are defined as x,y,z in this example, although they do refer to the latency levels to the A, B, and C landmarks.

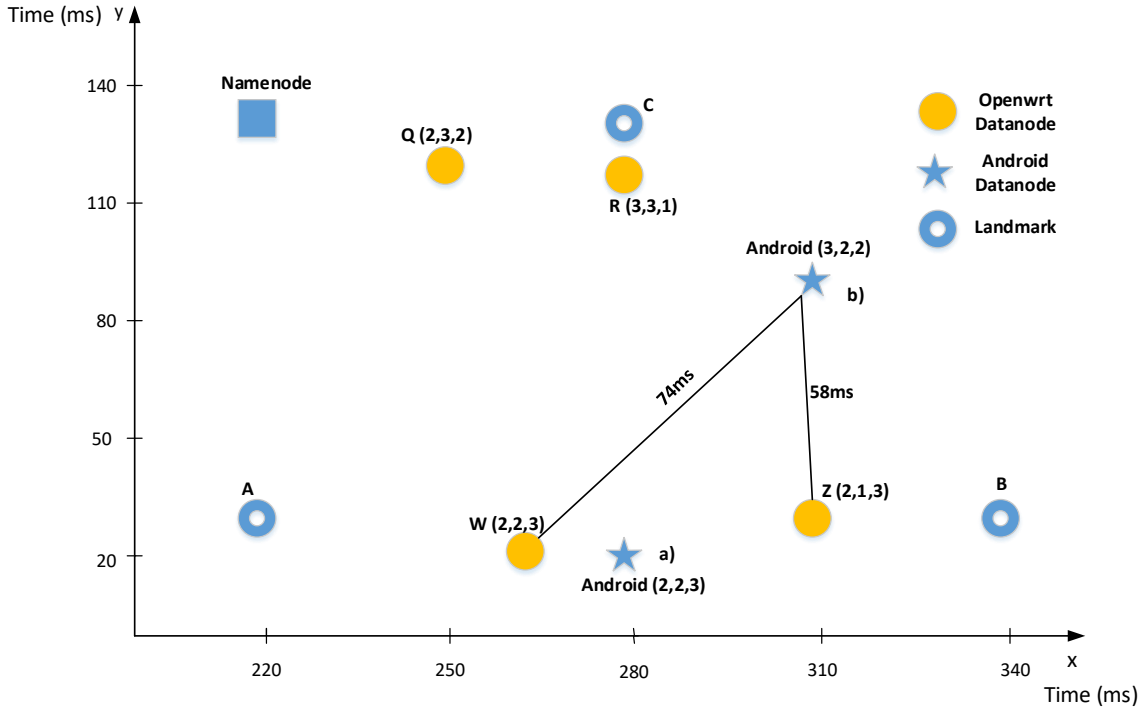


Figure 21 - Node layout in cartesian map using CAN topology network coordinates with Android in positions a) and b)

Euclidean distance between two points (A and B) with three coordinates (x,y,z) is given in Equation 5.

$$\boldsymbol{distance_{A\to B}} = \sqrt{(\boldsymbol{A_x} - \boldsymbol{B_x})^2 + (\boldsymbol{A_y} - \boldsymbol{B_y})^2 + (\boldsymbol{A_z} - \boldsymbol{B_z})^2} \qquad \textbf{(5)}$$

So, to calculate distances between the android and the four datanodes in Figure 21 would be as follows:

$$distance_{Android\to Q} = \sqrt{(Android_x - Q_x)^2 + (Android_y - Q_y)^2 + (Android_z - Q_z)^2}$$

$$distance_{Android\to Q} = \sqrt{(2-2)^2 + (2-3)^2 + (3-2)^2} = \sqrt{2} \quad ///$$

$$distance_{Android\to R} = \sqrt{6}\,;\ distance_{Android\to W} = \sqrt{0} = 0;\ distance_{Android\to Z} = \sqrt{1} = 1$$

As can be seen, disregarding battery level, the nodes with the smallest calculated distance would be the ones to store the files. Looking at Figure 21, and comparing the distance from Android to nodes Q and R, we see that Q is further away than R. So the calculated distances above should have $distance_{Android\to Q} > distance_{Android\to R}$, however, that is not evidenced by the distance results.. We study the NCS's accuracy with further in section 5.2.1.

4.2.2 Pharos

Pharos is a distributed network coordinate system. We have used an implementation of Pharos that is written in twisted python. Twisted python is an event-driven network engine written in python. It's an asynchronous language, meaning there is a single thread of control, that has different tasks work interleaved with one another. This allows for minimal blocking of the CPU.

Each datanode machine runs a client of Pharos. Getting Pharos to run on the mobile node was a matter of installing the python dependencies. Getting it to run on the fog node was trickier, as python and twisted python had to be compiled and installed to work on openwrt.

There is no implementations of Pharos that we know that accounts for node mobility. When a node moves to a new area (see Fig.10 in section 2.4.2), its cluster doesn't get updated. So every node remains in the cluster where it was placed at start-up, regardless of afterwards being closer to other areas that could belong to other clusters. For example: a mobile node starts in cluster A, and moves to an area which could be considered as belonging to cluster B. The mobile node will still determine its local coordinates by measuring latencies to nodes in cluster A, and determine its global coordinates by pinging the nodes in the other clusters (including B).

In practice, the current Pharos implementation can determine with accuracy the distances between its nodes, even if nodes move and don't have their cluster updated. This isn't however, the most optimized way for nodes to determine their network coordinates, and defies the whole point of having two sets of coordinates – accuracy in measuring distances [53]. In Appendix A – Part 1, a solution proposal for adapting Pharos to work in a mobile cloud scenario is presented. Its' implementation has been left for future work.

Pharos uses an iterative algorithm, so clients' network coordinates are constantly converging to a state in the system that is considered stable. The Pharos client offers a script, to retrieve its current NCs. By running this script, the datanode devices are now able to collect their network coordinates, and are ready to send this information to the namenode.

We follow-up by dicussing how the system calculates distance among nodes using Pharos.

Distance with Pharos

With the Pharos system, each node has two sets of coordinates, global and local. Each set of coordinates has two main components (x and y) as well as a height component (h). Each node also belongs to a cluster. The number of main components in the coordinates is configurable but we set it to two, because it has been shown that lower dimensional Euclidean coordinate systems have better accuracy [51].

Intra-cluster RTT predictions are calculated using nodes' local coordinates, while inter-cluster use their global coordinates. Equation 6 shows the formula that predicts RTT latencies between nodes, given their NCs. The formula is used for estimating RTTs between nodes using either local and global coordinates, depending on their cluster placement.

$$\boldsymbol{RTTest}_{\boldsymbol{Node1 \rightarrow Node2}} = \left(\sqrt{(\boldsymbol{Node1}_x - \boldsymbol{Node2}_x)^2 + (\boldsymbol{Node1}_y - \boldsymbol{Nod2}_y)^2}\right) + \boldsymbol{Node1}_h + \boldsymbol{Node2}_h \quad \textbf{(6)}$$

4.3 Integrating New Metrics

The goal is to make the datanode metrics (NC and battery level) accessible to the namenode, so it could be used in the data placement scheme. This involves two separate phases, which are:

1. Integrating metrics in the network topology map
2. Collecting the metrics on the datanode device; Sending them to the namenode; Updating the networking topology map on the namenode

We follow-up describing how we've implemented these two phases in our system.

Network Topology Map

In our implementation, we change HDFS' default network topology map, by associating network coordinates and battery levels to each node. In Tables 6 and 7, we can see how we've implemented the network topology map, depending on the used NCS.

Table 6 - Network topology map using Pharos

Network Topology Map
• Fixed Fog Node (Global NC, Local NC, Cluster) • Mobile Node (Global NC, Local NC, Cluster, Battery level)

Table 7 - Network topology map using CAN topology

Network Topology Map
• Fixed Fog Node (CAN NC) • Mobile Node (CAN NC, Battery level)

This way, when the system uses our distance-evaluating algorithm in equation 1 (in section 3.3), it will retrieve information from the modified network topology map. Keeping this map updated is important, as it is our context awareness tool, which we heavily rely on in our system.

Collecting, sending and updating metrics

Next, we focus on a way to periodically collect the datanode metrics in order to send to the namenode. The namenode should then acknowledge the received metrics and update its network topology map accordingly. This has been implemented in three different ways. We follow up by explaining each implementation and at the end make a comparison between them.

Gmetric4j + Ganglia

This implementation relies on Ganglia [67] - a scalable distributed monitoring tool for clusters, to collect and send the customized metrics to the namenode machine. We follow-up reviewing ganglia's architecture and describing its main components.

Ganglia uses the Gmond service, a monitoring daemon that collects information about a node and exports it via Java Management Extensions (JMX). It is installed on every node that is to be monitored. It can be extended to report custom metrics provided by scripts in any language. For this, the gmetric tool, which is incoporated in the Gmond, can also be used.

The Gmetad daemon runs on the central node. It periodically polls the data from the gmond daemon running on the slave nodes and saves it in the RRD disk.

The RRD (Round Robin Database) tool is placed on the central node and is used to store the retrieved data and their respective timestamps.

The initial idea was to install ganglia on the datanodes, to allow for collecting the custom metrics using the gmond. A script would need to be created on gmond, to retrieve the custom metrics and then publish them. Or gmetric tool could be used, to create custom metrics and give them values so that they can be exported. Installing gmond on android and openwrt is not so trivial. Instead, an open-source java implementation of the gmetric functionality – gmetric4j [72], has been used, as it is supported by our mobile and fixed fog nodes. We have found an implementation of gmetric4j to run on android as a service [71] but this has not been tested.

Figure 22 shows an illustratration of the interaction held between the ganglia tools and the HDFS entities in our implementation.

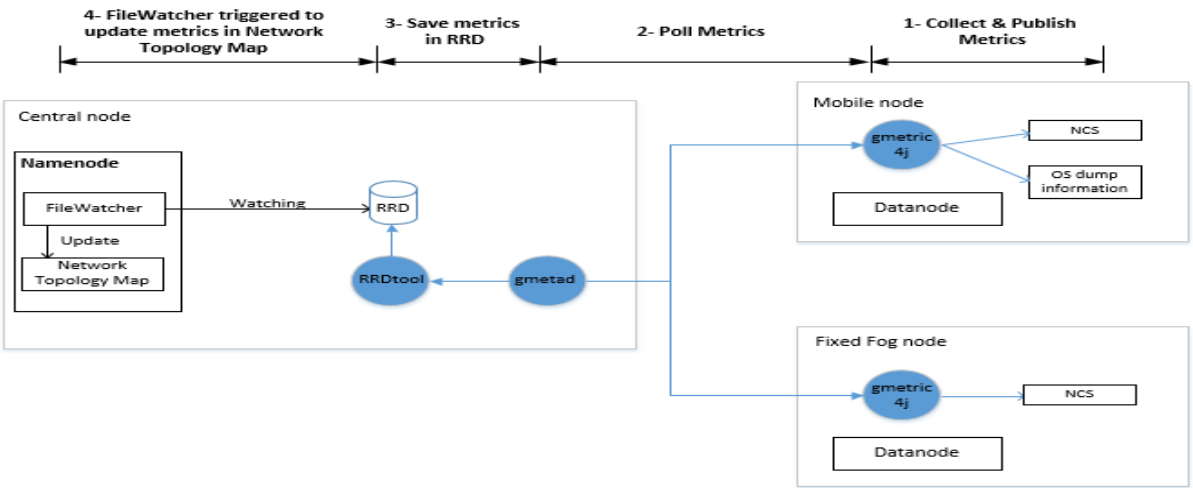


Figure 22 - Metric integration with gmetric4j implementation

Gmetric4j has been modified to retrieve the nodes' network coordinates from its NCS. On a mobile node, it additionally collects the nodes' battery level by dumping androids' system information and parsing it. Once it has collected the metrics, it then exports them, ready to be polled by the gmetad daemon on the namenode.

Next, the namenode needs to acknowledge that custom metrics have been received, and update the network topology map accordingly.

We have implemented a FileWatcher thread on the namenode, which starts running automatically on the namenode after its start-up. It constantly checks the RRD disk for customized metric updates. Our integration, visualized in Figure 22 works in the following way:

1. Gmetric4j periodically retrieves network coordinates and battery level metrics and exports them via JMX
2. Gmetad daemon periodically polls data-holding nodes for updated metrics
3. Metrics are parsed and saved in the RRD
4. If any of the customized metrics are updated in the RRD, the FileWatcher running on the namenode, triggers a method. This method retrieves the metrics from the RRD and updates the network topology map with them.

Keeping this information as updated as possible is a main concern. This can be addressed by lowering the periodicity of: gmetric4j collecting and exporting metrics; and of gmetad polling these metrics.

Disadvantages to this implementation are that it relies on a third party component to run on the datanodes – gmetric4j, to collect and send metrics to the namenode. Although not a lot, this wastes some RAM, around 15MBs. In addition, ganglia must be installed on the namenode, and gmetad must be running, to poll the metrics from the datanodes. We have tried to find an alternative option that would not overload so much datanodes and the namenode.

Metrics2 + Ganglia plugin

Hadoop has a metric system called Metrics2. It allows for Hadoop components to measure and collect data, and then export this data via JMX so it can be sent to a central node for monitoring purposes.

The idea is to incorporate our customized metrics in the list of metrics that the datanode can provide. Figure 23 clarifies how we've implemented the metric integration using the Metrics2 framework.

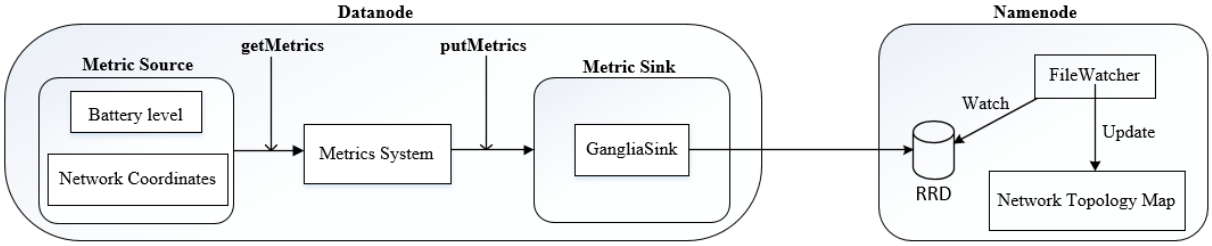


Figure 23 - Metric integration with Metrics2 framework implementation

The metric source is responsible for acquiring the datanodes' metrics. A metric source has been added to datanodes, to retrieve the customized metrics.

The Metrics System is responsible for retrieving the metrics from the metric source and putting them in the metric sink. The metric sink is where the metrics are consumed. It supports a few types of metric sinks (filesink, graphitesink, gangliasink). The filesink allows to save metrics in a file, on each datanode. But we need a way of sending the metrics to the namenode. The Hadoop metrics2 framework has a plugin compatible with ganglia - ganglia sink. This plugin allows to push metrics that are placed in the ganglia sink to the namenode to be stored in a RRD. The pushing periodicity is a configurable parameter that we have set to ten seconds (in Appendix A - Part 2). The namenode runs the FileWatcher, as in the previous implementation, to update the network topology map with the newly retrieved metrics.

The advantages compared to the gmetric4j implementation refer to the Metrics2 framework already being implemented in Hadoop. Also, although ganglia must be installed on the namenode, the gmetad daemon does not have to be running to poll the data from the datanodes. This happens because it is the ganglia sink, on the datanode, who is responsible for sending the metrics to the RRD on the namenode. In this implementation, the namenode and the datanodes have a smaller overhead when compared with the previous option, i.e gmetric4j.

Heartbeat

This implementation uses a HDFS feature designated by Heartbeat. Heartbeats are signals that datanodes send to the namenode, to prove that they are alive (online). They carry information about the total storage capacity, fraction of storage in use, number of data transfers in progress and number of failed volumes on the node. The heartbeat protocol has been modified to include sending datanodes' battery level and network coordinates (see Fig. 24). This has been done in a protocol configuration file in Hadoop's source code (in Appendix A – Part 4). We then modify code on the datanode to retrieve the current battery level and network coordinates, and send them in the heartbeat to the namenode. The namenode has been modified to update the network topology map, whenever a heartbeat is received. The default heartbeat interval is three seconds,

but it can be changed in a Hadoop configuration file (in Appendix A – Part 3). We have set it to twelve seconds.

Figure 24 shows a high-level representation of the heartbeat implementation.

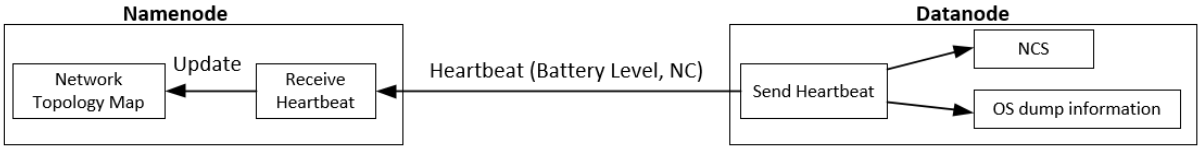


Figure 24 - Metric integration with heartbeat implementation

This implementation offers the least added overhead to the namenode and datanodes out of the three implementations, as it does not rely on third party components to keep the network topology map updated with the networking nodes' metric information.

Table 8 summarizes how the different implementations integrate the metrics with HDFS.

