Universidade do Minho
Escola de Engenharia

Diana Sofia Chaves Martins

Safe storage of medical images in NoSQL databases

December 2016

Safe storage of medical images in NoSQL databases
Diana Sofia Chaves Martins
Uminho | 2016

Universidade do Minho
Escola de Engenharia

Diana Sofia Chaves Martins

Safe storage of medical images in NoSQL databases

Master dissertation
Master Degree in Medical Informatics

Dissertation supervised by
Professor Doutor António Luís Sousa

December 2016

ACKNOWLEDGEMENTS

In first place, I would like to thank professor António Luís Sousa for helping me find this interesting thesis theme and accompanying me during its development with such effort and dedication. Also, I am thankful for the opportunity of working in the HASLab group with remarkable professionals.

To my lab colleagues, I am thankful for their constant assistance and guidance, specially to Rogério Pontes who showed himself always available to help in every need, sharing fundamental insight. The amount of technical knowledge I acquired by working with him is simply huge.

I am also grateful to my family for working hard and always supporting me, making this major achievement possible.

To my closest friends, who played an important role on maintaining my perseverance, showing themselves always present to give me hope and help me through stressful times.

A huge thank to Nuno Santos, for walking by my side during this journey, supporting me, cheering me, and also helping me in every need. Such journey is always more pleasant when made with someone as special and amazing as him.

ABSTRACT

Databases are a critical point for medical institutions since, without them, information systems can not work. In such way, the usage of a secure, easily scalable and highly available database is crucial when dealing with medical data.

Nowadays, medical institutions use RDBMSs – or SQL databases – in their medical images' information systems. However, this type of databases has drawbacks at various levels. For instance, they show scalability issues in distributed environments or when dealing with high numbers of users. Besides, their data schema is complex and difficult to create. In terms of functionalities, this kind of databases show a huge set of features, which add extra unnecessary complexity when dealing with large complex datasets.

An alternative to RDBMSs are NoSQL databases. These are databases design to deal with large amounts of heterogeneous data without giving up high performance rates. However, both SQL and NoSQL databases still present major concerns regarding security. In the medical field, since there is sensitive and personal data being exchanged, their protection is a crucial point when implementing a storage solution for medical purposes.

Having this said, this master thesis focuses on the development of a medical imaging database with increased availability by the employment of a NoSQL HBase backend. In order to protect the data being stored, a secure version of HBase protected with symmetric cipher was also implemented. Both protected and unprotected versions were implemented, tested, and compared with an open source toolkit.

From the results obtained by executing performance tests, it is possible to conclude that the HBase backend shows a better overall performance in terms of latency and throughput in comparison to the open source toolkit. However, the appliance of the encryption service implies higher latency and lower throughput.

RESUMO

As bases de dados são um ponto crítico das instituições médicas, uma vez que sem elas os sistemas de informação não podem funcionar. Desta forma, no que respeita a dados médicos, a utilização de uma base de dados segura, facilmente escalável e com alta disponibilidade é crucial.

Hoje em dia, as instituições médicas utilizam bases de dados relacionais – ou SQL – nos sistemas de informação que lidam com imagens médicas. No entanto, este tipo de bases de dados apresentam desvantagens a vários níveis. Por exemplo, estas demonstram problemas a nível de escalabilidade em ambiente distribuído, bem como no que respeita à sua utilização por parte de um número elevado de utilizadores. Para além disto, o seu modelo de dados é complexo e difícil de criar. Este tipo de bases de dados tem também um elevado número de funcionalidades que, quando o volume de dados é elevado, adicionam complexidade desnecessária ao sistema.

Uma alternativa a este tipo de bases de dados são as bases de dados NoSQL. Estas foram desenhadas para lidar com grandes volumes de dados heterogéneos, mantendo uma elevada performance. No entanto, tanto as bases de dados SQL como NoSQL, apresentam problemas a nível de segurança. Na área médica, uma vez que são manuseados dados sensíveis e de cariz pessoal, a implementação de uma base de dados segura é fundamental.

Desta forma, a presente dissertação de mestrado foca-se no desenvolvimento de uma base de dados para imagens médicas com disbonibilidade aumentada através da implementação de uma base de dados NoSQL – o HBase. De forma a proteger os dados sensíveis que são guardados, foi implementada uma versão de HBase protegida com cifra simétrica. Ambas versões – protegida e não protegida – foram implementadas, testadas, e comparadas com um tooltik open source.

Dos resultados obtidos a partir da execução de testes de performance é possível concluir que o HBase simples tem uma melhor performance em termos de latência e débito, em comparação com o toolkit open source. No entanto, a aplicação de cifra simétrica implica uma maior latência e um menor débito.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

A

ACID Atomicity, Consistency, Isolation and Durability.

AES Advanced Encryption Standard.

AET Application Entity Title.

API Application Programming Interface.

B

BASE Basically Available, Soft state, Eventually consistent.

BLOB Binary Large OBject.

C

CAP Consistency, Availability and Partition Tolerance.

CBC Cipher Block Chaining.

CBIR Content Based Image Retrieval.

CLOB Character Large OBject.

CNT Cosine Number Transformation.

CR Computed Radiography.

CRUD Create, Read, Update and Delete.

CT Computed Tomography.

D

DBMS Database Management System.

DE Digital Envelope.

DES Data Encryption Standard.

DICOM Digital Imaging and Communications in Medicine.

DS Digital Signature.

DSR Design Science Research.

E

EHR Electronic Health Record.

F

FMI File Meta-Information.

G

GUI Graphical User Interface.

H

HBASE Hadoop-based Project.

HDFS Hadoop Distributed File System.

HIPAA Health Insurance Portability and Accountability Act.

I

IOD Information Object Definition.

IT Information Technologies.

J

JVM Java Virtual Machine.

K

KPI Key Performance Indicator.

M

MRI Magnetic Resonance Imaging.

N

NEMA National Electrical Manufacturers Association.

NIST National Institute of Standards and Technology.

NOSQL Not Only SQL.

P

PACS Picture Archiving Communication System.

PKCS Public Key Cryptography Standards.

R

RC Rivest Cipher.

RDBMS Relational Database Management System.

S

SCP Service Class Provider.

SCU Service Class User.

SOP Service-Object Pair.

SQL Structured Query Language.

U

UID Unique Identifier.

URI Uniform Resource Identifier.

1

INTRODUCTION

This chapter has the purpose of introducing this master thesis' development. It will begin with a short contextualization and motivation of the theme on Section 1.1, followed by what are its main goals on Section 1.2. Also, all this study's conclusions and steps taken towards them are be described on Section 1.3. At last, a brief description of the present document's structure will be presented to make the reader's job easier – Section 1.4.

1.1 CONTEXT AND MOTIVATION

Currently, to deal with medical images, health institutions have usually implemented Picture Archiving Communication System (PACS). This is comprised by a sophisticated set of computers, peripherals and applications that can be connected to all image acquisition and reading modalities, whose purpose is to store, recover, distribute, and display medical images. The major advantage that can be taken of this system's usage is its low cost in comparison to other more traditional systems based on tape technology [2]. Typically, this system uses a Relational Database Management System (RDBMS) to support its action [3, 4]. From the usage of this kind of databases there are many problems that can emerge at several levels [5]:

- **Scalability:** it is not easily scalable in distributed environments or with a high number of users;
- **Relational complexity:** complex data cannot be easily inserted in tables, generating relational models with excessive complexity;
- **Functionalities:** in complex datasets, many features can add extra unnecessary complexity.

With the evolution of imaging techniques, the medical images' number and size have notoriously been increasing. Moreover, people are increasingly feeling the need of storing these images as the patient's medical history may become essential in the diagnosis of new pathologies [3, 6, 7]. In the hospital, because critical services are performed, it is

fundamental to ensure a correct and efficient work of medical images storage systems. Therefore, many efforts have been made in order to find database solutions capable of being scaled, fault tolerant, and rapidly accessed.

One of the existing solutions that allow us to achieve the above mentioned goals is the usage of Not Only SQL (NoSQL) databases. As examples of this kind of databases it is possible to mention: HBase [8], BigTable [9], CouchDB [10], MongoDB [11], OpenStack Swift [12], etc. These technologies were developed with the purpose of being capable to deal with great amounts of data that are not included in a specific schema while maintaining a high performance [3]. These databases can be grouped in three mainstream data models [5, 13–16]:

- Key-value oriented stores;
- Column oriented stores;
- Document oriented stores;

HBase can be found among the column oriented group. This is an open source distributed database, designed to deal with large amounts of data through several servers, with the ability of being persistent, strictly consistent and have near-optimal write. This technology is capable of dealing with heterogeneous datasets, meaning, textual or non-textual data [17–20]. In HBase, data is stored in labelled tables, each row having a sorting key and an arbitrary number of columns. To access tables' data, searches are made by key. Due to this, some people call HBase a key-value store or a hybrid between column and key-value oriented stores. Since DICOM files are have textual and non-textual data and need to be accessed rapidly with no faults, this can be a good solution for a medical image archive [3].

Although all these advantages that can be taken from the application of a NoSQL storage system to medical images, cloud computing has still a huge drawback in what concerns to security. When dealing with medical data there is always a great concern about who should and should not see this information. Thus, it is important to apply a security layer to a medical storage system in order to control the data's access [21].

1.2 OBJECTIVES

Having in mind the context and motivation that led to the development of this thesis – mentioned in Section 1.1 – some objectives had to be established in order to guide the work being produced. These comprise a compilation and assimilation of a great amount of information from different and heterogeneous sources and the development of a new, safe storage solution for medical images using a NoSQL database. Thus, this project's main goals are:

- Assess the adoption of new storage solutions for medical images:
  - Collect and analyse bibliographic documents about medical images storage systems;
  - Assessment of existing systems and types of files that are exchanged;
  - Analysis of characteristics and functioning of cloud based storage systems such as document oriented, key-value oriented and column oriented;
  - Assessment of the security measures usually taken regarding medical images;
  - Study on cryptographic techniques that can suit out purpose;
  - Identification of the best solution for the proposed problem.
- Increase the availability of medical images:
  - Design of a safe storage system for medical images based on the previously obtained conclusions;
  - Implementation of the designed system.
- Guarantee and assess the developed system's performance:
  - Performance tests execution;
  - Evaluation of the obtained results and comparison between them and the ones achieved with traditional storage systems.