Table 8 - Metrics integration implementation summary

	Collecting metrics	**Sending metrics**	**Receiving metrics**	**Updating metrics**	**Overhead**
Gmetric4j + Ganglia	Gmetric4j	Gmetric4j	Gmetad	FileWatcher	High
Metrics2 + Ganglia plugin	Metrics2	Ganglia plugin	Ganglia plugin	FileWatcher	Medium
Heartbeat	Heartbeat	Heartbeat	Heartbeat	Heartbeat	Low

We can see from Table 8, that the gmetric4j implementation relies on using the external component – gmetric4j to collect and send metrics. In order to receive metrics, both gmetric4j and Metrics2 implementation need ganglia installed on the namenode. However, gmetric4j needs to run the gmetad, while metrics2 doesn't. The heartbeat implementation is the implementation that works most fully integrated with HDFS, as it is done at protocol level. Advantages to this are, we do not need to run the FileWatcher to check for received updates, and we do not have to have ganglia installed on the namenode.

The heartbeat implementation has been selected to integrate the custom metrics in the HDFS. This option has been chosen due to the low added overhead it generated on the nodes, when compared to the other options.

4.4 Data Placement Scheme

Data placement scheme modifications were made on the namenode. Main modifications made were: changing the network topology map, as explained in section 4.3; replacing the replication strategy as well as the sorting strategy to incorporate our algorithm (equation 1 in section 3.3);

recording the diverse times each file has been requested; these diverse instances of time are used to update the file popularity; and creating the replica management module based on file popularity.

We follow-up describing each of these modifications in more detail.

4.4.1 Replication and Sorting Strategies

HDFS deploys its replication strategy in a class designed as block placement module. We replaced its code for our WeightedDistance evaluating algorithm, in Equation 1 (in section 3.3). This way, whenever a file is written to the filesystem, the block placement module calculates the WD from the writer to all the other nodes in the system. Depending on the NCS being used, the WD algorithm will be using either Equation 5 (in section 4.2.1) or Equation 6 (in section 4.2.2) to calculate distances amongst nodes, as discussed in section 4.2. It then sorts the WDs and chooses which nodes to write to, with a criteria of closest node first.

When a file is requested for a read operation, the system checks the sorting strategy. Again, we replaced the code in the sorting module for our WD evaluating algorithm in Equation 1. We then have the module sort distances according to closest, and return the list of sorted nodes.

4.4.2 Tracking Popularity

Calculating a files' popularity is given in equation 3 (in section 3.3). The equation expects the AvgΔtn for that file. The AvgΔtn is calculated in equation 4 (in section 3.3). It needs the Δtn associated to the time the read request was received, and Avg Δt_{n-1}.

This has been implemented by logging read requests in a structure like:

Hashmap1(File, Hashhmap2(timestamp, AvgΔtn))

When a file is requested to be read, a new entry in Hashmap1 is created for that file, which has its value pointing to Hashmap2, which saves its key as the timestamp of the received request time, and the value as the AvgΔtn associated to that timestamp. The AvgΔtn for that timestamp is calculated using equation 4 (in section 3.3), by first retrieving Δtn. Δtn is the difference between the current timestamp and the last saved timestamp for that file. Second, it retrieves the previously stored Avg Δt_{n-1}, which is associated to the last saved timestamp.

This way, the RM module can easily retrieve the files' popularity.

4.4.3 Replica Management

According to our proposal, a RM module should run whenever a file read is requested. It is responsible for deciding if a file should be further replicated closer to the reader, according to

its popularity, and the geographical distribution of the file relative to the read requesting node's position. Eventually, it is responsible for triggering the files' replication, and evaluating where it should be replicated to.

We have divided this implementation into two parts:

- Replication decision
- Replicating the file

Where the second part only occurs if the first part decides it should do so. We will now look at how both parts have been implemented.

Replication Decision

Figure 25 shows a flow diagram that summarizes what happens on the namenode when it receives a read file request. It shows the replication decision process.

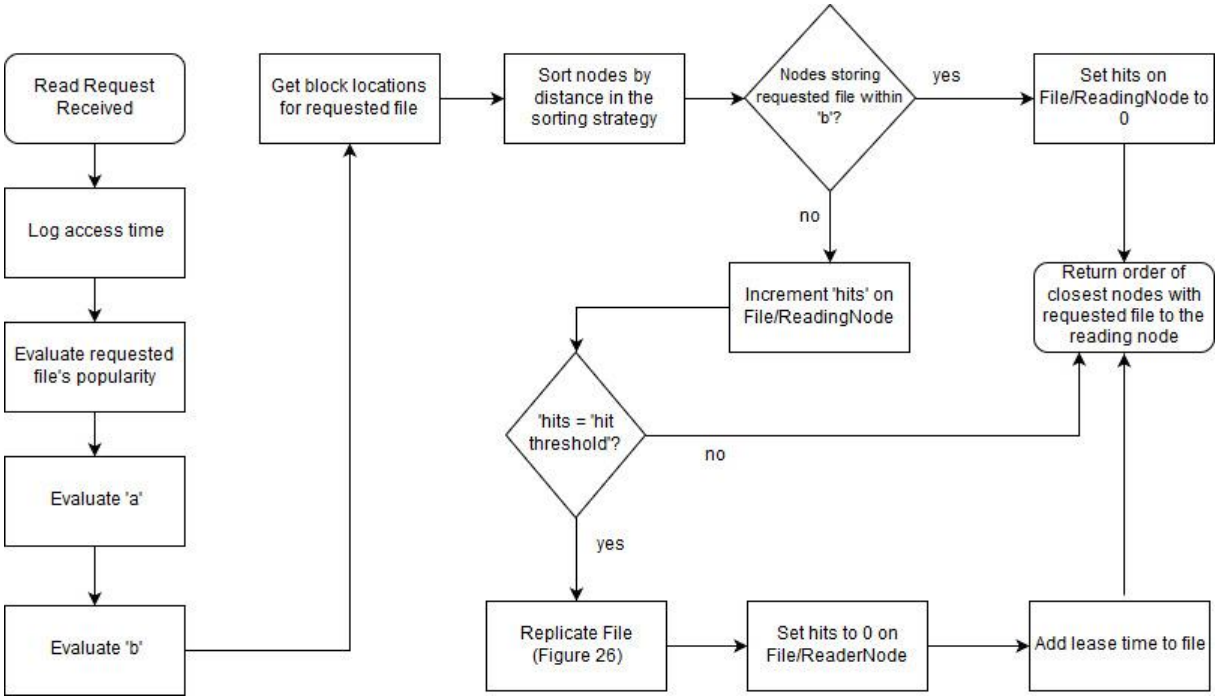


Figure 25 - Namenode flow diagram when a read request is received

When a file read is requested, the RM module starts by retrieving a 'b' search radius using equation 2 (in section 3.3). To evaluate 'b', both the 'a' parameters and the file popularity are needed, and thus need to be implemented. Calculating file popularity is discussed in section 4.4.2. We follow-up describing how calculating parameter 'a' has been implemented.

The parameter 'a' refers to the 'NrDefaultReplicas' closest node's distance to the reader. This is calculated by cycling Node2 in Equation 1 (in section 3.3) for all nodes in the readers' cluster and choosing the 'NrDefaultReplicas' closest to the reader node. Once both 'a' and the file's

popularity are calculated, the ‘b’ search radius can also be calculated by applying equation 2 (in section 3.3).

When the searching radius ‘b’ is determined, the module proceeds to retrieve the block locations of the nodes that have the requested file. The module then uses this information in the sorting strategy, to calculate WeightedDistance between these nodes and the reader. The sorting strategy then returns a list of nodes, sorted according to the smallest evaluated distance first.

After the sorting strategy returns this list of nodes, the module checks their NC and verifies, for each node, if their evaluated WeightedDistance to the reader is smaller than the ‘b’ search radius (first decision block in Figure 25).

If the evaluated distances are all bigger than the ‘b’ radius, then the system logs a ‘hit’. There are two ways we see of accounting for ‘hits’. One is to keep track of individual user requests for files, by logging ‘hits’ for the combination file/node. The other approach is to log ‘hits’ for the combination file/area. In this last approach, for a ‘hit threshold’ of three, three nodes close to each other, could each request the same file once, and that would account for three ‘hits’ for that file/area, leading to replication. The first approach is more user-centered, while the second approach is more ideal for sharing data amongst users. We have implemented the first approach.

If at least one of the evaluated distances are smaller than the ‘b’ radius, than the system sets the ‘hits’ for that file/node down to zero.

If the module registers a number of ‘hits’ for a specific file/node, that is equal to the configured ‘hit threshold’, then it decides to replicate the file. We follow-up dicussing how the file replication has been implemented.

Replicating the File

When the ‘hit threshold’ is reached in Figure 25, then file replication process, depicted in Figure 26, starts.

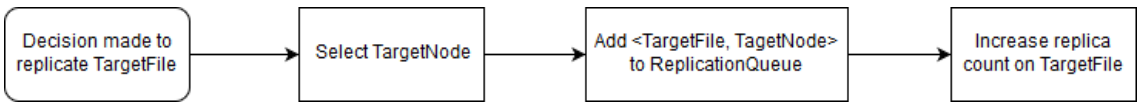


Figure 26 - File Replication process

Replicating a file requires a TargetFile to replicate, a TargetNode to replicate to, and a way of actually replicating the TargetFile to the TargetNode. We already have the TargetFile from Figure 25. The TargetNode is selected as the node that is closest to the reader node, using the distance evaluating algorithm in Equation 1 (section 3.3). Afterwards, the combination <TargetFile, TargetNode> is added to a ReplicationQueue.

The last part is implementing a way of actually replicating the data. We follow-up explaining why this is not that trivial in HDFS, and detail how we have implemented this aspect. The two last blocks in Figure 26 will be explained in the following paragraphs.

HDFS has mover and balancing utilities. What these do is allow for content to be moved, not replicated. HDFS is designed to place data to maximize data availability. It doesn't incorporate replicating files to specific nodes.

HDFS supports changing the replica count on its files. This feature has been combined with another feature that HDFS has – block recovery. Block recovery is managed in the block management module (see Fig. 27). Its purpose is to ensure that the files' replica counts are the same as the amount of replicas that are in the filesystem. The block recovery method runs at a configurable period of time, that we have set to one second as it is the lowest possible value (in Appendix A – Part 3). It works so that: after increasing the replica count on the TargetFile by one, the block recovery feature will verify that the TargetFile has one replica missing in the filesystem, and replicates it to a TargetNode according to the default replication strategy. We replaced the default replication strategy, so this last part had to be tweaked. Instead, the TargetNode is chosen by accessing the ReplicationQueue and consuming the entry associated to the TargetFile, that has been previously queued for replication (in Appendix A – Part 5). In this way, we are capable of replicating the TargetFile to the TargetNode, for a lease period.

Figure 27 shows the relation between the HDFS layers that we have used in implementing the replica management module. Both, Figures 25 and 26 are represented in the RM module in Figure 27.

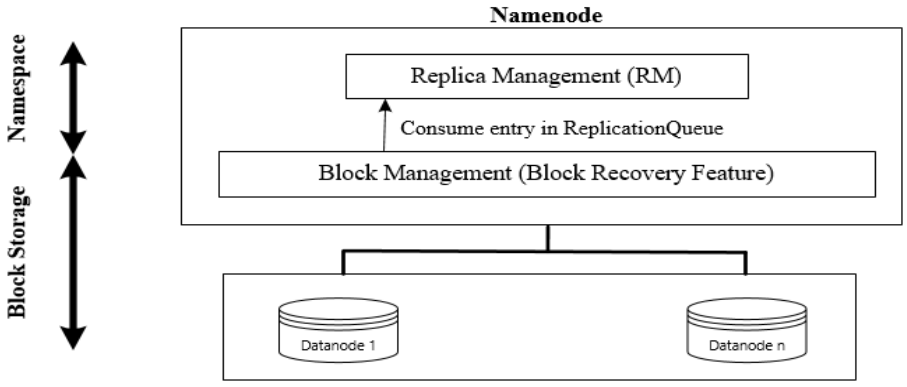


Figure 27 - HDFS layers interaction in replicating popular data [68]

The RM module is placed in the namespace layer. This layer is responsible for basic file operation such as creating and deleting files and answering read requests. The decision-making process shown in Figure 25 is held in the RM, placed in the namespace layer.

When there is replication work, then the process depicted in Figure 26 starts. Meanwhile, the block management module, which is part of the block storage layer, periodically checks if there is replication work to be done. When it verifies that there is, then it retrieves information on which node to replicate to, from the ReplicationQueue, which is in the namespace.

4.4.4 De-Replication Management

We've implemented the de-replicating data module, by firstly adding the replicated files to a list at replication time. We create a thread that is launched at namenode start-up, and constantly verifies, for each file in the list, if the last access time has passed the configured 'lease time'. If it has, the system decreases the replica count for that file by one. Similarly, to the replicating process, the block recovery feature has another method that checks if a file has more replicas in the filesystem than it should. In this case, it calls the delete replica method on the replication strategy. We change this method to delete the replica more far away from the last known position of the original writer. The file is then deleted from the replicated files list when it reaches 'NrDefaultReplicas' replicas.

Another approach could be to delete replicas on nodes who's lease time have expired. In this case, the namenode would need to keep track of access times for every replicated file, on every node it had been replicated to.

We have not implemented the second approach as it brings a much bigger overhead on the central node, than the first approach.

In the following chapter, we dicuss the tests that have been made to evaluate our system. In addition, the obtained results are comprehensively discussed.

Chapter 5. Tests and Results

This chapter initiates by describing the testbed we've used to test our implementation. We then look at how we've setup the network to carry out our tests, namely emulating latency between the testbed nodes. After that, we specify some common configurable parameters that have been used throughout the test scenarios. We follow-up by presenting a series of functional tests in section 5.1, which have the goal of proving that the implementation is working according to our proposal specifications. We also present performance tests in section 5.2, which have the goal of validating performance metrics in our system by putting the system under heavy workload and seeing how it performs. We will be testing the two NCS implementation for their accuracy in section 5.2.1. In addition, we will also be testing the data placement scheme in section 5.2.2, for the following performance metrics: latency delay in accessing resources in section 5.2.2.1; bandwidth at the backhaul links in section 5.2.2.2; and fairness in energy consumption in section 5.2.2.3.

Testbed

Our testbed is virtualized. All components run on VM's running over hypervisors: VirtualBox and Genymotion. Table 9 shows the specifications of the VMs we've used for testing. During our system evaluation we have used one namenode and a maximum of twelve datanodes.

Table 9 - Testbed specification

		CPU	**RAM**	**Storage**	**OS**
Host		Intel Dual-Core i5-4200M CPU 2.50Ghz	16GB	1TB	Windows
Node Type	**Virtualization Platform**				
Namenode	VirtualBox	Intel Dual-Core i5-4200M CPU 2.50Ghz	4GB	30GB	Ubuntu
Fog Datanode	VirtualBox	Intel Core i5-4200M CPU 2.50Ghz	512MB	34GB	Openwrt
Mobile Datanode	Genymotion	Intel Core i5-4200M CPU 2.50Ghz	2GB	34GB	Android

A VM inherits its CPU from its host. With that said, the namenode is the entity with the highest computational demand. Datanodes don't need so much processing power; they need RAM to run the datanode component, which handles the data transmissions; and storage resources to save the data.

We have configured the fog nodes' RAM to only 512MB, as there are plenty of routers nowadays sold with 512MB+ RAM. The namenode needs more RAM since it is the central node of our system. We set it to 4GB in our tests. We set the mobile node's RAM to 2GB which

can also be found in a considerable number of smartphones sold nowadays. We have connected nodes using a host-only adapter provided through the hypervisor.

Network Latency Emulation

In a real-life scenario, communication between nodes have an inherent latency that can be justified by a variety of factors, like distance, traffic load, type of internet access, etc.

Since our nodes are all on the same computer, the RTT latencies between them are negligible. In order to test our implementation, we need to somehow emulate the latency we see between nodes on the Internet. We have done this by developing a network latency emulation system on our testbed machines.

It works by spreading nodes throughout a two-dimensional cartesian map – NodeLayoutFile. The latency between nodes is considered equivalent to their euclidean distance on the map. We refer to the map as a latency-map, throughout the current work.

Routing on the Internet doesn't always take the shortest path possible, so it doesn't behave like a euclidean space. Nontheless, the latency-map helps us visualize the separation between nodes, and we expect it to be a sufficient criteria to run our tests. In addition, this implementation does not introduce triangle inequality violation related inaccuracies, in calculating distances using the NCS. Figure 21 in section 4.2.1 shows an example of a latency-map.

To emulate the latency, we use the traffic control (TC) utility available in the linux kernel. In particular, we use netem, a network emulation functionality used in TC [69]. It allows for emulating delay, loss, duplication and re-ordering of packets between nodes. It creates network latency between nodes by checking the ip of outgoing packets, and waiting the configured latency time before sending out the packet. Tc netem was cross-compiled to work in openwrt.

To configure latency links on every datanode, a shellscript is run on the namenode, that: retrieves every node's position from the NodeLayoutFile; calculates the euclidean distance between all nodes; translates these distances into latencies in miliseconds (ms); and then ssh's into each node in the cluster to issue the tc commands that create the latency links.