1.3 STUDY DESCRIPTION

According to the defined objectives – Section 1.2 –, there are some main steps that need to be taken in order to achieve those. In a generic way, each one of these steps represents a chapter of this master thesis. To develop this project was followed the Design Science Research (DSR) which is described on Chapter 3.

First of all, was conducted a study that comprised the main topics important to have in mind before starting the development of a solution for the proposed problem. In this study were addressed the medical standards regarding the storage and communication in medical imaging – Section 2.3 – as well as the directives that are now being used in what concerns to the personal data privacy – Section 2.7. Furthermore, in a more technical perspective, a review of the existing storage and security solutions was made – Sections 2.4 and 2.6, respectively. Also, a more specific study was performed over the HBase storage solution – Section 2.5.

Having in mind the theoretical concepts in which the project is based, it was possible to develop two prototype systems for storing medical images. To do so, an open-source DICOM toolkit – dcm4che [22] – was used as basis – Section 4.2. The system consists on

a version of the dcm4che with an unprotected HBase backend – Section 4.3. A second dcm4che version uses as backend a HBase protected with symmetric cipher – Section 4.4. These databases were used to store the image's metadata and replace the existing database.

After the development of prototype systems, the assessment of their performance is crucial. In such way, some benchmarking techniques were applied in order to acquire and compare some performance metrics – Chapter 5. With the results obtained, some charts were constructed in order to facilitate their analysis – Chapter 6.

From this project's development it was possible to conclude – Section 7.1 – that the dcm4che with the simple HBase backend, in comparison to the baseline dcm4che, shows better results in terms of latency and throughput in the `put` and `get` operations. The `scan` shows a slightly worse performance, however, the system's query capabilities were improved. The system with a protected HBase backend shows a worse latency and throughput performance due to the encryption job in every operation, with exception for the `get` which is equivalent to the system with a simple HBase backend.

Despite the good obtained results, there is some work that can be done in this field of work – Section 7.2. For instance, it would be nice to implement a distributed filesystem such as Hadoop Distributed File System (HDFS) to store the actual medical image. Furthermore, the implementation of a more robust security technique, such as multi-party computation, would be interesting. Since medical institutions sill make use of Structured Query Language (SQL) databases, it would be interesting to implement a layer capable of translating SQL to NoSQL, letting institutions make use of NoSQL databases without changing their entire interfaces.

1.4 Document structure

This master thesis is organized in six chapters. Besides this first introductory chapter which contextualizes the reader on the subject in study, its motivations and objectives of the work, five more chapters are written hereafter:

- *Chapter 2*: Literature review;
- *Chapter 3*: Research methodology;
- *Chapter 4*: System architecture;
- *Chapter 5*: Benchmarking;
- *Chapter 6*: Results;
- *Chapter 7*: Conclusions and future work.

2

LITERATURE REVIEW

This chapter will describe the fundamental theoretical and scientific concepts related with this thesis' theme, as well as the state of the art and literature review. Section 2.1 exposes this chapter's structure and each section's importance for the present case study.

2.1 INTRODUCTION

With the evolution of healthcare industry, the need of acquiring and storing medical images has increased [4]. Thus, in order to implement an economical and convenient access to medical images, a PACS was developed [23, 24]. Having this in mind, the need to create a standard whose implementation would guarantee the interoperability and compatibility between all medical imaging stations is understandable. Hence, the DICOM [25] standard was created. Section 2.2 will describe how theses systems work and what are its main elements, and in section 2.3 the DICOM standard will be addressed, describing what are its main services and concepts.

With this in mind, one is ready to study the available NoSQL solutions fitting this work's purpose. Thus, to accomplish this, a comparison between traditional SQL – Section 2.4.1 – and newer NoSQL – Section 2.4.2 – solutions will be made. Section 2.4.2 will be divided into three subsections, with each one of these subsections addressing one of three main kinds of NoSQL databases. A description on HBase NoSQL storage system functioning will be made on Section 2.5. Its architecture and data model will be described as well. Furthermore, NoSQL applications in the healthcare industry will be referenced in Section 2.5.4.

Besides the storage of the medical data it is also important to protect it from external attacks. Hence, Section 2.6 focuses on briefly exposing symmetric-key encryption algorithm. Afterwards, the security measures, which should be taken when dealing with health-related information, are addressed – Section 2.7. In this section are exposed the most important guidelines about security and privacy in healthcare that are implemented in the United States of America and the European Union – Sections 2.7.1 and 2.7.2, respectively – as well as some applications that already are concerned about safe storing medical images – Section 2.7.3.

2.2 PICTURE ARCHIVING COMMUNICATION SYSTEM – PACS

The Picture Archiving Communication System and its principals were firstly discussed in radiologists' meetings in 1982. Despite Duerinckx and Pisa [26], reportedly, first used the term in 1981, the credits for introducing term went to Samuel Dwyer and Judith M. Prewitt [27].

Initially, hospitals purchased film digitalizers in order to convert physical images – such as X-Rays – into digital format [28]. To create a better alternative to this process, the PACS [23] was developed as a way of creating a cheap storage solution with fast retrieval and multiple and simultaneous access from different workstations. Thus, this system can be defined as a healthcare technology focused on storage – long and short-term –, retrieval, management, distribution and presentation of medical images, allowing the healthcare institution to view and share all types of images internally or externally [2].

When deploying a PACS, the environment in which it will work must be considered. This means that there is no general PACS architecture since it depends on the requirements of the departments where it is installed and the technological resources of the institution. However, it is possible to identify the main components of this system [2, 23, 29, 30]:

- **Acquisition Workstations**: primary acquisition device (imaging acquisition systems);
- **Secure network of transmission**: it can be a local or wide area network for distribution and exchange of patient information;
- **Reading stations**: workstations or mobile devices for viewing, processing and interpreting images;
- **Storages**: short or long-term storages for storing and retrieving images and related data;
- **Data management system**: specialized computer that manages data work-flow on the network.

The above mentioned components are the most important and the ones that should be present in all PACS. Although, there are other components that are frequently used in this kind of systems and that can increase the efficiency of the system. An example of these components is a database server. Such device is a robust central computer capable of processing information in high speed. [28].

With the previously indicated components it is possible to establish many workflows between them. As an example, the following can be described [31]:

1. Modalities workstations send objects to a gateway that assures that information in the object are correct.

2. After being checked and eventually corrected, objects are sent to an archive module;

3. The archive module sends the objects to reading workstations that request them.

Since the implementation of these systems has been massive, it is legitimate to assume that the advantages that institutions can profit of are several. One important advantage for healthcare institution's managers is the decrease of storage costs by using digital storage instead of film based. Besides this, the system allows a faster and more reliable image retrieval along with remote access, saving much time to physicians. PACS is also a good choice because it enables an easy integration of medical images in the hospital's information systems. Moreover, the digital transmission of medical images and reports, and an effective workflow management permits a faster diagnosis. At last, PACS allows an easier security implementation, which is a great advantage. In spite of the advantages of these systems' implementation being major in comparison to the disadvantages, these are not null. For instance, given that these systems are extremely complex, it is possible that their end users will not be capable of efficiently use the system. Thus, it is important to continuously make an effort to simplify and improve the user interface, in spite of the system's complexity [30, 32].

2.3 DICOM STANDARD

When working with different imaging systems, to establish a connection and being capable of transferring data between them, the implementation of a common speech between those systems is necessary. Thus, in what concerns to medical images, the development of a standard that took care of communication protocols between stations and the files' structure was need. In this context, in 1985 the National Electrical Manufacturers Association (NEMA) published the ARC-NEMA, posteriorly renamed, in 1993, to Digital Imaging and Communications in Medicine [33]. This is a structured multi-part document where each part describes a single layer that can be used for different services and objects [34].

DICOM can be defined as *the* international standard for medical images and information related with them, being a very commonly used standard in what concerns to storage, transfer, and visualization of medical images [35]. In other words, DICOM is used as a data transfer protocol about diagnosis, treatments, images, and any information related with medical images. This standard enhances connectivity, compatibility, and workflow of medical images, enabling data exchange with the quality needed for clinical use [25, 34, 36–38].

In the following subsections, the basic concepts of the DICOM standard will be addressed. Section 2.3.1 exposes the structure of the exchanged DICOM objects and the data syntax

used in this standard. Then, in Section 2.3.2, are referred the main services implemented by DICOM as well as the established Service-Object Pairs (SOPs).

2.3.1 *Object structure*

As said before, this standard is responsible for the definition of data structures for medical images and associated data. In what concerns to data structures implemented by the DICOM standard, the central components are the Information Object Definitions (IODs). These define a data model used to represent objects – e.g. DICOM images or DICOM Structured Reporting). IODs represent a class of objects and help the application entities having an overview about the object that will be exchanged.

Every DICOM object has a "header" that contains a list of the attributes concerning to the object. In this list, in what concerns to DICOM images, it is possible to find information like patient related data, performed procedures, image's dimensions, etc. Here, each attribute has a well defined meaning. Sometimes, to avoid ambiguous information, Unique Identifiers (UIDs) are used. These identifiers are appropriate in parameters such as case study number or image series. An important characteristic of these attributes is that they are organized into groups and each group refers to a kind of data. For instance, group 8 includes information about the examination and group 10 includes the patient's data.