In order to emulate these calculated latency values on the remote datanodes, we end up running a local script on the remote datanodes. This local script is kept on the namenode machine, and has the TC commands in it, and receives the latency values as parameters from its calling-script. Some relevant code is presented in Appendix A – Part 6. Latency links between all nodes are in this way successfully emulated.

System Parameters Configuration

We have configured some system parameters, and keep these values throughout all tests. These parameters can be seen in Table 10.

Table 10 - System parameters configuration

Parameter	Value
'NrDefaultReplicas'	2
'hit threshold'	3

We have set the 'NrDefaultReplicas' to two. We argue that keeping one replica per file in the filesystem would put data availability in jeopardy. Having three replicas per file, which is the default for HDFS, seems too much, as the trade-off of storage used for data availability gained for unpopular files doesn't sound reasonable. So we have decided to manage two replicas per file because this value adjusts well to our test scenarios, where there are few nodes available. Similarly, we set the 'hit threshold' to three. Setting it to less would mean data got replicated too quickly. Setting it to more, would mean that a file only got replicated when it had been requested more often, which would mean more time spent transfering files, and so, longer running test scenarios. As such, three seemed a reasonable middle ground for the tests we want to carry out.

5.1 Functional Tests

The following operations have been tested to verify if the system is working as predicted:

1. Write file - File should be written to the 'NrDefaultReplicas' closest datanodes to the writing node.
2. Read file – Reader should retrieve the file from the closest data node to it.
3. Node mobility - System should verify when a datanode changes position, when replicating content.
4. Replicating popular content - The system should be able to replicate popular content close to the reader.

A test scenario has been created that incorporates all of the operations above. This test scenario uses the Pharos implementation as its NCS, due to the accuracy results that have been obtained in testing the NCS's, which is discussed ahead in section 5.2.1.2.

Clients normally access the HDFS using its java API, which creates a java process on the datanode, who takes care of the client's request. One java process is created per request, and it uses a substantial amount of RAM. Datanodes have 512MBs of RAM, and cannot handle the

flood of requests using the normal access API. To solve this limitation, an alternative filesystem accessing mechanism has been used throughout all test scenarios. Hadoop supports a HTTP REST API that supports the complete filesystem interface for HDFS. This system is called WebHDFS. The REST API has been launched on the namenode, enabling all the datanodes to request their files using curl commands and without the above-mentioned problems.

The following test scenario has been carried out while having wireshark capturing traffic in background. The traffic has been filtered, and processed to be presented in a graphical representation. The graphic results are then discussed.

Test scenario

The node layout for this test can be found in Figure 32 (in section 5.2.1.2). This layout has been used to force file replication. We start by instructing the Android node to write a file of 1MB in the HDFS from the (275, 25) position. The Android node then reads this file from the same position. Afterwards, the Android node is moved to (275, 137.5), where it is instructed to read the file three times. After the three read requests, the system should replicate the file from a far away datanode to a closer datanode. Once the file is replicated, the file is read one last time.

Results and Evaluation

Figure 28 shows the I/O graphic obtained with wireshark captures.

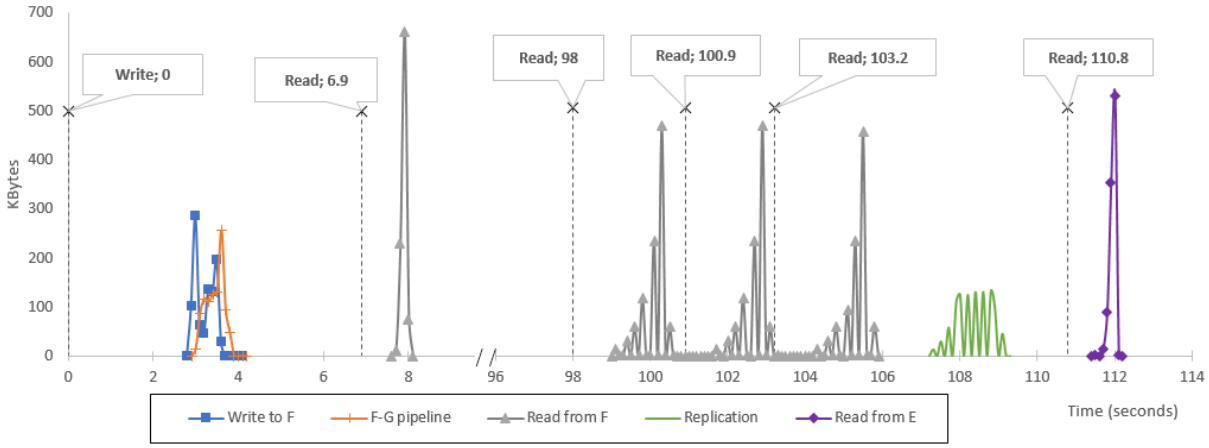


Figure 28 - Functional test scenario

Looking at Figure 28, we see that it takes 2.9s to start transferring data from the Android node to the F node. This time can be explained by the long RTT's between the partaking datanodes and the namenode, and the fact that the HDFS handshake requires many traded messages between these (see Figure 29). Summing the RTT's in the handshake, 2.1s have passed. Additionally, the datanodes show blocking periods during the handshake, which make up for the remaining 0.8s.

Overall the time it takes to start writing a block is considerable. Since writing is done in pipeline, the writing speed to each node is very similar. It takes 1.1s to receive the file on nodes F and G. The transferring rate without TCP and HDFS handshake is the following: $WriteRequest\ Rate = \frac{10^6\ x\ 8\ \text{bits}}{1.1s} = 7.27Mbps$. Future work can be done in validating if HDFS write handshake can be changed, to decrease the time it takes to start writing a file to it.

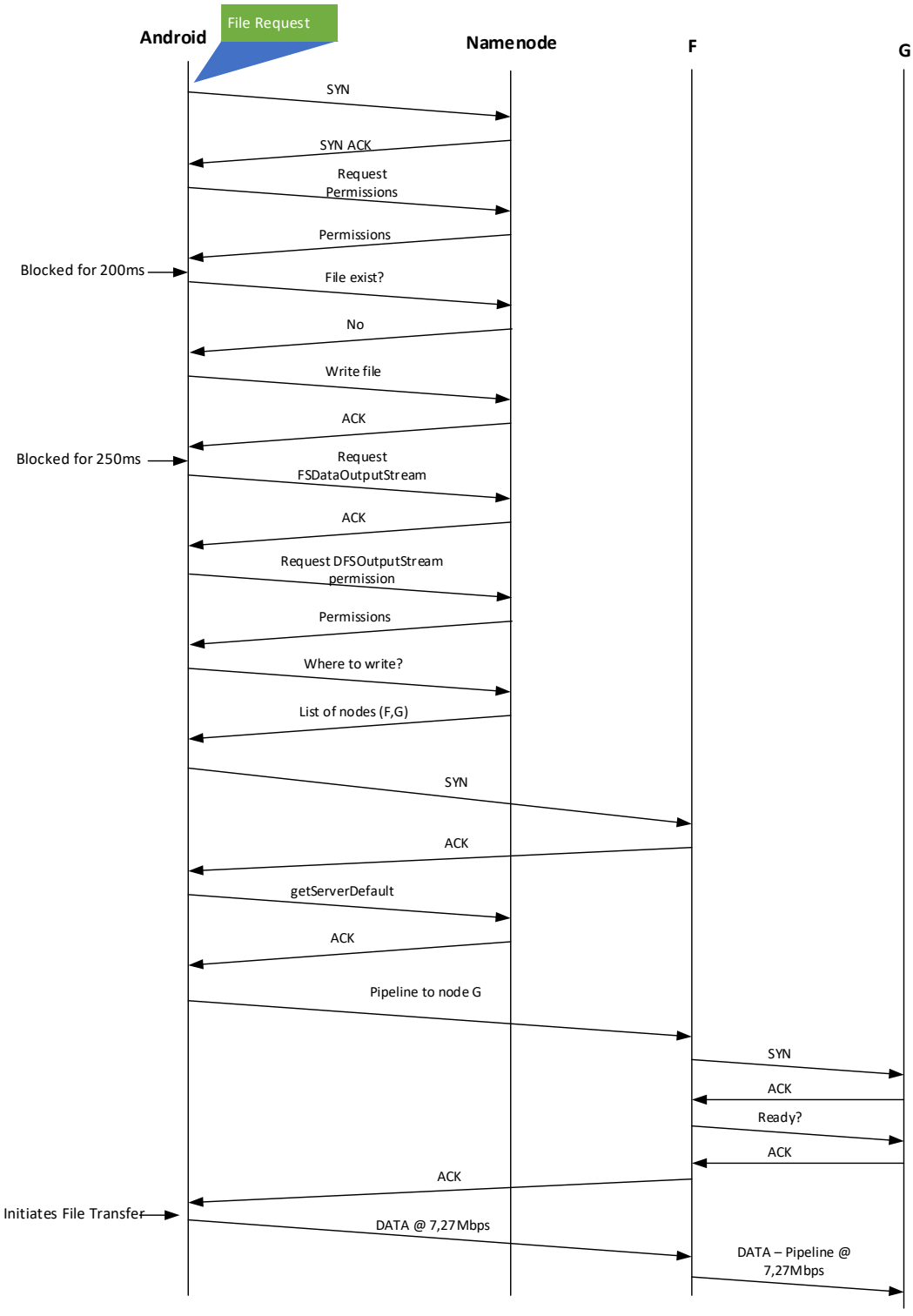


Figure 29 - HDFS write operation communication protocol

Figure 30 shows the communication protocol used with WebHDFS for read file operations. In the following text, we discuss the HDFS read file operation.

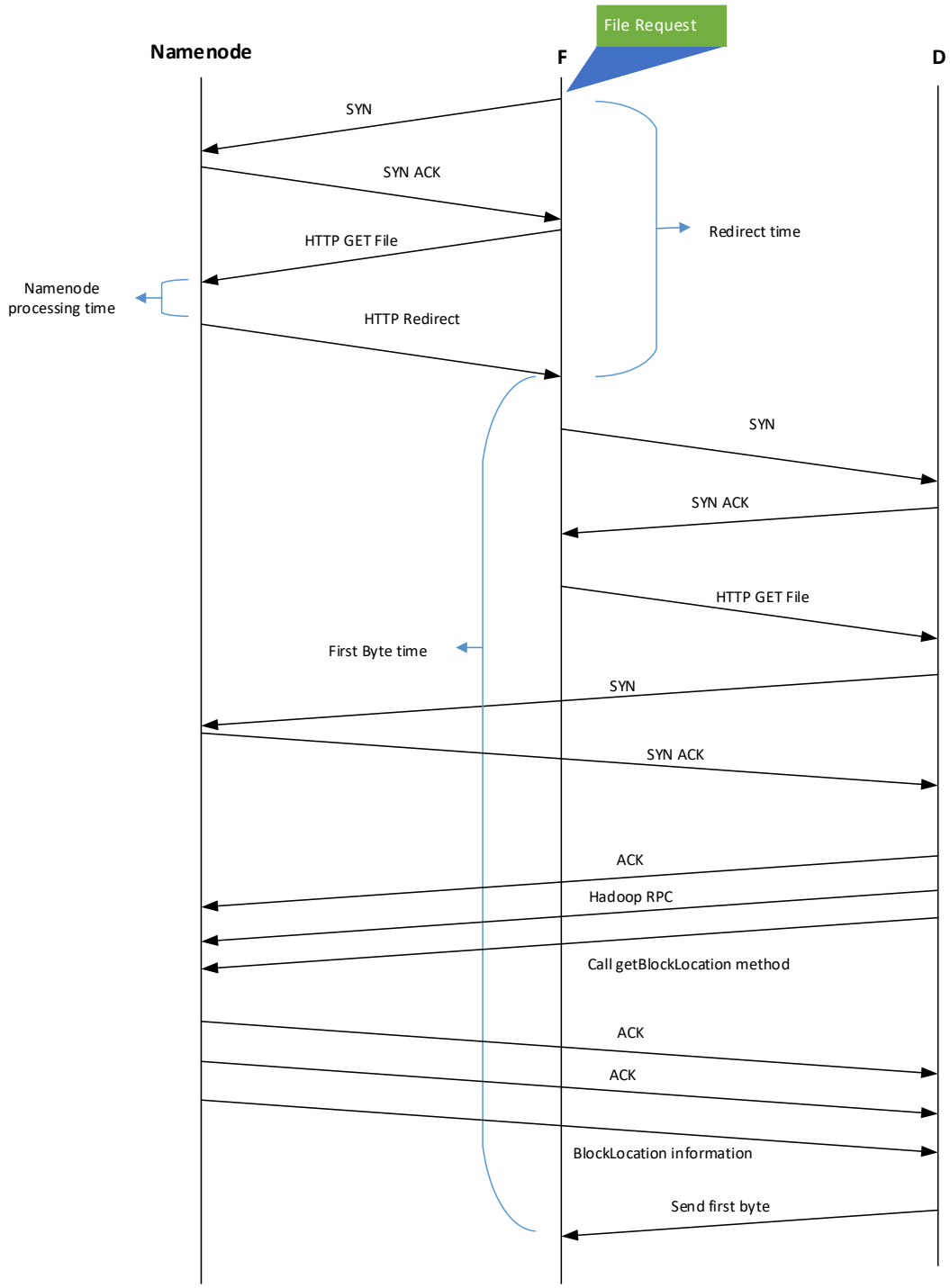


Figure 30 - WebHDFS activity diagram for read operation on filesystem

When a datanode requests a file for a read operation, it establishes a TCP connection with the namenode and sends its file request via a curl command – HTTP GET. The namenode then checks its records and, using equation 1 (in section 3.3) in the sorting strategy, decides which datanode to read files from – we call this the namenode processing time (see Figure 30). It then

sends a HTTP REDIRECT message back to the reader, telling it which datanode to retrieve the file from. The reader then establishes a TCP connection with the datnode present in the Redirect message, and sends HTTP GET request for the file. At this point, the file-holding datanode establishes a TCP connection with the namenode, and sends a Remote Procedure Call (RPC) request to it. This packet is followed up by another packet, with a serialized representation of the getBlockLocation method, with the requested file as argument. The namenode then runs the getBlockLocation method for the file and sends the requested information back to the node. When received, it sends the requested file to the requesting datanode.

All datanodes that are involved in the file transfer must establish a connection with the namenode in the first place. It works as a security measure to control the flow of data in the system. Due to this functionality, there is some induced delay associated with the arrival of the first byte at the read requesting node.

After the file write operation has finished, there is a configured wait of two seconds, before instructing the Android node to read the file it just wrote. At 6.9s the Android establishes a connection with the namenode with the intent of reading the file. It takes 0.8s to start the file transfer, which is explained due to protocol communication in HDFS read operation, represented in Figure 30. The transfer takes 0.3s. The current read requests' transmission rate, without considering the overhead due to TCP and HDFS handshakes, is about: $ReadRequest1\ Rate\ = (10\text{^}6\ x\ 8\ \text{bits})/0.3s = 26.7Mbps$

Comparing the time it takes to start writing, with the time it takes to start reading, we see that HDFS has more advantages in accessing data quickly, than writing to the filesystem. This was expected, since HDFS has been more optimised for reading operations. The difference is explained by the read operation not requiring the namenode to do various permission and filesystem checks.

After the first read has finished, the Android node's position is changed to (275, 137.5) in the NodeLayout file, where we expect the network latency emulator to set the new latencies between the networking nodes. Since we are using Pharos as our NCS, and because Pharos uses a converging algorithm, it takes time for the NCS to stabilize the changes that have been made to the node layout. As already mentioned, in section 5.2.1.2 we will discuss the performance of Pharos in terms of stabilizing time and location awareness accuracy.

After 90s of waiting, the Android node is instructed to perform three sequential read requests from its new position. This read request starts at the 98s mark, and has a transfer time of 1.6s.

These values are not obvious in Figure 28 but we have used precise values taken from the wireshark captures. HDFS read speed between datanodes is delayed by the slow-start approach used in TCP. Slow-start forces a low throughput for the first transmitted bytes. The number of bytes sent per RTT only increases when the sending nodes receives an acknowledge message from the receiver. In the first read request, the nodes were close so the maximum rate was reached fairly quickly. In the second read request, nodes are separated by a 177ms latency link. This severely delays the download speed, as the acknowledges take 177ms to reach the sending node. Due to this, throughput increase is slower. Calculating the transmission rate for the second read request we have: $ReadRequest2\ Rate\ = (10\text{^}6\ x\ 8)/1.6 = 5Mbps$

The second read throughput (5Mbps) is smaller than the first (26,7Mbps) because in the second read request, the source node and receiver node are much further apart.

After the Android node requests the file three times (106s), the 'hit threshold' is reached and replication is in order. The namenode then determines the closest datanode (TargetNode) to it (using equation 1 in section 3.3) so that this datanode can receive the extra replica. In this case, the selected datanode was node E, although it should have been node C. As mentioned, we will study the NCS' accuracy later on in section 5.2.1.2. The replication starts at 1.1s after the third read request read finishes, possibly explained by the block recovery feature (see Fig. 27) running every 1s. It takes 2s to transfer the file, which is longer than the previous 1.6s of transfer time for the previous read requests, for similar distances and file size. This happens because HDFS throttles the replication speed, to not interfere with cluster traffic when failures happen during regular cluster load. As such, we see a maximum of 130k bytes being sent per message.