Attributes – or data elements – have a proper structure itself. These are composed of four fields [39]:

- **Tag**: is the element identifier and is always represented in the format (GGGG,EEEE), being "GGGG" the group and "EEEE" the element number;
- **Value representation**: is an optional field and depends on the transfer syntax that is being used. Describes the data type and format of the attribute's value;
- **Value length**: denotes the length of the attribute's value;
- **Value field**: contains the actual data element value.

In Appendix A.1 is possible to find the data dictionary that was used to develop this project. There, it is possible to identify some of the data elements an image can contain and the values that each of the above mentioned fields can take. As a practical example, Appendix A.2 exposes the content of a real medical image metadata.

2.3.2 *DICOM services*

Besides the objects' structure, the DICOM standard also describes some services that should be implemented and how they should be implemented. There are two main groups of

services: *composite services* and *normalized services*. Generically, the first mentioned service applies to composite objects and it is oriented to the image and data exchange. The second one applies to normalized objects and it is oriented to information management [25, 38, 40]. When dealing with these services, some messages have to be exchanged. Messages that refer to composite services are identified by a "C" in the beginning, for instance, *C-STORE*. Messages that refer to normalized services are identified by an "N" in the beginning, for example, *N-DELETE*.

The most used DICOM services are [34, 41]:

- Verification service;
- Storage service;
- Storage commitment service;
- Query/retrieve service;
- Modality Worklist;
- Modality Performed Procedure Step;
- Print service;
- Display Gray Scale Standard;
- Gray Scale Soft Copy Presentation State Storage.

Before explaining in a more extensive way the most important services of the DICOM standard, it is important to understand the concept of SOP Class. When working with the previously mentioned services two roles, that function as a pair, can be distinguished [34]:

- Service Class Provider (SCP): role of the service's provider;
- Service Class User (SCU): role of that service's user.

These roles are not steady. Meaning, workstations and archive can both be SCP and SCU depending on which one is sending objects to which [34]. Hence, a SOP Class must be defined. This is a function agreement defined by the combination of an object and a service group that decides what can be transmitted between each pair of stations. This means that the two stations in communication must support the same service and object besides the different roles. This is an important part of the DICOM standard because not every station supports every type of object and service [42].

Now one is able to start exploring the most important services for this case study.

Storage service and Storage commitment service

In the DICOM standard is described a service that is responsible for the data's safe commitment in the storage. When a SCU requests (`C-STORE`) an object storage to an SCP, it confirms the acceptance of the request by sending a message. However there is no guarantee that the object was safely saved. It only assures that the request was accepted. After a SCP's acceptance of the storage and safekeeping request, the notification of storage commitment will only be successful if the SCP safe-stores all objects.

If a physician wishes to delete an object from the internal storage of a modality's equipment, he sends a storage commitment request (`N-ACTION`) to assure himself that the files that he is deleting from the internal storage of the equipment are safely stored and, ergo, can complete the action [43].

Query/Retrieve service

The query/retrieve service is responsible for querying a content to a DICOM archive (usually PACS server) and eventually retrieve content to any DICOM node. This process comprises two phases [44]:

- **Query phase:** in this phase, a first DICOM node (SCU) queries (`C-FIND`) a second node (SCP) for a specific data. This query can contain parameters that help the information's selection.
- **Retrieve phase:** The SCU node requests some DICOM data retrieval (`C-MOVE/C-GET`) to the SCP. In this phase it is implied a data transfer from one component to another, if matches are found.

2.4 STORAGE SOLUTIONS

Since the 80's, all data was stored in RDBMS, meaning, SQL databases. Although, with the appearance of web applications and big data, this solution faced some new challenges and new users' needs, especially when dealing with large scale systems where concurrency is a keyword. This made Stonebraker et al. [45] declare "the end of an architectural era" [46]. In this context, NoSQL databases have risen, keeping up the promise of high user satisfaction along with a low management and operational cost [14].

In the following Sections – 2.4.1 and 2.4.2 –, both SQL and NoSQL solutions will, respectively, be explored with the purpose of exposing what conducted to the choice of a NoSQL solution over a mainstream SQL solution.

2.4.1 *SQL databases*

In 1970 E. F. Codd [47] was already concerned about data's organization in databases and how these should be transparent to the user. He believed that the user did not need to be aware of data's organization on the database. Also, this investigator was convinced that unstructured databases could lead to inconsistent and redundant information. Based on these ideas and with the desire to create a simple structured data model that all developers could have the same understanding about, he developed the *relational model* which gave birth to the famous *relational databases* [48]. These are also known as *SQL databases*, named after the Structured Query Language (SQL), which is the query and maintenance language used on these systems.

A relational database is a collection of data organized in tables that can change along with time and can be accessed and altered as if they were organized in non-hierarchic relations. Using this model, data is addressed by values, not positions. This would boost the productivity of programmers and users since items' position is always subject to change and, thus, it is difficult to keep track of it [48, 49]. The main characteristics of this data model are [50]:

- Structural aspects;
- Rules to insert, update and delete data;
- Support for a sub-language at least as powerful as relational algebra.