At the end of the third read request from the new position, there is a configured wait of five seconds, which gives the system enough time to replicate the file to a datanode closer to the current Android position. Then the Android node requests the file one last time. At 110.8s the request is received by the namenode, and it verifies that the closest file is stored now on node E. Since node E and the Android node are now closer to the namenode, as well as closer to one another, the access time also decreases, from 0.8s to 0.6s (see HDFS read protocol in Figure 30). The transfer time is 0.4s which is similar to the transfer time of the first read request (0.3s), that have a similar distance between the source and reading datanodes.

This test scenario allows us to prove that both our data placement scheme and our NCS are working as expected. We can see this because the initial write request stores the files on the closest nodes to Android, therefore both the NCS and replication strategy work. Furthermore,

after Android moves position and reads the file three times, the system's replica management replicates the file closer by, once again proving the NCS working. Finally, the sorting strategy is proved to work as expected, when the Android reads the file for the last time, and it gets retrieved from the close-by node that received the extra replica. Consequently, the context awareness implementation is also working.

5.2 Performance Tests

Now we will study the system's performance metrics by carrying out some stressing tests. We then discuss the results. We split our performance tests into two main categories:

1. Network Coordinate System
2. Data Placement Scheme

We test the NCS for accuracy, because it is the backbone of our location awareness feature. We then put the data placement scheme to the test by going through a series of test scenarios, each with the goal of highlighting a different performance metric that the system provides.

5.2.1 Network Coordinate System

We want our datanodes to provide the namenode with their correct position, so the latter can replicate content to where it is "supposed" to be. So, it is important to determine how accurate the NCS are, at calculating distances between datanodes using their NC.

We describe the test scenario that has beencarried out for both NCS implementations. We then compare the obtained results from the two NCSs.

Test scenario

We propose to use a node layout as illustrated in Figure 21 (in section 4.2.1). The idea here is that nodes W and Z have a file, which the Android node will be requesting. We move the Android node across the latency-map from (220,0) to (330, 220). We start at (220, 0) and add iterations of 27.5ms to the y component until it reaches (220, 220). After that we iterate the x component with a positive increment of 27.5ms, and decrease the y component all the way down to 0. We cycle this sweeping pattern until the Android node reads position (330, 220). At every position Android is at, it retrieves W, Z and Android's NCs as well as the latency links between W-Android and Z-Android. We then verify if the NCS determines the closest node correctly, by comparing the lowest calculated distance using the retrieved NC, with the lowest latency link between the two connections.

In this way, we effectively see the percentage number of times that the NCS accurately determines the closest node to retrieve the file from. We follow-up by reviewing this test scenario for each NCS.

5.2.1.1 CAN Topology

We found, that using CAN topology, a variety of factors ended up dramatically influencing the system's accuracy in our test scenario. Those factors include: changing the latency discretization intervals used for each level; changing the number of levels; changing the relative position between the landmarks as well as changing the number of used landmarks.

We have experimented with three landmarks fixed in the same position, as seen in Figure 21 (in section 4.2.1). Table 11 shows the number of levels and the latency discretization intervals that we have used in our tests.

Table 11 - CAN Topology latency discretization for different number of levels

Number of Levels	Latency discretization (ms)
3	0-30; 30-100; 100+
4	0-30; 30-50; 50-75; 75+
5	0-30; 30-50; 50-75; 75-100; 100+
6	0-30; 30-50; 50-75; 75-100; 100-120 ; 120+

The Figure 31 shows the performance of the CAN topology for the described scenario, using the number of levels for the CAN Topology that are in Table 11.

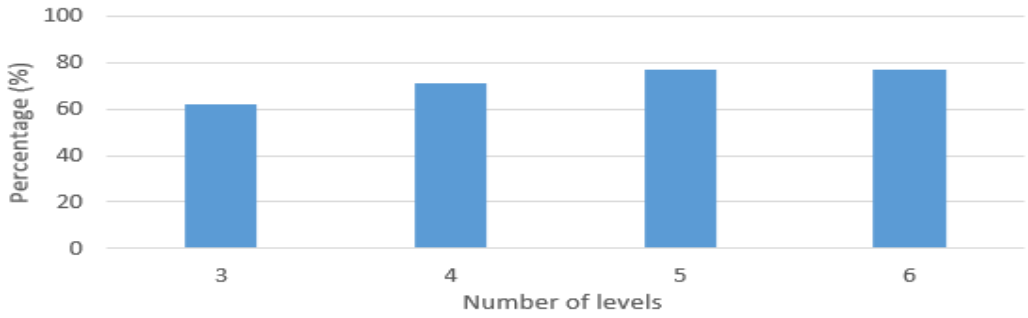


Figure 31 - CAN topology performance graphic using different number of levels

As we can see in Figure 31, we cap our success at 78%, using five and six levels in the NC. This means that, for all the positions that Android passed through, 78% of the time, the NCS calculated the closest node accurately. Our standard three level proposal was accurate only 61% of the time.

The clustering effect that the CAN topology uses, impacts negatively the accuracy of the system. NC are attributed to zones, which nodes fall under. This introduces error in calculating distances between nodes. To demonstrate this problem we will study a scenario, shown in Figure 21 where Android is in position b), that evidentiates the systems' inaccuracy.

In Figure 21 b), we can see that the Android node is separated by 74ms from the W node, and 58ms from the Z node. So the Z node is clearly closer to the Android. By calculating their distance using their network coordinates, we should arrive at the same conclusion.

We now calculate the distance between the Android-W and Android-Z nodes using their NC, using equation 5 (in section 4.2.1). The nodes' NC are: W(2,2,3), Z(2,1,3), Android(3,2,2).

$$distance_{Android \to W} = \sqrt{(3-2)^2 + (2-2)^2 + (2-3)^2} = \sqrt{2}\,;\; distance_{Android \to Z}\ \sqrt{3}$$

So the evaluated distance from Android-Z is bigger than the distance Android-W, which means node W is closer to Android. Looking at Figure 21 b), we can see that that is not the case.

5.2.1.2 Pharos

We have used a node layout similar to the one used with CAN topology, but with more datanodes. We don't expect the results to be influenced by this change. Figure 32 shows the node layout that has been used for testing Pharos.

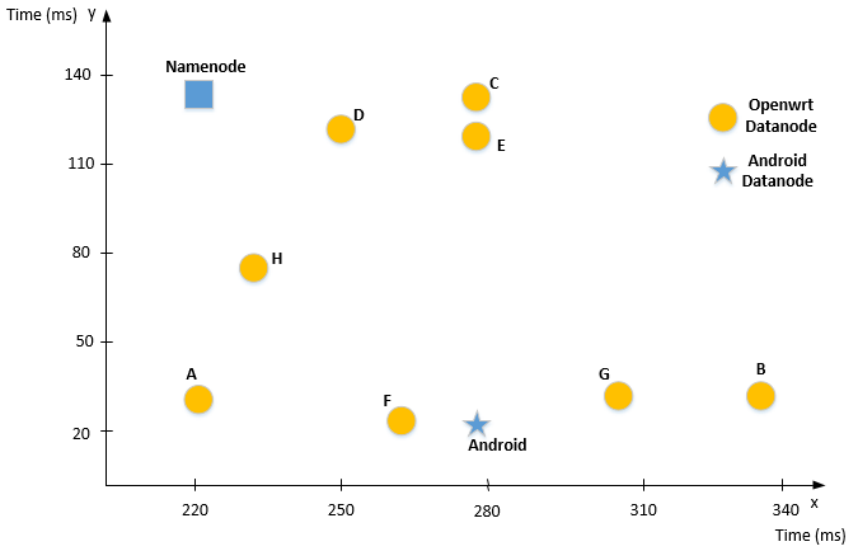


Figure 32 - Node layout with Pharos

Pharos clients iterate periodically – 'Pharos client iteration time', converging their NCs to an optimal solution for predicting latencies between nodes with accuracy.

We have run the same test scenario as the one to test the CAN topology, with a slight modification. We now introduce a delay, which we call the 'PositionDelay', at each position where the Android node arrives at. After the 'PositionDelay', the NC are retrieved for Android, F and G nodes, as well as the latency links between them.

In this way it is possible to study the NCS' accuracy for different number of Pharos client iterations at each position the android is at. We basically study the system's accuracy for different converging periods, which is relevant given the mobile environment portayed in the

current work. We expect to see higher accuracy with higher values of 'PositionDelay', because it allows the Pharos clients to run more iterations and thus converge to a better solution.

We have set the Pharos client iteration time to six seconds, and change the 'PositionDelay' between tests. This gives us different number of iterations per position. For example: with a pharos iteration time of six seconds, and twenty seconds of 'PositionDelay', we have 20/6 = 3,3 iterations per Android position, which rounds down to three iterations per position. Figure 33 shows a graphic of the Pharos performance test for different numbers of pharos iterations per Android position.

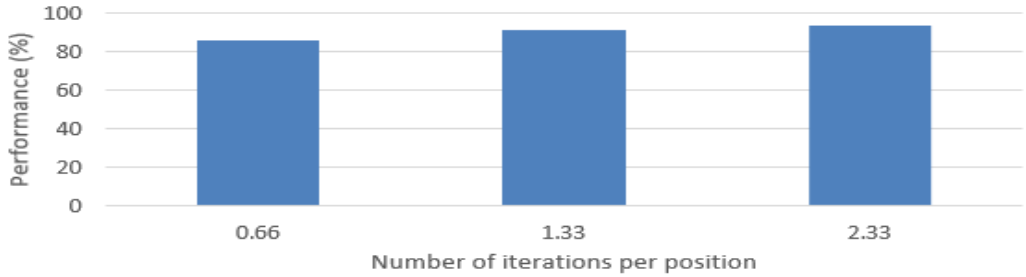


Figure 33 - Pharos performance accuracy test based on number of iterations per position

In our tests, we cycled the 'PositionDelay' between four, eight and fourteen seconds, which represent the 0,66 ; 1,33 and 2,33 iterations per position, respectively.

As seen in Figure 33, the more iterations the algorithm performed at a certain position, the higher the overall performance accuracy. This behavior was expected. We can see that even with 0,66 iterations per position, the system's accuracy is around 86%, which is higher than the 78% we registered from the best CAN topology test, in Figure 31. If enough time is given, the system's accuracy reaches close to 95%.

Comparing Pharos and CAN topology performance, we see that each have their own advantages. Pharos can achieve better accuracy if it is given enough time to stabilize its iterative algorithm; while CAN has no stabilizing time, but it is less accurate. With Pharos, the NCs converge by having clients cooperate with each other. This makes it harder for a user to forge their NC for selfish purposes. Future work could be made in this regard, to have the namenode verify if received NC make sense. CAN topology fails in this regard, as the NCs are calculated locally on each node, so any user can forge their position.

In a mobile cloud network, nodes are constantly changing position. This could be a setback for Pharos, as the stabilizing time is not as controlled as in our test scenarios. But since our system is designed having the idea of transferring data between mobile nodes and fog nodes that are close-by, an argument could be made that mobility in our system, should be interpreted as

mobile nodes being online in one position, and later on, online in another position (nomadic behavior). This differs from the usual network mobility concept of a mobile node changing its position. In a system using the mobility characteristics that we describe, Pharos ends up being the better candidate for keeping an accurate location awareness map. The 'PositionDelay' time is more controlled when the mobile node spends most of its time within the same area.

5.2.2 Data Placement Scheme

In this section, we put our system through a series of heavy workload tests, and see how it reacts. We will analyse performance metrics such as system responsiveness, available bandwidth at the backbone, and fairness in energy depletion as the content is distributed within the network.

Three different test scenarios have been created to test the performance metrics mentioned above. The three scenarios are:

- Small Files Scenario – Tests the system in terms of responsiveness in accessing data.
- Big Files Scenario – Tests the system in terms of available data rate when transferring big files.
- Battery Level Scenario – Tests the system's fairness in distributing content according to nodes' energy levels.

Pharos has been used as the NCS for the following performance tests, as it showed the best results in terms of location accuracy in section 5.2.1. For every node layout used, we have waited for the NCS to converge to an optimal solution, so that its' accuracy is near 100%. We follow up by detailing each of these scenarios and by discussing the results obtained.

5.2.2.1 Small File Scenario

The goal of this scenario is to see how the system performs in terms of system responsiveness, when accessing data. The network is flooded with file requests from every node, and for every request, the time it takes to receive the first byte is registered, as well as total download time. We expect to see the system's reaction to the successive file requests made from nodes, in particular how they are replicated throughout the cluster. Replication should bring data closer to the end-user, which should give lower delay for the first byte being received, as well as smaller download times. We have used small file sizes in this scenario because the system responsiveness is more evident.

We follow-up by detailing the test scenario. We then walk through the configurable replication parameters that have been used in this scenario. Finally, we present and comment the obtained results. The node layout that has been used for this scenario is seen in Figure 34.

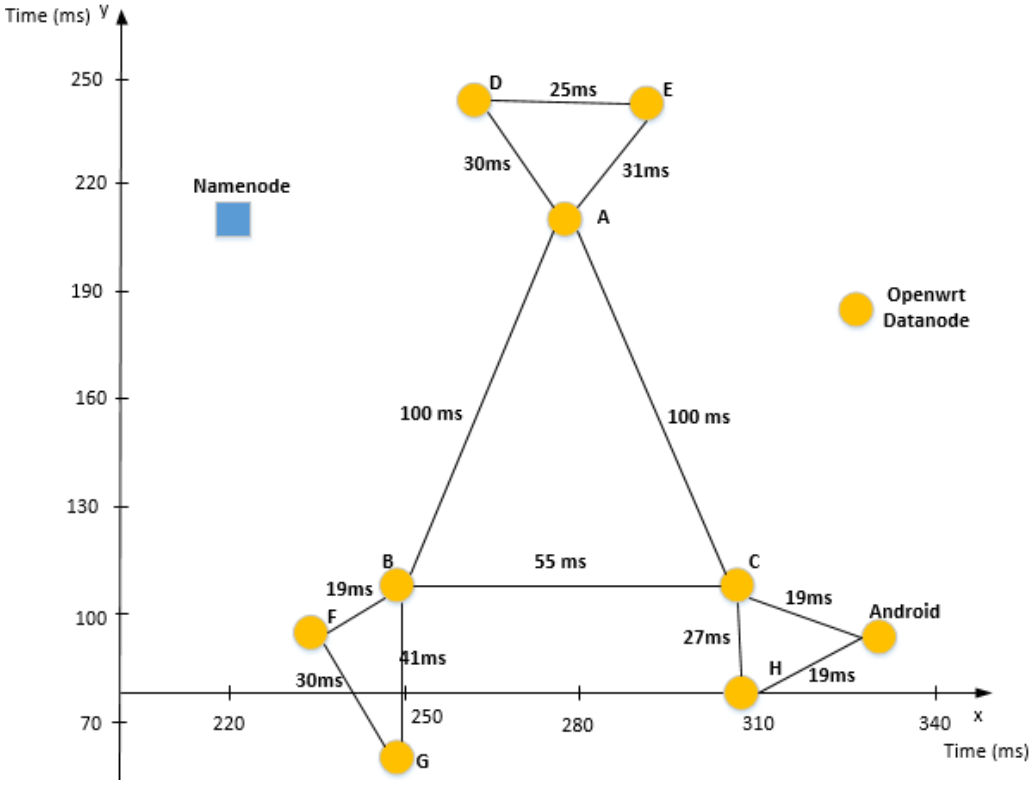


Figure 34 - Node layout for small file test scenario

The nodes have been placed into simulated edge networks, separating the A, B, C nodes by 100ms and 55ms latency links. This translates into at least 200ms and 110ms RTT links between nodes on different edge networks. In each edge network, nodes are placed close to each other.

Test scenario

In this scenario, every datanode stores five files of 1MB each in the filesystem. Since there are nine datanodes, we end up with a total of forty-five files in the HDFS. Files get stored in the closest two datanodes according to our configuration of the parameter 'NrDefaultReplicas' in Table 10. After this step, we proceed to stress-testing the system, by instructing each datanode to retrieve all the files in the filesystem, in the same order. This task is then looped a configurable amount of times, on each datanode. We will be referring to this previous task as loop attempts. Each datanode requests forty-five files per loop attempt. Note that the 'hit threshold' is set to three, so replication will only happen if there are at least three loop attempts.

Each curl command (file request) made on a datanode, has special options to log the time it takes to receive the redirect request, receive the first byte of the file, and the total download

time of the file (see Figure 30). Every request on every node has been logged. The collected data was then processed to produce the evaluation graphics shown in the next pages.

Configurable Replication Parameters

The goal is to study how the configurable replication parameters influence the system's behaviour in terms of system responsiveness, over time. The replication parameters that have been used were: 'k' and 'β' in equation 2 (in section 3.3), and 'Δtn' and '∝' in equation 4 (in section 3.3). Parameters 'k', 'β' and '∝' are specified in a configuration file, at the beginning of the test. Parameter 'Δtn', which indicates a file's hit interval, is controlled by instructing the datanodes to start their loop attempts, with 'Δtn' interval between each other.

Here, each datanode performs ten loop attempts, meaning each datanode requests all files in the HDFS ten times. This allows for perceiving how the configurable replication parameters affect data responsiveness.

Results and Evaluation

Faster delivery of data is expected, when there are higher levels of replication in the system. But it is important to detail the system overhead and know how it reacts to the different replication parameters. We follow up presenting relevant graphics of the obtained results for the current test scenario. We look to interpret these graphics and reach some interesting conclusions.