In a simplistic way, relational databases can be defined as a set of tables containing data that fits into categories – columns – previously defined. It is important to notice that each row in the table is a unique data tuple. Besides, data on the database can be accessed and altered by the user without having to restructure the data model [5].

An interesting and attractive set of features that are important to notice in a relational database are the so called *ACID transactions*, where ACID stands for Atomicity, Consistency, Isolation and Durability. Hereafter is described the meaning of each one of these characteristics [16, 46, 51]:

- **Atomicity**: all operations of the transaction are executed or none of them (all or none);
- **Consistency**: at the begin and end of the transaction, the database will be consistent (the resultant tables will be valid);
- **Isolation**: transactions are independent;
- **Durability**: the transaction will not be reversed at the end of the process (the database can survive to system failures).

The concept of relational database brought significant improvements in terms of productivity to programmers and end users because of data independence, structural simplicity and relational processing. Besides, since the acceptance of the model was huge, it was necessary to build a Relational Database Management System (RDBMS) to accomplish all the previously mentioned objectives and gains [48]. Having set this evolution, IBM decided to create the first prototype of a RDBMS in 1976, the *System R* [52]. Although, only in 1979 was created a fully commercially available RDBMS, by the giant Oracle [45, 53].

Over times, more and more companies developed their own RDBMS. Nowadays, according to a study carried out by DB-Engines and released in January 2016 [54], the market leaders on this field are:

1. Oracle;
2. MySQL;
3. Microsoft SQL server.

In spite of all the advantages that E. F. Codd found on relational databases, it seems that with the market's evolution, this solution was not enough for its storage needs . The society's demands for data storage are increasing as much as the data volume, sometimes making the existing storage solutions inadequate or just too small. This is responsible for the need of scaling up the storage systems. When it comes to RDBMS facing this problem they do not perform so well. To scale a relational system it is necessary to replace the existing machines by a more powerful (and expensive) one – vertical scaling – which is a hold-back for companies. Besides, when this upgrade is not enough to support all data volume the solution is the system's distribution across multiple servers. At this point the problem is not just financial. Relational databases are not easily adaptable to distributed environments because the join between tables across multiple servers is not easy, since these databases are not designed to deal with data partitioning. Furthermore, when data does not easily fit into a table, it can add extra complexity to the data model, which should be avoided. The long set of RDBMS's features can also be a disadvantage because they add unnecessary complexness to the database. Also, SQL, the query language used in these systems, can be a problem since it only can deal with structured data [5, 46, 51].

Having all these issues in mind, researchers made an effort to find an alternative to SQL databases. In such way, a panoply of NoSQL databases were developed. In the following Section (2.4.2) are explained the fundamentals of these new databases and why they are a good alternative to SQL ones.

2.4.2 *NoSQL databases*

With the expansion of the internet, web applications have grown in popularity and services that required big data/data-intensive products started to appear and changing the market's needs. With the web application's adoption increase, the bottleneck of the service will rely on the storage service [51]. This evolution started to unveil the weaknesses of traditional Database Management System (DBMS) since scaling is a key-point of cloud computing [16]. Since the late 90's the need of letting go the concept of relational databases and SQL started to be felt [55]. Thus, in 1998 Carlo Strozzi [56] coined the first use of the NoSQL term. However, at this time, he did not refer to it as a non-relational database as it is used nowadays. This concept concerned to an open-source relational database that did not offer a SQL interface. About ten years later, in 2009, Eric Evans [57] reintroduced the term as a way of deviating from the relational model.

The term NoSQL, in fact, works exactly as two words – "No" and "SQL". However, this has been interpreted by the community as Not Only SQL since NoSQL databases mostly do not use SQL as query language, but some do. Probably, the creators of the term wanted to refer themselves to NoSQL as a "No relational" or "No RDBMS" even making some researchers use the term *NonRel* as a more intuitive alternative to NoSQL. Independently of what the acronym means, the NoSQL database designation comprises all databases that are not based on the RDBMS principals [14, 16, 58].

As scientists gained more experience on the development of distributed systems, where partitions are inevitable, they started to realize that it was almost impossible to achieve both consistency and availability at the same time [5, 59]. This "fight" between availability and consistency was first described in 1997 [60] as a discussion ACID vs BASE, where BASE stands for Basically Available, Soft state, Eventually consistent. As its acronym shows, this concept is exactly the opposite of ACID, which is a characteristic of SQL databases. While ACID forces consistency at the end of each operation, BASE accepts that the database can be in a transitory state and, thus, only eventually consistent. This consistency state allows the system to support some partial failures without crashing the whole platform. Yet, not everything is perfect in this BASE approach. BASE requires a much more intensive analysis of operations and temporary inconsistency cannot be hidden from the end user. This requires a much more careful picking of inconsistency opportunities [51, 61].