- Replication on the third loop attempt

Figure 35 shows the average redirect (RD), FirstByte and transfer times, of a request made, per loop attempt. First, we explain how these times relate to one another, and then detail how the averages that are displayed in Figure 35, have been calculated.

These times are retrieved for each file request, on every datanode. They are independent from one another. Meaning that the transfer time, is calculated by subtracting the total download time with the first byte time as well as the redirect time.

Regarding transfer time, each value in Figure 35, represents an average of the transfer times for all the files requested in that loop attempt, for a single datanode, averaged between all datanodes. Each datanode requests forty-five files per loop attempt, and there are nine datanodes retrieving files, therefore each value presented in Figure 35 is an average of 45*9 = 405 values. In this way, an overview of the system's responsiveness in accessing data is displayed, over several loop attempts. This method, that has been used to calculate and display the average times shown in Figure 35, is used throughout the remaining graphic results in the

current work. The replication parameters that have been used in Figure 35 are: Δtn=2s; β=1000; k=1.0.

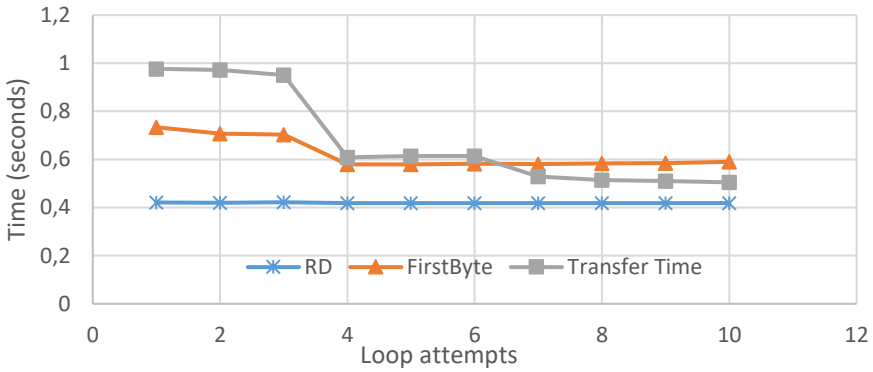


Figure 35 - Filesystem average download time progression

The average RD time is constant because it results from the communication between the requesting datanode and the namenode (see Figure 30). Since nodes don't change positions, the average RD time remains the same.

On the fourth loop attempt, the average transfer and first byte received times, decrease. This indicates that the system replicated data, by spreading files throughout the edge networks, bringing data closer to the requesting datanodes. This decrease on the average access time to data on the fourth loop attempt, reveals an increase in system responsiveness.

On the seventh loop attempt, the transfer time dips below the first byte time, which can be explained by files being retrieved from datanodes that are far away from the namenode (see Fig. 30).

- Namenode processing overhead

Figure 36 shows the average time that the namenode spends on processing file requests, in each loop attempt, with a confidence interval (CI) of 95%.

The processing time for each file request has been retrieved, by subtracting the RD time with 2'RTTs associated to the latency link between the requesting node and the namenode (see Figure 30). Each processing time value in Figure 36 results from the average of the collected processing times for all the file requests in that loop attempt, for the same test that was carried out previously in Figure 35.

On the third loop attempt, the processing time increases, as well as its variance. This increase can be explained because the first datanodes requesting files on the third loop attempt, make the namenode replicate files to other datanodes. By the time the last datanodes start requesting the files on their third loop attempt, the namenode has already received confirmation of

successful replication and is tasked with updating its' cache with the new file locations, which is time consuming (although not much).

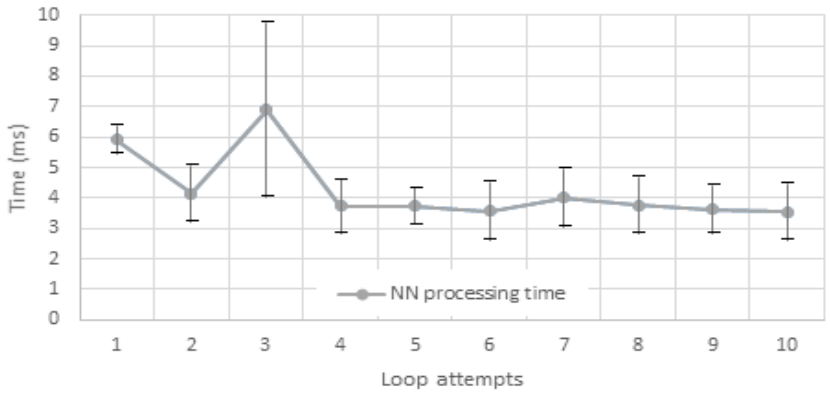


Figure 36 - Average time spent processing a file per loop attempt on the namenode

Looking at the average transfer time for a file in the system on the first loop attempt (1s) in Figure 35, one can conclude that the namenode processing time (6ms) is negligible in comparison with the former latency. This partially proves that the overhead introduced by our proposal in the namenode is rather small.

Now we will study the system's behaviour when changing each of the configurable replication parameters. We also present the equations proposed that use these parameters:

$b = k.a + \beta \frac{1}{P}$ (s) **(7)** $P = \frac{1}{\text{Avg } \Delta t_n}$ (s^{-1}) **(8)**; $\text{Avg } \Delta t_n = \Delta t_n . \propto + (1 - \propto).\text{Avg } \Delta t_{n-1}$ (s) **(9)**

- <u>System's responsiveness to node proximity</u>

Here, we will study how the system is influenced by changing node proximity. Looking at the first parcel in equation 7, 'a' refers to the distance between the second closest node to the reader, and itself (the reader). Node mobility is not accounted for in this scenario, so 'a' remains the same. In this way, the node proximity will be studied by isolating the influence of parameter 'k' in equation 7. It is expected that by configuring lower levels of 'k' (higher node proximity), data being requested needs to be closer to the reader, reflecting higher data proximity requirements. In this way, replication intensity should increase. Figure 37 shows for β=800, Δtn=2s and α=95%, the average transfer time of a file in the system, per loop attempt, for different values of 'k'.

Figure 37 shows that when the system was configured with k=3.0 - low node proximity – the average transfer time remained the same throughout the test. This is expected, because files are configured to replicate only when the reader-sending nodes are very far apart (further apart than the two furthest-distanced nodes used in this test).

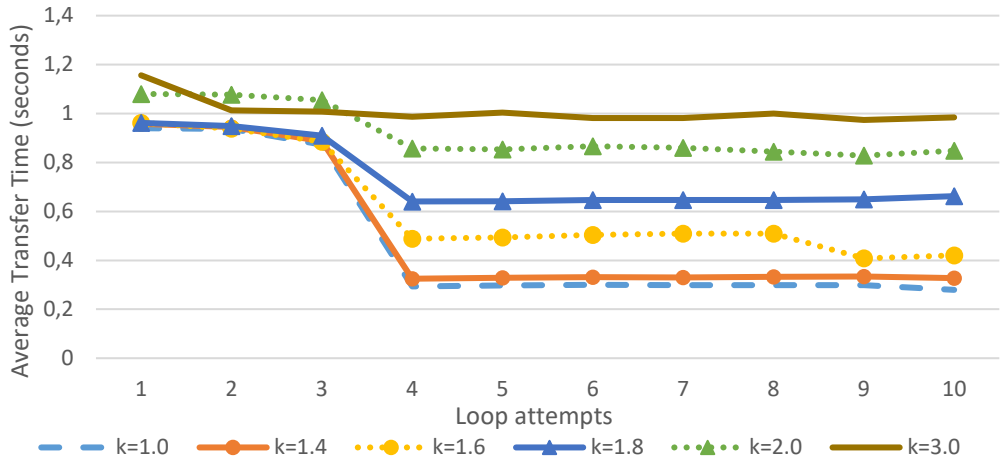


Figure 37 - System responsiveness to node proximity by varying 'k'

On the other hand, with higher node proximity (e.g. k=1.4), data is configured to be closer to the readers, therefore data is more easily replicated. This reflects faster access times to data.

- System's responsiveness to popularity

Here, we will be studying how the system reacts to changes in popularity level. This will be done by first isolating the influence of parameter 'β', in contributing to the 'b' search radius, and then studying the consequences of varying Δtn, in equations (8) and (9).

Looking at the second parcel of Equation 7: lower values of 'β', contribute to creating smaller search radiuses. Therefore, files do not have to be so popular, for replication to occur. Figure 38 shows for k=1.4, Δtn=2s and 'α'=95%, the average transfer time of a file in the system, per loop attempt, for different values of 'β'.

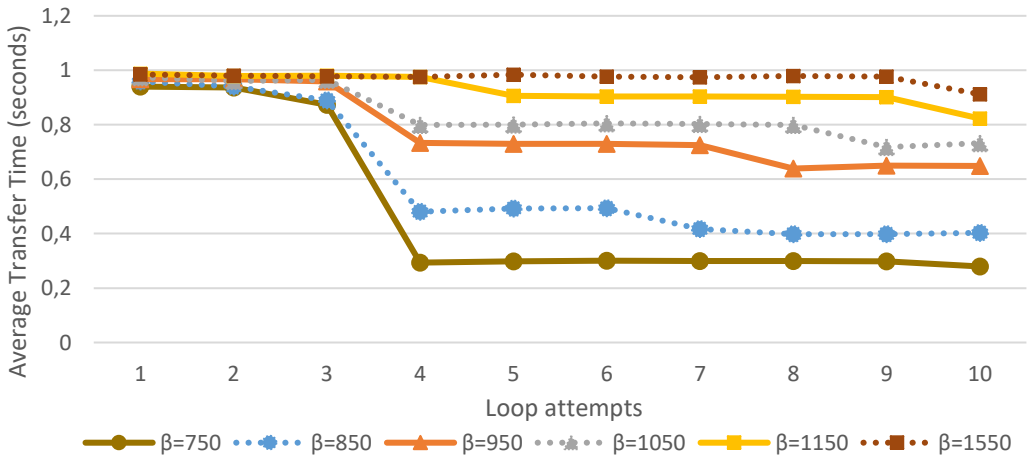


Figure 38 - System responsiveness to data popularity by varying β

Figure 38 shows that lower values of 'β' give quicker access times to data. This happens because the replication requirement regarding popularity is soft, and so data doesn't need to be very popular for it to get replicated. Consequently, data are replicated in a more intensive way,

and latency in accessing data is overall decreased. By giving higher values of 'β', the replication requirement regarding popularity are stricter, meaning data needs to have higher popularity levels in order to get replicated. This can be seen in Figure 38, for values of 'β'=1550, where download times remains practically the same throughout the test, meaning there has hardly been any file replication.

Here, we will be studying how the system reacts to different hit intervals in requesting files. This has been done by varying parameter Δtn, while setting the other replication parameters to constants. We expect to see lower hit intervals reflect more popularity, and therefore more replication in the system. Figure 39 shows, the average transfer time of a file in the system, per loop attempt, for different values of Δtn, when using k=1.8, β=1550 and α=95%

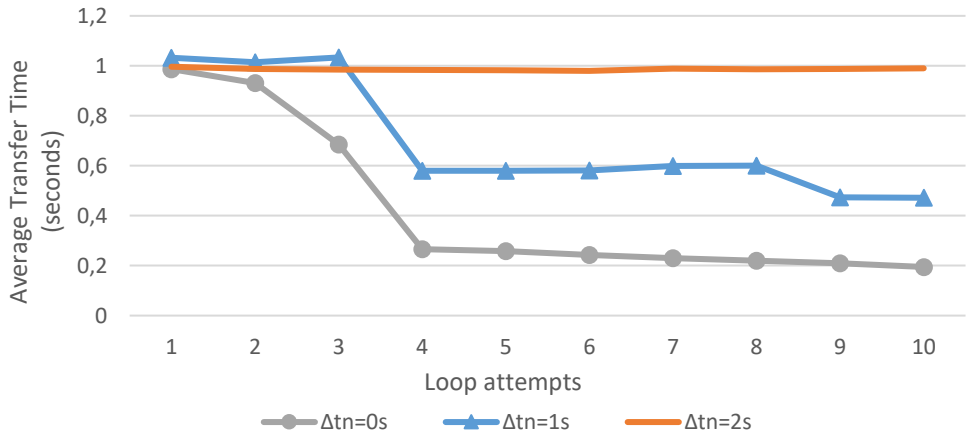


Figure 39 - System responsiveness to data popularity by varying Δtn

Figure 39 shows that for hit intervals of Δtn=0s, data quickly gets replicated in the system. On the third loop attempt, data is partially retrieved from nodes that have already received replicated files. What is seen is expected, as higher hit intervals (e.g. Δtn=2s) reflect less interest in files, which translates into less popularity, and therefore less replication and higher delays in accessing data.

- System's responsiveness to popularity history

Here, we will be studying how the system reacts to changing the popularity history reactivity parameter - 'α'. Looking at equation 9, it can be seen that using higher values of 'α', the system calculates average access frequencies by giving more weight to the last Δtn. This means that the system becomes more reactive (to the last Δtn), when calculating popularity (in equation 8).

We follow-up studying the influence of 'α' in calculating popularity (see Figure 40) and its consequences in calculating the 'b' search radius (see Figure 41) in the current scenario.

Figure 40 shows the popularity levels of a file, using different values of 'α', when Δtn=2s. There are ninety samples because there are nine datanodes requesting the file over ten loop attempts.

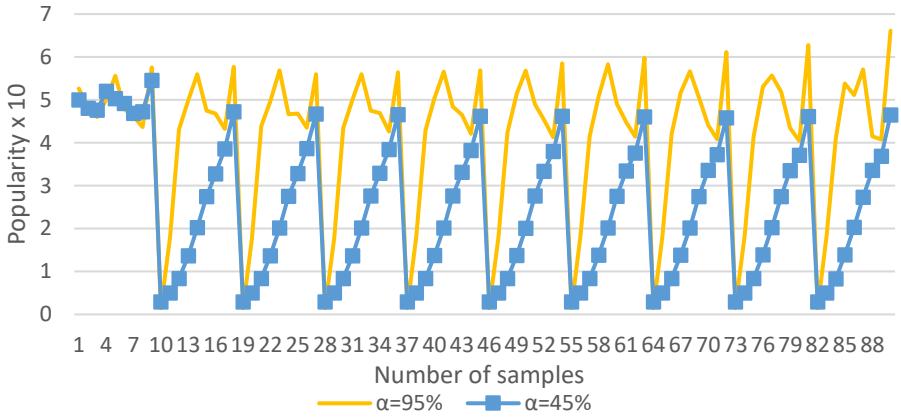


Figure 40 - Influence of 'α' in calculating popularity

We'd expect popularity levels to be constant, since Δtn had been configured to remain the same throughout the test. However, that did not happen. We will first calculate the popularity level that was expected, and then detail why the results are different than expected. This will lead to understanding how 'α' has influenced the popularity calculations shown in Figure 40.

The constant expected popularity level has been calculated by first calculating Avg Δt_n, using equation 9, with values of Δtn=2s and α=19/20=95%:

$$\text{Avg } \Delta t_n = 2.\frac{19}{20} + \left(1 - \frac{19}{20}\right).2 \qquad \text{Avg } \Delta t_n = 2\left(\frac{19}{20} + \frac{1}{20}\right) = 2s$$

And then substituting in equation 8 we get: $P = 1/2$. We have calibrated this to 5 (x10) to avoid working with decimal numbers. So, the expected popularity level measured throughout both tests would be five.

Given that Δtn=2s, the time spent between the first time a file is requested and the last time it is requested, for a specific loop attempt, is 18s (9 datanodes * 2s). Looking at Figure 40, it can be seen that during nine requests, the popularity does in fact register the expected value of five, for both values of α. The tenth attempt however registers a higher Δtn, due to a waiting time (WT) that has been configured between file requests, on each datanode. It has been set to 2s. This WT has been introduced to control congestion on the namenode. Since each datanode requests forty-five files, it takes 90s to finish a loop attempt. So, each file registers a Δtn of 90s – 18s = 72s, every tenth attempt. Looking at equations 8 and 9, this explains the dip in popularity levels every tenth request, in Figure 40.

This allows us to study the influence of parameter α on calculating popularity, according to the last measured Δtn. Looking at Figure 40, it can be seen that when using a higher reactiveness

value for α (e.g. 95%), the system quickly recovers to the expected popularity level. This happens because a big weight (95%) is given to the last Δtn, which is set to 2s. On the other hand, using lower values of α, the system became less reactive to the last Δtn. In this way, the past popularity history is more relevant. As such, popularity dips every tenth request, and the following nine requests, only slightly increase the file's popularity.

In Figure 41 shows for each file request, the RTT contribution of each parcel in equation 7, in calculating the 'b' radius, for different values of α, with k=1.4; a=50ms; β=850 and Δtn=2s.

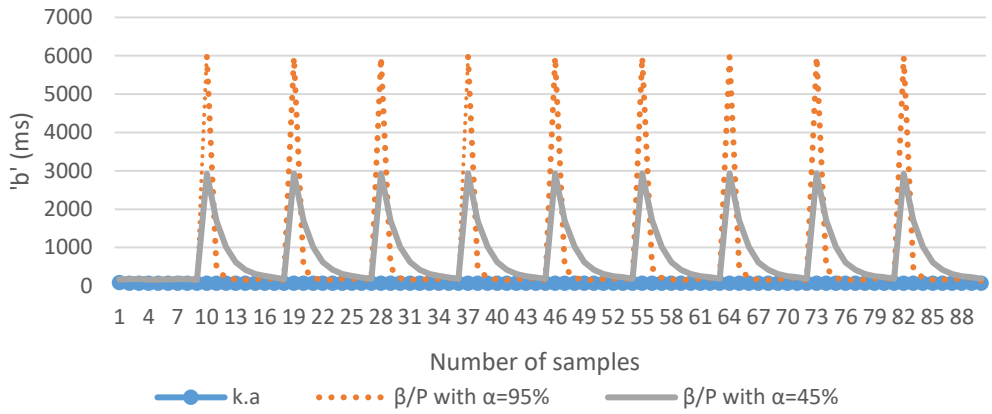


Figure 41 - 'b' search radius RTT contributions by varying α

Looking at Figure 41, it can be seen that with α=45%, that the 'b' search radius varies in proportion to the popularity history registered for each file request. On the other hand, α=95% shows high 'b' search radiuses only every tenth attempt, as expected. Although it isn't perceivable, the average 'b' associated to node proximity in this test is around 250ms.

Although the test scenario's particularity has enabled showing how α influences the system, it has also introduced a limitation in demonstrating the extent of this influence. Parameter α should be configured to a value that reflects the importance given to popularity history. When using a very high value of α, then popularity history basically isn't considered, which is nowhere near optimal. When using a very low value, then it might be too hard to get files to reach popularity levels sufficient for replication.

- <u>System responsiveness when accessing files at distinct edge networks</u>

In Figure 42 we compare access times for data that is retrieved from edge networks at several distances, for the same test that was carried out in Figure 35, which showed medium replication. The edge networks that have been considered, from closest to furthest, have the following datanodes: Close - (B, F, G); Medium – (H, C, Android); Far – (A, D, E).

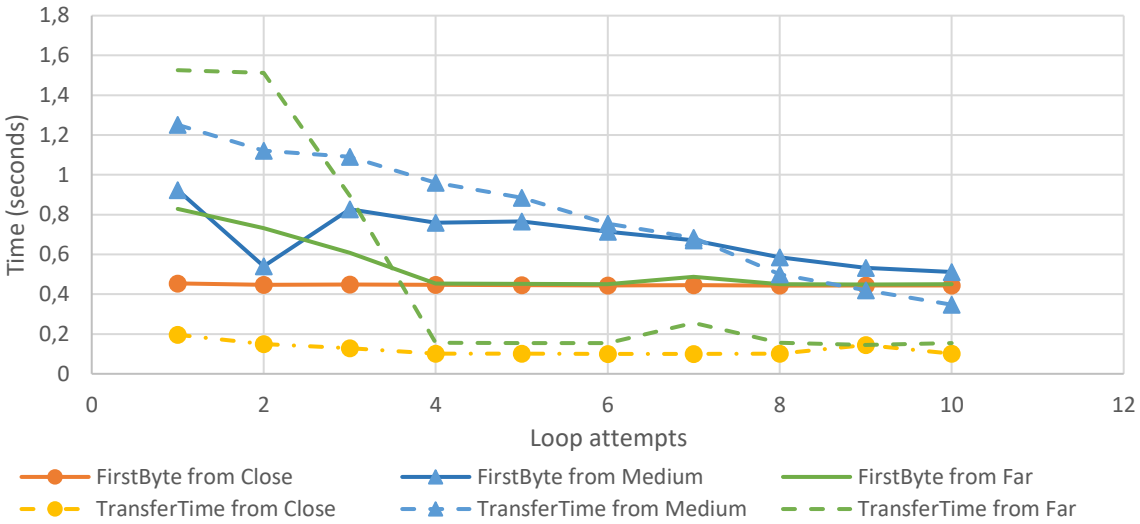


Figure 42 - Data access times for files on different edge networks

Figure 42 shows that files belonging to the closest edge network to node F, have been retrieved the fastest, at around 0.2s. Files from the medium distanced network present the second fastest access time, at around 1.25s. Files from the edge network that is the furthest apart, have the slowest access time, at around 1.5s, as expected. The reason the medium distanced network starts off with a higher first byte time than the furthest-away network, is due to the fact that the medium distanced network is located furthest away from the namenode. Recalling Figure 30, contact between the sending node and the namenode must be established in a read operation, therefore accessing data from a datanode closer-by does not necessarily mean faster access time, as the sending datanode may be further away from the namenode.

As mentioned, this test showed medium replication (see Figure 35). This is demonstrated on the third loop attempt, where files from the furthest network started getting replicated closer to node F. However, files from the medium distanced network were replicated much more slowly.

This test shows that for a configured set of replication parameters, the system is capable of replicating data according to the latency distance between the reading node and the requested files' location. In this specific test, the furthest-away files to the F node were all replicated in close-by nodes; while the medium-distanced files were replicated with much less intensity.

Overall, the replication parameters can be configured to replicate data according to the files' popularity. We have limitations in this scenario due to the low number of cluster nodes we work with. However, we are still able to verify responsiveness gains in our tests.

5.2.2.2 Big File Scenario

When transferring big files, eventual gains in system responsiveness are negligible. The time it takes to download a big file from a faraway location is pretty much the same as it takes to

download a file from a close-by location. Available bandwidth is a much more relevant factor in the time it takes to transfer big files.

In this scenario, we want to prove that the system can bring advantages in terms of transfer speeds. In the networking world, nodes reside on different networks, and the backbone connection is what connects these networks. Nodes located within an edge networks usually have fast connections (high bandwidth) amongst them. Nodes on different networks have their data pass through the backbone connection. This connection is a bottleneck due to all the traffic that must pass by it, and as such it is much more limited in bandwidth.

We will be flooding the network with file requests and then studying how long it takes to fulfil all those requests, for different levels of replication in the system.

By replicating data to the edge, the edge nodes can retrieve content faster with their fast connections. Also, the backbone bandwidth is made available for other data transfers.

Test scenario

This test scenario used here is similar to the small file test scenario. The only difference is that each datanode stores two files of 10MB each, in the filesystem instead of five files of 1MB each. The datanodes are then instructed to retrieve all the files from the filesystem a configurable number of times. In order to emulate the network conditions of the above described scenario, a new network has been configured, as shown in Figure 43.

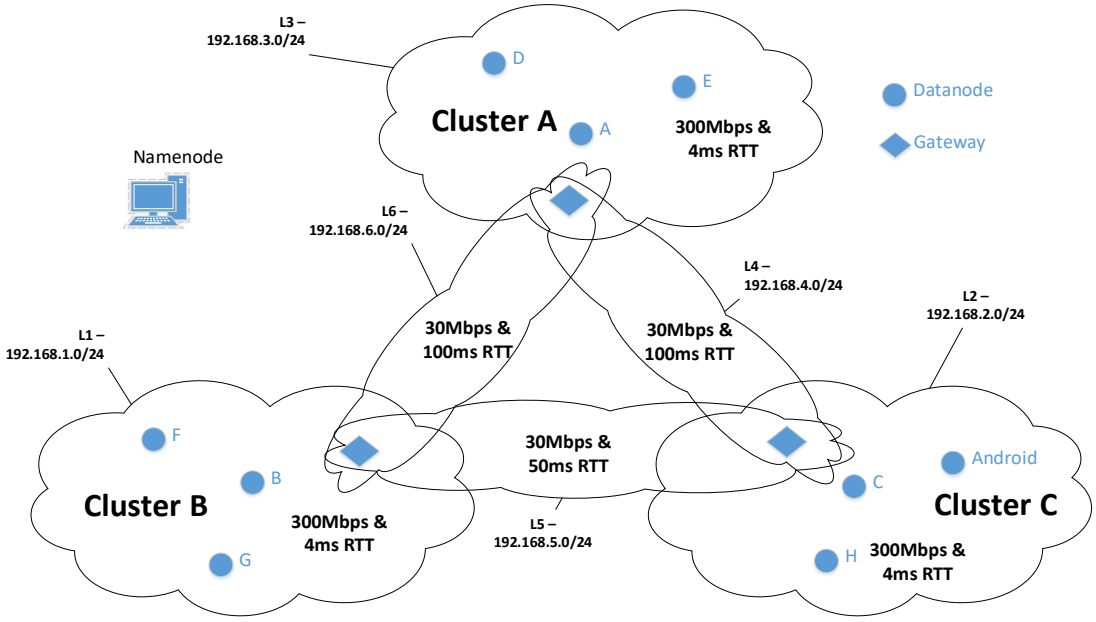


Figure 43 - Network configuration for big file test scenario

The nodes have switched from using a single host-only adapter to communicate with each other, to using internal networks. Each datanode is on a different internal network – L1, L2 and L3,

which has a different associated network. These networks emulate edge networks. Connections between nodes on the same network have their bandwidth set to 300Mbit. Each of these networks have a gateway machine, which forwards traffic to machines on different edge networks, through their gateway device. Connections between gateways emulate backbone connections. These connections have their bandwidth set to 30Mbit (10 times less than edge connections), by using tc netem. Each one of these gateway connections is an internal network. As an example, the gateway on cluster A, has network interfaces that communicate with nodes on cluster A, nodes from cluster B and from cluster C.

Figure 43 doesn't show the network connections to the namenode machine, to avoid unnecessary confusion. The links from the gateways to the namenode machine have high bandwidth (not a concern, as data is not transferred, only control information). The RTT latency between the namenode and the datanodes is also not important, as its impact on the reading throughput is negligible due to the size of the files being transferred.

Configurable Parameters

The replication parameters that have been tested are 'β' and 'k'. In addition, different values have been set for the time that datanodes spend waiting between file requests – WT. This time has been used so that congestion did not interfere with our results. The datanodes have been instructed to run ten loop attempts, similarly to the small file scenario.

Results and Evaluation

It is expected for data to get delivered faster when the replication parameters are configured for intensive replication. We will be studying the impact of these parameters for this current scenario. We follow up presenting relevant graphics of the obtained results for the current test scenario. The graphics are then discussed.

- System's responsiveness to network load

As previously explained, parameter 'WT' has been introduced to control the amount of congestion in the system. By having the replication parameters remain the same, and only changing WT, an opportunity is created for studying the test scenario's limits in terms of bandwidth and congestion. Figure 44 shows two tests carried out varying only parameter WT, with k=1.2, β = 750, Δtn=2s, 'α'=99%. Parameter α has been increased to 99% to increase the reactivity in the system, because here, files register a much higher Δtn every tenth request, than with WT=2s (value used in small file scenario).

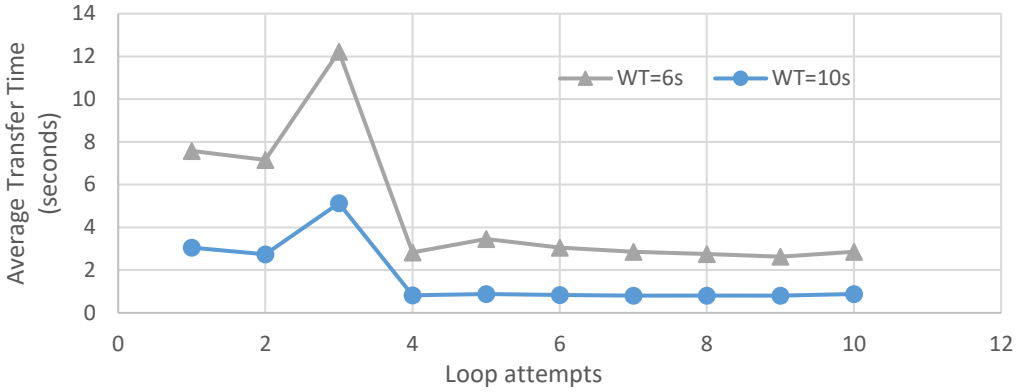


Figure 44 - System's responsiveness to network load by varying WT

WT has also been set to 3 seconds, but the system experienced too much congestion and stopped the majority of the file downloads halfway through. Figure 44 shows how the different WTs impact the average transfer time in the system. When WT has a lower value, the communication links in the system have less time to transfer data. In this way, the system retains more congestion, and therefore transfer times are higher.

- System's responsiveness to popularity

Here, we want to study the immediate impact of popularity in the system, by changing parameter 'β', and keeping the other replication parameters as constants. Figure 45 shows the average transfer time for a file for all datanodes, per loop attempt, with k=1.2, Δtn=2s, α=99% and WT=6s.

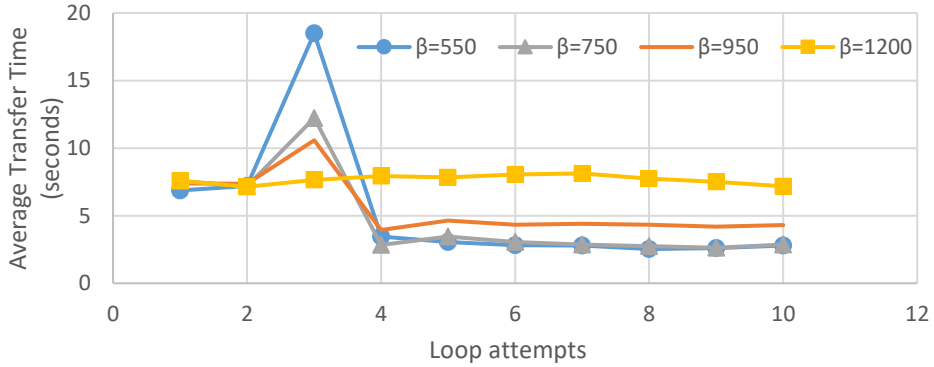


Figure 45 - System's responsiveness to popularity by varying 'β'

Figure 45 shows that with lower values of β (e.g. 550), there is more replication in the system. Higher replication levels in the system are made evident on the third loop attempt, as data getting replicated takes up bandwidth that was being used by the file transfers. In this way, the file transfers are left with less bandwidth and therefore their transfer time are increased. Making the system replicate data with more ease just so data can be accessed faster is not so straightforward though. For β=750, the replication requirements are stricter, and therefore there

is less replication. This is made evident by the lower average download time for a file in the third loop attempt. From the fourth loop attempt onward, the time it takes to access the data is the same as with the β =550. This happens because with β =550, data is being over-replicated to edge networks.

- System's responsiveness to node proximity

Here, we want to study the immediate impact of node proximity in replicating data and the latency associated in retrieving data. In this way, parameter 'k' has been changed while the other replication parameters were kept the same. The parameters that have been used to obtain the results present in Figure 46 are: β =700, Δtn=2s, α=99% and WT=7s. The WT has been increased to 7s due to disparity in obtaining results.

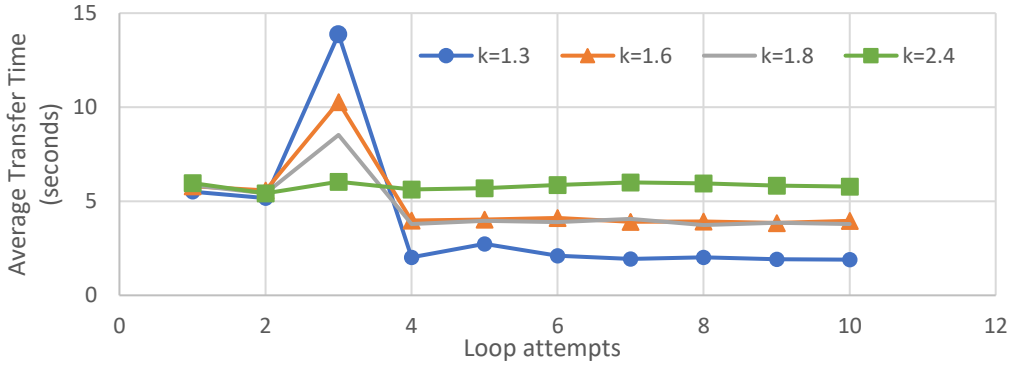


Figure 46 - System's responsiveness to node proximity by varying 'k'

Figure 46 shows that using lower values of 'k', associated to higher node proximity, empowers the system with higher replication intensity. This can be seen by the higher average download time on the third loop attempt, associated to heavy data replication. This is expected, as lowering 'k' means that data being requested has to be located closer to the reading node, and therefore when it isn't, it gets replicated. Comparing the average download time with k=1.6 and k=1.8, the first test seems to have had a little more replication. But in reality, the access time to data after that, for both tests are the same. This is because with k=1.6, data has been over-replicated to the edge networks. Given the limited size of our network, studying this effect is not straightforward.

- Core to edge offloading

Here, we study the systems backhaul offloading capabilities. As data gets replicated closer to the consuming datanodes, located at the network edge, it no longer needs to pass through the backbone link, leaving it with more available bandwidth. Incoming traffic has been monitored on the edge and gateways node's interfaces. This data has been processed and presented in Figure 47. This figure shows the total accumulated traffic transferred in the system, either at

the edge or at the core, for a test scenario with β=550; k=1.2; WT=6s; Δtn=2s and α=99%. Edge traffic refers to transfers between nodes in the same edge network. Core traffic refers to transfers between nodes in different edge networks. At the initial part of this experiment, more traffic is expected at the core. As data starts to get replicated to the edge, the core links should get less used.