The ACID vs BASE discussion was the first one to evolve at this level. Nevertheless, there are more theories that have emerged from this availability vs consistency paradigm. In 1998 appeared the CAP theorem, however it only was published in 1999 by E. Brewer and A. Fox [62]. CAP stands for Consistency, Availability and Partition Tolerance and it states that it is impossible to achieve three of the three CAP characteristics in a distributed system. Still,

systems can be improved towards the maximum proximity of the three characteristics [55, 61–64]. Hence, three states are possible [55, 62]:

- **CA**: databases can only provide consistency and availability at the same time if they give up of the partition of data into multiple server peers;
- **CP**: when working with a partitioned database, it is only possible to achieve consistency by giving up of temporary availability because, at some point, transactions will be blocked until partitions heal in order to avoid inconsistency.
- **AP**: when working with partitioned data across multiple servers it is only possible to achieve total availability when giving up of full consistency. Meaning, the system will be eventually inconsistent during a certain amount of time.

Since NoSQL databases are not specifically designed to support ACID, they are, thus, less rigid. This lets the database leave the relational model aside and create a bunch of new data models. The most common NoSQL data models are: key-value, document based and column oriented [5, 14–16].

In the following subsections each one of the above mentioned most common NoSQL data models will be detailed.

Key-value stores

Key-value stores are the ones with the most simple data model since, as the name implies, it is a database based on a model that indexes values by retrieval keys. This means that it does not need tables or semi structured documents because data is organized as an array of entries. This suggests that this model can handle both structured and unstructured data. In these databases all actions are performed by key. Meaning, if each row has a unique identifier key, it is possible to identify, alter, delete and insert rows uniquely by giving their identifying key [5, 55].

Document stores

Document stores are able to store many kinds of documents. This is possible by encapsulating documents in a specific document type – JSON or XML. These documents support nested documents, lists, pointers and scalar values [14, 46, 65]. Additionally, these can contain structured data such as a tree with nodes. Thus, each record – or document – can be an autonomous set of data and can also describe a schema that is not necessarily dependent on other documents [66]. It is also important to notice that the attribute names are dynamically defined for each document in runtime [46].

Documents can be indexed, enabling requests by key and making this kind of store a subclass of key-value stores [14, 65]. If these keys are not provided by the user, the system

can generate its owns. Keys can also be used as a way to establish relations between documents in a store [66]

As a way of knowing what does each document refer to, these stores group the documents in collections. In these collections can be included any documents about the same "matter" in spite of their own schema. This feature helps the retrieving process when working with data requests of a specific collection. Another strategy that can be used to speed the retrieving process is the inclusion of a "type" field to label the document. Besides this, the inclusion of documents in collections is not a developer or user's concern. The system is able to identify documents and decide whether they should or not be included in a collection [66].

Column stores

The first column oriented data stores appeared in 1969, being a system for information retrieval in biology [67, 68]. In 1975 was developed the second column store and the first one with application in healthcare, being a storage solution for clinical records [69]. However, only in 1993 was developed the first commercially available column store solution, matured by KDB. About 10 years later, the first open source solution rises as *MonetDB* [70].

In a generic way, *column stores* – also called *Extensible Record Stores* – store data in columns instead of rows. It stores each column of each table separately, as sections of columns, where values of the same column are store contiguously, compressed and densely packed. This compression allows a fast execution of operations. In order to maintain together the data referent to the same record, data is stored in a away that, for instance, the fifth entry of column A matches the fifth entry of column B. This allows the access to data individually by column as a group instead of individually row-by-row [65, 67, 71].

2.5 HADOOP-BASED PROJECT - HBASE

Hadoop-based Project (HBase) [72] is a distributed, persistent, strictly consistent and with near-optimal write, column oriented database system built on top of HDFS to create an highly scalable and high performance platform capable of dealing with large heterogeneous datasets (BLOB, CLOB, etc.) [73]. However, as we will see bellow, all HBase operations' execution are based on a row-key. In such way, some investigators that consider this database more of a key-value store or a hybrid architecture between key-value and column oriented stores [17–20, 74].

HBase was started in 2006 by Chad Walters and Jim Kellerman at Powerset and it was initially developed as a contributive part of Hadoop. Although, some time later, it gained its own project and became Apache Software Foundation's. This project is a middleware

that runs on top of HDFS and is the application that Hadoop uses when real time data access is required in large datasets [17, 19, 20].

The following sections address the HBase system architecture (Section 2.5.1), its data schema (Section 2.5.2), and the basic operations that can be performed over an HTable (Section 2.5.3). At last, Section 2.5.4 gives an overview about the usage of HBase in healthcare applications.

2.5.1 *System architecture*

HBase's architecture can be described as a set of three systems orchestrated to make use of the existing systems on top of which it is built (HDFS and ZooKeeper [75]). These three systems are:

- Client library (API);
- Master server;
- Region servers.

Hereafter is described the operation mode of each one of these three components as well as HDFS and ZooKeeper.

The *master server* [19] is responsible for assigning regions to region servers and dealing with the load balancing across them, maintaining the cluster's state. In such way, it is never involved in the storage or retrieval path and does not assign data services to region servers. Furthermore, this component is also responsible for taking care of schema changes and other metadata operations. It is important to notice that this server has the help of ZooKeeper on the accomplishment of its tasks.