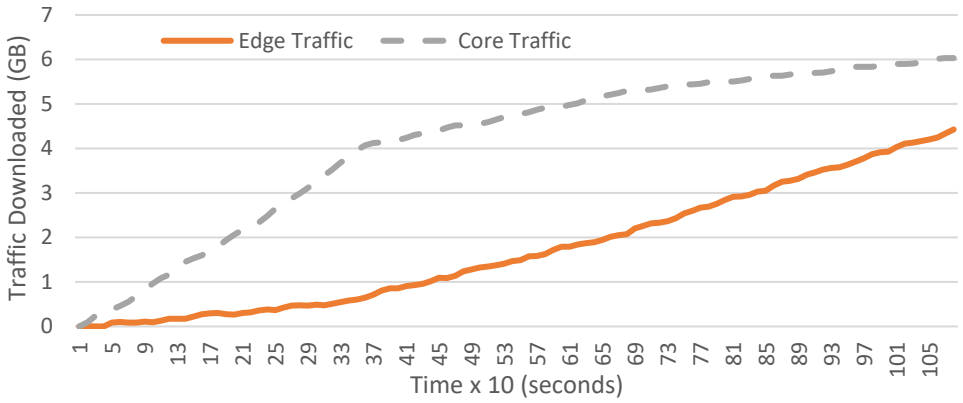


Figure 47 - Core to edge offloading

Figure 47 shows that the core connections are heavily used up until around 360s, while the edge connections are not very used. This is expected as many file requests are for data belonging to other edge networks, so data has to pass through the core. At around 360s, data finishes being replicated to the edge. From this point on, the core traffic increases very slowly, while the edge traffic increases at a much higher rate. This is because now, most data being requested is located on the same network as the reader. Since the edge network has more bandwidth than the core, access speeds to data should also increase. This can be confirmed, for example, by comparing the average transfer time on the first (7s) and fifth request (3s) in Figure 45 with β=750.

5.2.2.3 Battery Level Scenario

In this scenario, we want aimt study the system's fairness in distributing content according to energy level. A few datanodes will be strategically placed to more often fulfil read requests. The system is expected to increase data availability by not letting nodes that store popular files completely drain their energy level, by effectively offloading data requests to datanodes with higher energy level.

Test scenario

In this scenario, three "new" datanodes (L1, L2 and L3) have been placed in between the other nodes on each cluster, as in Figure 48. The other datanodes then write a single file into the HDFS. In each edge network, a replica from each node is placed on the new datanode, as they are positioned to be the closest node to all three previous datanodes. Since they are closest, the

namenode will consider them as the main option to retrieve data from, for every request from each edge network.

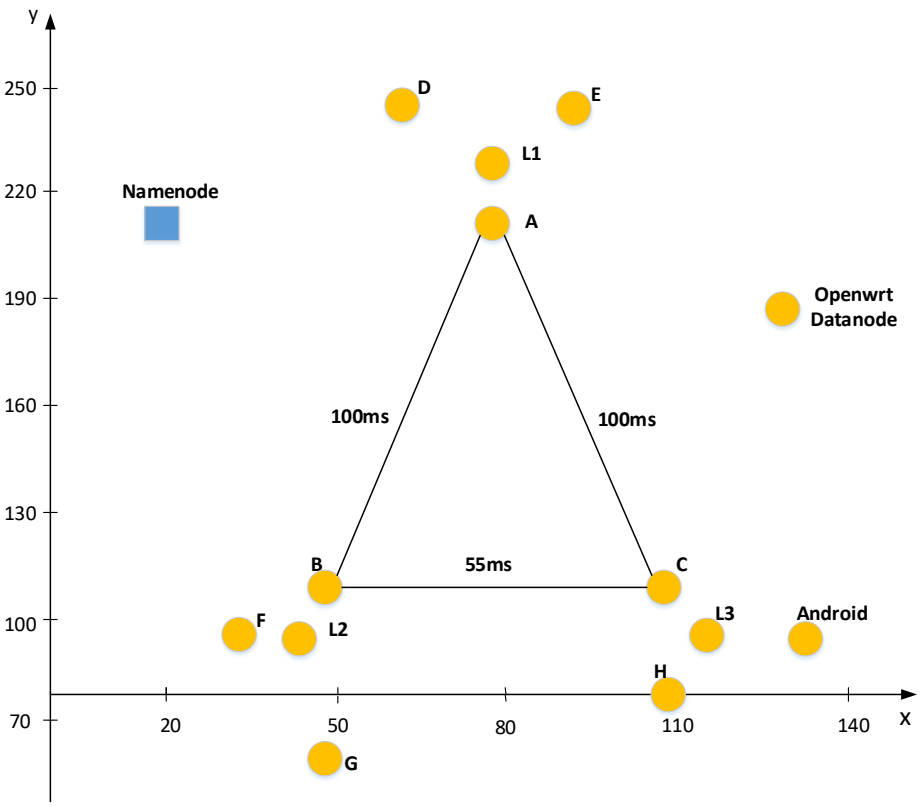


Figure 48 - Node layout for battery level test scenario

Afterwards, all datanodes except for L1, L2 and L3, are instructed to request all files in the filesystem. This task is set to run a configurable number of times – the same as in the previous test scenarios.

All datanodes in the figure have an initial battery level of 100%, and for every transfer between two nodes, their battery level decrease by 1%. At the end of the test, battery levels have been collected from each node, and the obtained results have been processed to present in a graphic.

When we run the tests, it is expected that the system starts by having the nodes that hold popular files to fulfil the requests. As their battery level gets lower, the namenode should guarantee that other nodes, with more battery level will be added to fulfil the requests. In practice, content will be offloaded to nodes with more energy resources.

Configurable Parameters

The configurable parameter that has been tested is ‘ω’ (equation 1 in section 3.3). By increasing its value, more importance is given to saving energy level in the system. This parameter is

changed in a configuration file at the beginning of the test scenario. The number of file requests per node has been set to seven.

Tests and Evaluation

Figure 49 shows the standard deviation of the battery levels collected at the end of the test, for different values of 'ω'.

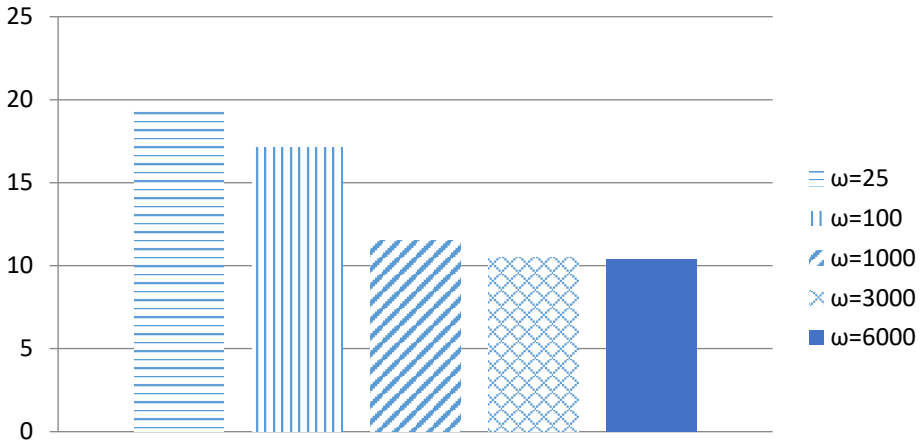


Figure 49 - Standard deviation for different values of 'ω'

Figure 49 shows that lower values of 'ω' (e.g. 25) have higher standard deviation. Higher values of standard deviation indicate that the values of battery level that have been collected at the end of the test, are very diverse. This means that some nodes finished the test with a rather low battery level, and others with high battery level. Therefore, using lower values of 'ω' indicates less fairness in distributing data, as nodes that store popular data get penalized by draining their battery level. On the other hand, when 'ω'=1000, the nodes at the end of the test all have similar battery levels, which is made evident by the low standard deviation. In this way, there is a fairer distribution of energy consumption, by effectively distributing content in the system.

Chapter 6. Conclusions and Future Work

6.1 Conclusions

The main goal of the current work was to research on the possibility of using mobile devices as resource providers for a cloud storage system. These storage resources could then be accessed by mobile device users. For this purpose, we have implemented a system that pools the storage resources of mobile devices and fog nodes (e.g. common equipment placed at user's premises) to create the cloud storage. Content distribution has been managed by taking into account the networking node's energy levels and positions. The content distribution module includes a replication manager based on data popularity. In this way, popular data is consistently replicated to the edge networks, closer to end users that have a high expectation for that data.

Our results show that the system reduces the latency experienced by the end users when they access data. This responsiveness is due to the design of keeping data close to end users, by using node's localization information. Furthermore, data gets consistently replicated to the edge network, where end users reside. This also benefits system responsiveness, as available bandwidth at the edge is usually higher. The system's responsiveness can be controlled by configuring the replication intensity in the system, through parameters that determine the requested data's required proximity to reader nodes, and how popular data needs to be to get replicated.

Additionally, results show that due to the system's design of keeping data close to the data consumers, the backhaul links use less bandwidth, as data get replicated to the edge, mitigating the backhaul bandwidth limitations that early proposals of MCC suffer from.

Furthermore, the system includes a configuration parameter for regulating the fairness in distributing energy consumption among nodes. Results show that the system evidences fairness in energy depletion when disseminating content. This is important because it enhances the network's lifetime and data availability by saving energy on mobile nodes with lower levels of available energy.

Overall, the current work contributes to a new data storage solution for emerging mobile services that require responsiveness in accessing popular data, by using the increasing available resources on mobile devices, while managing their energy consumption in a fair manner.

6.2 Future Work

There is room for improvement in our system, as we detail as follows.

The network coordinate system that showed the best results in accuracy in section 5.2.1 was Pharos. However, the implementation of Pharos that we have used does not account for node mobility. Therefore, the system is not being used at its full potential. The Pharos implementation could be altered to integrate node mobility. We have proposed a solution, that can be seen in Appendix A – Part 1.

The implemented data placement scheme utilizes networking nodes' energy levels and node position and popularity, for evaluating where data should be in the system. We expect that the use of other metrics could further increase the network's lifetime and performance. Some of the metrics that could be researched in the future, are:

- Available node storage
- Mobility speed
- Available heterogeneous access technologies
- Connectivity level

Another suggestion left for future work is running a balancer thread periodically. This thread could eventually move data closer to expected consumers in a proactive way. This would add to our reactive replication module, which only replicates after a 'hit threshold' is reached by a user.

The last suggestion refers to the replication strategy regarding the scenario where there are only mobile nodes nearby. In such a situation, it is not fair to simply store data on other nodes, and retrieve data from them, potentially draining their energy level. Caching techniques that take into consideration battery level could be added as a tool for the replication strategy to deploy, depending on the writer's surrounding circumstances (nearby node types and available energy level).

Bibliography

[1] L. Guan, X. Ke, M. Song, and J. Song, “A Survey of Research on Mobile Cloud Computing,” in 2011 10th IEEE/ACIS International Conference on Computer and Information Science, 2011, pp. 387–392.

[2] T. H. Luan, L. Gao, Z. Li, Y. Xiang, G. Wei, and L. Sun, “Fog Computing: Focusing on Mobile Users at the Edge,” pp. 1–11, 2015.

[3] S. Abolfazli, Z. Sanaei, E. Ahmed, A. Gani, and R. Buyya, “Cloud-based augmentation for mobile devices: Motivation, taxonomies, and open challenges,” IEEE Commun. Surv. Tutorials, vol. 16, no. 1, pp. 337–368, 2014.

[4] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso, “Calling the Cloud: Enabling Mobile Phones as Interfaces to Cloud Applications,” Springer, Berlin, Heidelberg, 2009, pp. 83–102.

[5] “Is it a Cloud? Is it a Hacker? No, It’s a Spof | Nasstar PLC.” [Online]. Available: http://blog.nasstar.com/is-it-a-cloud-is-it-a-hacker-no-its-a-spof/. [Accessed: 15-Sep-2017].

[6] M. Firdhous, O. Ghazali, and S. Hassan, “Fog Computing: Will it be the Future of Cloud Computing?,” Third Int. Conf. Informatics Appl., pp. 8–15, 2014.

[7] P. Hu, S. Dhelim, H. Ning, and T. Qiu, “Survey on fog computing: architecture, key technologies, applications and open issues,” J. Netw. Comput. Appl., vol. 98, no. September, pp. 27–42, 2017.

[8] K. Dolui and S. K. Datta, “Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing,” in 2017 Global Internet of Things Summit (GIoTS), 2017, pp. 1–6.

[9] R. Mahmud and R. Buyya, “Fog Computing: A Taxonomy, Survey and Future Directions,” pp. 1–28, 2016.

[10] M. Satyanarayanan, P. Bahl, R. Cáceres, and N. Davies, “The case for VM-based cloudlets in mobile computing,” IEEE Pervasive Comput., vol. 8, no. 4, pp. 14–23, Oct. 2009.

[11] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang, “A Survey on Mobile Edge Networks: Convergence of Computing, Caching and Communications,” IEEE Access, vol. 5, pp. 6757–6779, 2017.

[12] W. Paper, “Cisco Fog Computing Solutions : Unleash the Power of the Internet of Things,” White Pap., pp. 1–6, 2015.

[13] “Number of smartphone users worldwide 2014-2020 | Statista.” [Online]. Available: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. [Accessed: 15-Sep-2017].

[14] B. Depardon, G. Le Mahec, and C. Séguin, “Analysis of Six Distributed File Systems,” 2013.

[15] E. Marinelli, "Hyrax: Cloud Computing on Mobile Devices using MapReduce,", Master Thesis Draft, Computer Science Dept., CMU, September 2009.

[16] M. A. Hassan, M. Xiao, Q. Wei, and S. Chen, “Help your mobile applications with fog computing,” 2015 12th Annu. IEEE Int. Conf. Sensing, Commun. Netw. - Work. SECON Work. 2015, pp. 49–54, 2015.

[17] R. Monteiro, J. Silva, J. Lourenço, and H. Paulino, “Decentralized Storage for Networks of Hand-held Devices,” in Proceedings of the 12th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, 2015.

[18] Y. Jararweh, L. Tawalbeh, F. Ababneh, A. Khreishah, and F. Dosari, “Scalable Cloudlet-based Mobile Computing Model,” Procedia Comput. Sci., vol. 34, pp. 434–441, 2014.

[19] N. Mohan and J. Kangasharju, “Edge-Fog cloud: A distributed cloud for Internet of Things computations,” 2016 Cloudification Internet Things, CIoT IIE, 2016. p. 1-6.
[20] E. Zeydan et al., “Big data caching for networking: Moving from cloud to edge,” IEEE Commun. Mag., vol. 54, no. 9, pp. 36–42, 2016.
[21] “ETSI - Multi-access Edge Computing.” [Online]. Available: http://www.etsi.org/technologies-clusters/technologies/multi-access-edge-computing. [Accessed: 28-Sep-2017].
[22] “ETSI - ETSI Multi-access Edge Computing group releases a first package of APIs.” [Online]. Available: http://www.etsi.org/news-events/news/1204-2017-07-news-etsi-multi-access-edge-computing-group-releases-a-first-package-of-apis. [Accessed: 28-Sep-2017].
[23] F. Gabry, V. Bioglio, and I. Land, “On Energy-Efficient Edge Caching in Heterogeneous Networks,” IEEE J. Sel. Areas Commun., vol. 34, no. 12, pp. 3288–3298, 2016.
[24] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, “Cloudlets,” Proceedings of the third ACM workshop on Mobile cloud computing and services - MCS ’12, 2012, p. 29.
[25] S. Abolfazli, Z. Sanaei, M. Shiraz, and A. Gani, “MOMCC: Market-oriented architecture for Mobile Cloud Computing based on Service Oriented Architecture,” 2012 1st IEEE Int. Conf. Commun. China Work., pp. 8–13, 2012.
[26] Q. Wang, Z. Hu, M. Wang, and H. Liu, “CACTSE: Cloudlet aided cooperative terminals service environment for mobile proximity content delivery,” China Commun., vol. 10, no. 6, pp. 47–59, 2013.
[27] K. Habak, M. Ammar, K. A. Harras, and E. Zegura, “FemtoClouds: Leveraging Mobile Devices to Provide Cloud Service at the Edge.” In: Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on. IEEE, 2015. p. 9-16.
[28] R. Lacuesta, J. Lloret, S. Sendra, and L. Peñalver, “Spontaneous ad hoc mobile cloud computing network,” Sci. World J., vol. 2014, article 232419, 2014.
[29] C. Barca et al., “A virtual cloud computing provider for mobile devices,” in Proceedings of the 8th International Conference on Electronics, Computers and Artificial Intelligence, ECAI 2016, 2017, pp. 1–5.
[30] R. K. Panta, R. Jana, F. Cheng, Y.-F. R. Chen, and V. A. Vaishampayan, “Phoenix: Storage Using an Autonomous Mobile Infrastructure,” IEEE Trans. Parallel Distrib. Syst 2013, 24.9: 1863-1873.
[31] A. Moon and H. Cho, “Energy Efficient Replication Extended Database State Machine in Mobile Ad Hoc Network,” IADIS Int. Conf. Appl. Comput., 2004. p. 224-228.
[32] Y. Lu, B. Zhou, L.-C. Tung, M. Gerla, A. Ramesh, and L. Nagaraja, “Energy-efficient content retrieval in mobile cloud,” Proc. Second ACM SIGCOMM Work. Mob. cloud Comput. - MCC ’13, p. 21, 2013.
[33] C. Science and C. Paniagua, “Establishing Peer-to-Peer Distributed File Sharing System With Mobile Host,” May, 2013.
[34] “OMNeT++ Wiki | Main / OMNeT++ 3.x browse.” [Online]. Available: https://omnetpp.org/pmwiki/index.php?n=Main.Omnetpp3. [Accessed: 09-Oct-2017].
[35] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman, “A survey of information-centric networking,” IEEE Commun. Mag., vol. 50, no. 7, pp. 26–36, Jul. 2012.
[36] T. White, “Hadoop: The definitive guide 4th Edition,” Online, vol. 54, p. 258, 2012.
[37] “What is Hadoop Metrics2? – Cloudera Engineering Blog.” [Online]. Available: http://blog.cloudera.com/blog/2012/10/what-is-hadoop-metrics2/. [Accessed: 02-Oct-2017].
[38] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, “CRUSH: Controlled,

Scalable, Decentralized Placement of Replicated Data.” In: Proceedings of the 2006 ACM/IEEE conference on Supercomputing. ACM, 2006. p. 122.

[39] G. Paschos, E. Batu˘, I. Land, G. Caire, and M. Debbah, “Wireless Caching: Technical Misconceptions and Business Barriers,” IEEE Communications Magazine, 2016, 54.8: 16-22.

[40] M. Ahmed, S. Traverso, M. Garetto, P. Giaccone, E. Leonardi, and S. Niccolini, “Temporal Locality in Today’s Content Caching: Why it Matters and How to Model it,” ACM SIGCOMM Computer Communication Review, 2013, 43.5: 5-12.

[41] Z. Cheng et al., “ERMS: An elastic replication management system for HDFS,” Proc. - 2012 IEEE Int. Conf. Clust. Comput. Work. Clust. Work. 2012, pp. 32–40, 2012.

[42] S. He, H. Tian, and X. C. Lyu, “Edge Popularity Prediction Based on Social-Driven Propagation Dynamics,” Ieee Commun. Lett., vol. 21, no. 5, pp. 1027–1030, 2017.

[43] M. Dehghan et al., “On the Complexity of Optimal Routing and Content Caching in Heterogeneous Networks,” In: Computer Communications (INFOCOM), 2015 IEEE Conference on. IEEE, 2015. p. 936-944.

[44] D. Liu, B. Chen, C. Yang, and A. F. Molisch, “Caching at the wireless edge: Design aspects, challenges, and future directions,” IEEE Commun. Mag., vol. 54, no. 9, pp. 22–28, 2016.

[45] T. Yang, H. Pen, W. Li, D. Yuan, and A. Zomaya, “An Energy-efficient Storage Strategy for Cloud Datacenters based on Variable K-Coverage of a Hypergraph,” IEEE Trans. Parallel Distrib. Syst., vol. 9219, 2017.

[46] B. Liao, J. Yu, T. Zhang, G. Binglei, S. Hua, and C. Ying, “Energy-efficient algorithms for distributed storage system based on block storage structure reconfiguration,” J. Netw. Comput. Appl., vol. 48, pp. 71–86, 2015.

[47] R. T. Kaushik, “GreenHDFS : Towards An Energy-Conserving , Storage-Efficient , Hybrid Hadoop Compute Cluster,” HotPower, pp. 1–9, 2010.

[48] R. N. M. Carlos Meralto, José Moura, “Mesh Networks for Handheld Mobile Devices,” Conftele, 2015.

[49] M. Satyanarayanan, “Pervasive computing: Vision and challenges,” IEEE Pers. Commun., vol. 8, no. 4, pp. 10–17, 2001.

[50] S. Ratnasamy et al., “A scalable content-addressable network,” Proc. 2001 Conf. Appl. Technol. Archit. Protoc. Comput. Commun. - SIGCOMM ’01, vol. 31, no. 4, pp. 161–172, 2001.

[51] B. Donnet, B. Gueye, M. A. Kaafar, and M. Ali Kaafar, “A survey on network Coordinates systems,design, and security,” IEEE Commun. Surv. Tutorials, vol. 12, no. 4, pp. 488–503, 2010.

[52] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, “Vivaldi: A Decentralized Network Coordinate System.” In: ACM SIGCOMM Computer Communication Review. ACM, 2004. p. 15-26.

[53] Y. Chen, Y. Xiong, X. Shi, B. Deng, and X. Li, “Pharos: A decentralized and hierarchical network coordinate system for internet distance prediction,” in GLOBECOM - IEEE Global Telecommunications Conference, 2007, pp. 421–426.

[54] Y. Chen et al., “Phoenix: A weight-based network coordinate system using matrix factorization,” IEEE Trans. Netw. Serv. Manag., vol. 8, no. 4, pp. 334–347, 2011.

[55] “Why GPS eats so much battery power, explained by a Google engineer | ITworld.” [Online]. Available: https://www.itworld.com/article/2833266/mobile/why-gps-eats-so-much-battery-power--explained-by-a-google-engineer.html. [Accessed: 02-Oct-2017].

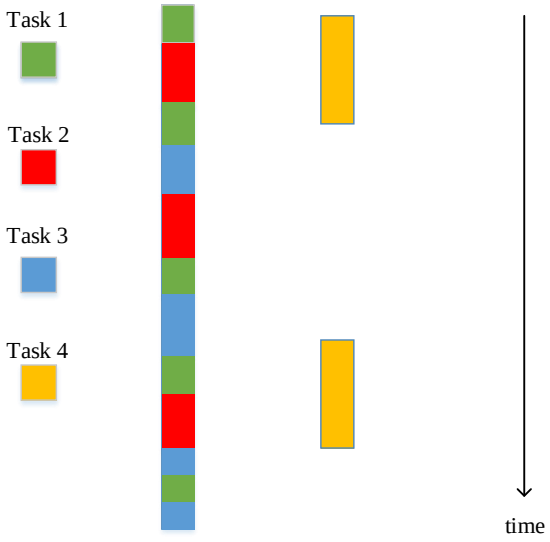
[56] “How does TCP round trip time (RTT) estimation work? How different is the implementation across operating systems? - Quora.” [Online]. Available: https://www.quora.com/How-does-TCP-round-trip-time-RTT-estimation-work-How-

different-is-the-implementation-across-operating-systems. [Accessed: 02-Oct-2017].
[57] "OpenWrt." [Online]. Available: https://forum.openwrt.org/. [Accessed: 17-Oct-2017].
[58] "3 Sistemas Operativos Linux para instalar no seu smartphone." [Online]. Available: https://pplware.sapo.pt/smartphones-tablets/3-sistemas-operativos-linux-smartphone/. [Accessed: 22-Sep-2017].
[59] "Está prestes a chegar um novo smartphone com Linux? - Pplware." [Online]. Available: https://pplware.sapo.pt/linux/chegar-novo-smartphone-linux/. [Accessed: 22-Sep-2017].
[60] "Makefile in packages/lang/jamvm – OpenWrt." [Online]. Available: https://dev.openwrt.org/browser/packages/lang/jamvm/Makefile?rev=20608. [Accessed: 17-Oct-2017].
[61] "[HDFS-4387] libhdfs doesn't work with jamVM - ASF JIRA." [Online]. Available: https://issues.apache.org/jira/browse/HDFS-4387. [Accessed: 10-Jul-2017].
[62] "Run java code on OpenWrt (Page 1) — General Discussion — OpenWrt." [Online]. Available: https://forum.openwrt.org/viewtopic.php?id=53689. [Accessed: 10-Jul-2017].
[63] "Java Environment on OpenWrt (OXNAS) (Page 3) — General Discussion — OpenWrt." [Online]. Available: https://forum.openwrt.org/viewtopic.php?id=54849&p=3. [Accessed: 10-Jul-2017].
[64] Namrata B Bothe, Snehal S Karale, Anagha N Mate, and Nayan D Kumbhar, "Migration of Hadoop To Android Platform Using 'Chroot,'" Ijirct, vol. 1, no. 5, pp. 486–488, 2016.
[65] "Linux on Android." [Online]. Available: https://arachnoid.com/android/LinuxOnAndroid/. [Accessed: 14-Sep-2017].
[66] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: a measurement study and implications for network applications," Proc. 9th ACM SIGCOMM Conf. Internet Meas. Conf., pp. 280–293, 2009.
[67] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," Parallel Comput., vol. 30, pp. 817–840, 2004.
[68] "Introduction to HDFS Federation in Hadoop - DataFlair." [Online]. Available: http://data-flair.training/blogs/hadoop-hdfs-federation-tutorial/. [Accessed: 18-Sep-2017].
[69] "tc-netem(8) - Linux manual page." [Online]. Available: http://man7.org/linux/man-pages/man8/tc-netem.8.html. [Accessed: 07-Oct-2017].
[70] "In Which We Begin at the Beginning – krondo." [Online]. Available: http://krondo.com/in-which-we-begin-at-the-beginning/. [Accessed: 14-Sep-2017].
[71] "Add a basic implementation of GMonitor as an Android." [Online]. Available: https://github.com/opentelecoms-org/lumicall/commit/8ed0698081aa11570c4aa87ec550a95b79a43375. [Accessed: 31-Oct-2017].
[72] "gmetric and basic gmond functionality in Java." [Online]. Available: https://github.com/ganglia/gmetric4j [Accessed: 29-Oct-2017].
[73] "Linux deploy" [Online]. Available: https://play.google.com/store/apps/details?id=ru.meefik.linuxdeploy&hl=pt_PT [Accessed: 29-Oct-2017].
[74] "VX Connectbot." [Online]. Available: https://play.google.com/store/apps/details?id=sk.vx.connectbot&hl=pt_PT [Accessed: 29-Oct-2017].

Appendix A

Part 1 - Proposal for adapting Pharos to mobile scenarios

The following figure shows our proposal for the Pharos mobility model – an asynchronous and threaded model [70].



The continuous single-thread in the middle is how the asynchronous model that Pharos uses, works. A single thread of control carries out a number of tasks one after the other. A task can only start when the previous task has finished. Tasks 1, 2 and 3 refer to tasks that have already been implemented in Pharos. In an effort to optimize Pharos to work in a scenario where node mobility is a factor, we propose to add Task 4 to Pharos, as a parallel running thread, as shown in Figure 1. Task 4 should periodically run to re-determine the cluster that the client belongs to. This must be done in parallel because of the duration of this task, compared to the ones in the single thread. It ends up delaying the normal functionality of the system if it is added as a task in the pure asynchronous model. Code would also have to be added to update global and local neighbour lists in case there is a cluster update.

Part 2 – Hadoop-metrics2.properties Configuration

The following images enable ganglia sink and configure the periodic interval for sending metrics to the namenode (10s).

```
datanode.sink.ganglia.servers=namenode:8649
```

```
#
# Below are for sending metrics to Ganglia
#
# for Ganglia 3.0 support
# *.sink.ganglia.class=org.apache.hadoop.metrics2.sink.ganglia.GangliaSink30
#
# for Ganglia 3.1 support
 *.sink.ganglia.class=org.apache.hadoop.metrics2.sink.ganglia.GangliaSink31

 *.sink.ganglia.period=10
```

Part 3 – Hdfs-site.xml Configuration

The following figure shows the configured heartbeat interval time (12s) and the block recovery executing periodicity (1s).

```
<property>
<name>dfs.namenode.replication.interval</name>
<value>1</value>
</property>

<property>
<name>dfs.heartbeat.interval</name>
<value>12</value>
</property>
```

Part 4 – Adding custom metrics to Heartbeat protocol

The following figure shows how variables have been added to the heartbeat protocol. The changes have been made in a class designated by DatanodeProtocol.proto.

```
message HeartbeatRequestProto {
  required DatanodeRegistrationProto registration = 1; // Datanode info
  repeated StorageReportProto reports = 2;
  optional uint32 xmitsInProgress = 3 [ default = 0 ];
  optional uint32 xceiverCount = 4 [ default = 0 ];
  optional uint32 failedVolumes = 5 [ default = 0 ];
  optional uint64 cacheCapacity = 6 [ default = 0 ];
  optional uint64 cacheUsed = 7 [default = 0 ];
  optional VolumeFailureSummaryProto volumeFailureSummary = 8;
  optional uint32 batterylevel = 9 [ default = 0 ];
  optional string coords = 10;
}
```

After adding these metrics to the heartbeat; code was added on the datanode to retrieve and send the metrics via heartbeat; and code was added on the namenode to update the topology map with the metrics received in the heartbeat.

Part 5 – Block recovery feature consuming entry in ReplicationQueue

```
if((FSNamesystem.queue.get(rw.block.getBlockId())!= null)){
    Node clientnode =getDatanodeManager().getDatanodeByHost(FSNamesystem.queue.get(rw.block.getBlockId()).get(0));
    FSNamesystem.queue.get(rw.block.getBlockId()).remove(0);
    if(FSNamesystem.queue.get(rw.block.getBlockId()).isEmpty())
        FSNamesystem.queue.remove(rw.block.getBlockId());
    DatanodeDescriptor testing = (DatanodeDescriptor) clientnode;
    rw.targets=testing.getStorageInfos();
}
```

The block recovery feature checks if there is any entry in the ReplicationQueue regarding a particular block. If there is, then the system retrieves the node that should receive this replica – clientNode (TargetNode). The entry is consumed in the ReplicationQueue, and replication work is registered for that particular file to be stored on the clientNode.

Part 6 – Network Emulation Scripts

The following figures shows code referring to the central machine ssh'ing into nodes A and B and running a local script – exemplo22.sh – which receives as arguments the latency delays to all the other networking nodes and their respective IP. These latency delays have previously been calculated by using the Euclidean distance.

```
Aip="192.168.1.2"
Bip="192.168.1.3"
Cip="192.168.1.4"
Dip="192.168.1.5"
Eip="192.168.1.6"
Fip="192.168.1.7"
Gip="192.168.1.8"
Hip="192.168.1.9"
NNip="192.168.1.145"
Androidip="192.168.1.48"

#A
ssh root@192.168.1.2 "bash -s" < /usr/local/hadoop-2.7.0-src/exemplo22.sh $ABdelay $ACdelay $ADdelay $AEdelay $AFdelay $AGdelay $AHdelay
$ANNdelay $AndroidAdelay $Bip $Cip $Dip $Eip $Fip $Gip $Hip $NNip $Androidip &

#B
ssh root@192.168.1.3 "bash -s" < /usr/local/hadoop-2.7.0-src/exemplo22.sh $ABdelay $BCdelay $BDdelay $BEdelay $BFdelay $BGdelay $BHdelay
$BNNdelay $AndroidBdelay $Aip $Cip $Dip $Eip $Fip $Gip $Hip $NNip $Androidip &
```

The following caption shows the code in the exemplo22.sh script that is hosted on the central machine, and ran on remote machines, to create latency. It has the TC commands, responsible for emulating latency on a node. The script receives latency delay and IP arguments.

```
#!/bin/bash

tc qdisc del dev eth0 root
tc qdisc add dev eth0 root handle 1: htb
tc class add dev eth0 parent 1: classid 1:1 htb rate 300mbit burst 50kb mtu 10000
tc class add dev eth0 parent 1:1 classid 1:11 htb rate 20mbit burst 50kb mtu 10000
tc class add dev eth0 parent 1:1 classid 1:10 htb rate 20mbit burst 50kb mtu 10000
tc class add dev eth0 parent 1:1 classid 1:9 htb rate 20mbit burst 50kb mtu 10000
tc class add dev eth0 parent 1:1 classid 1:8 htb rate 20mbit burst 50kb mtu 10000
tc class add dev eth0 parent 1:1 classid 1:7 htb rate 20mbit burst 50kb mtu 10000
tc class add dev eth0 parent 1:1 classid 1:6 htb rate 20mbit burst 50kb mtu 10000
tc class add dev eth0 parent 1:1 classid 1:5 htb rate 20mbit burst 50kb mtu 10000
tc class add dev eth0 parent 1:1 classid 1:4 htb rate 20mbit burst 50kb mtu 10000
tc class add dev eth0 parent 1:1 classid 1:3 htb rate 20mbit burst 50kb mtu 10000
#tc class add dev eth0 parent 1:1 classid 1:2 htb rate 1017kbps
tc qdisc add dev eth0 parent 1:11 netem delay $1
tc qdisc add dev eth0 parent 1:10 netem delay $2
tc qdisc add dev eth0 parent 1:9 netem delay $3
tc qdisc add dev eth0 parent 1:8 netem delay $4
tc qdisc add dev eth0 parent 1:7 netem delay $5
tc qdisc add dev eth0 parent 1:6 netem delay $6
tc qdisc add dev eth0 parent 1:5 netem delay $7
tc qdisc add dev eth0 parent 1:4 netem delay $8
tc qdisc add dev eth0 parent 1:3 netem delay $9
#tc qdisc add dev eth0 parent 1:2 netem delay $8
tc filter add dev eth0 parent 1:0 protocol ip u32 match ip dst ${10} flowid 1:11
tc filter add dev eth0 parent 1:0 protocol ip u32 match ip dst ${11} flowid 1:10
tc filter add dev eth0 parent 1:0 protocol ip u32 match ip dst ${12} flowid 1:9
tc filter add dev eth0 parent 1:0 protocol ip u32 match ip dst ${13} flowid 1:8
tc filter add dev eth0 parent 1:0 protocol ip u32 match ip dst ${14} flowid 1:7
```