In what concerns to the *region servers* [19], these are responsible for all read and write requests of the regions they serve. Region servers deal directly with the client application and, thus, are responsible for handling data-related operations. These servers can be added or removed during runtime to adjust the system's workload changes. These hardware alterations can be made without affecting the system.

The *Client Application Programming Interface (API)* [19] is the primary client interface to HBase and is responsible for providing the user all the functionalities needed to store, retrieve and delete data from the database.

HDFS is a distributed filesystem designed to tolerate faults and deploy on low-cost hardware [20, 76]. It provides a high throughput access to applications, even when working with large datasets. This is where files are stored and it guarantees that files are written across multiple physical servers so they are never lost. The HDFS is responsible for storing both metadata and application data, with each of them being stored on a different kind

of server. *NameNode* is the master server and is responsible for the file system's metadata. *DataNodes* are servers dedicated to application data. These servers are fully connected using reliable protocols (TCP-based). The HDFS namespace is hierarchical and represents files and directories. Files and directories are represented in the NameNode by *inodes*, which are structures that store attributes like permissions, modifications and access times. File contents are split into large blocks that are posteriorly replicated through a defined number of DataNodes. Each one of these blocks is represented in the DataNode by two files: one with the actual data and one with the metadata of the block. The namespace maintains the mapping of file blocks to DataNodes. In a practical way, the NameNode receives a reading request, localizes the DataNodes that have the requested data and assigns the job to the one that is closer to the client. When working with writing, the NameNode chooses the DataNodes that will carry the replicas. It is important to notice that the HDFS namespace is always kept in RAM, speeding up the access to the database's content. When starting up the system, an *handshake* is perform between the NameNode and DataNodes. During this process, namespace's ID is checked as well as the software version. This prevents the join to the namespace of DataNodes of other namespaces. The software version is important to prevent data corruption and loss due to incompatibilities [19, 73, 77, 78].

The distributed coordination service of Hadoop is called *ZooKeeper* [17, 75, 79]. This is a coordination service developed to manage client requests in a wait-free way – which is important for fault tolerance and performance – with order guarantees. Furthermore, this system can provide a set of tools that help the database dealing with partial failures. ZooKeeper presents an hierarchic filesystem, providing an abstraction of a set of nodes (*znodes*), where data is stored and from where clients can read and write data. This allows a better organization of the system's metadata for coordination reasons. There are two types of *znodes*: regular and ephemeral. *Ephemeral znodes* are created by the client and can be deleted by the system (when the session ends) and by the client itself. It is important to notice that these nodes can not have children. *Regular znodes* are created and deleted explicitly by the client and, in contrary to ephemeral *znodes*, are able to have children. Each client application has its own regular *znode*. Then, this *znode* can have as much ephemeral *znodes* as the number of processes that it has running on the ZooKeeper. An important characteristic of *znodes* is that they are labelled with timestamp and version, allowing the client to execute conditional orders based on the *znode* label. ZooKeeper is also provided with a set of configurable primitives allowing the developer to change the away the coordination is executed. An interesting and important feature of this coordination service is that its namespace is kept in memory, allowing high throughput and low latency times, and implements data replication through many servers, achieving high availability. However, clients only link to a ZooKeeper service.

2.5.2 *Data model*

HBase stores data into labelled tables composed of rows and columns, resulting in table cells [17, 80]. These cells have a time-stamp associated, automatically defined by HBase, that corresponds to the time of insertion of the corresponding data in the table. Each table row is identified by a key and tables are sorted by these keys. By default, this sort is byte-ordered so the related rows stick together. Since the table is sorted using these keys and can be accessed through them, they can be seen as primary keys.

In HBase, a column is defined by a *column family* and a *column qualifier* separated by a colon – e.g.: `Patient:Name` [8, 17, 80, 81]. Thus, every column family is composed by columns with the same prefix. Table's column families must be pre-specified at the definition of the database schema and have a set of storage properties that help the system decide whether it must save values in memory, which ones are encoded or compressed, between other characteristics. However, column qualifiers can vary greatly from row to row since they can be added dynamically in runtime, as long as the column family that they belong to already exists. Column qualifiers work as a kind of index to help the identification of the data that can be found on that column. Knowing this, one is capable to understand that a cell is identified by its table, row-key, timestamp and column. It is important to note that columns can also have associated an attribute that can be used to take some kind of annotation to the column, for example, if its value is protected or not.

An important aspect of HBase schema is that tables are organized into *regions* [8, 17, 80, 81]. Initially, tables can be included in only one region. However, when they start growing and exceed a certain configurable limit, the table is split into more than one region. These regions are characterised by the first (inclusive) and last (exclusive) row-keys. Regions are units that are distributed across several nodes of the cluster.

2.5.3 *Basic operations*

HBase supports a group of basic operations, usually called CRUD operations which stands for Create, Read, Update and Delete [19]. In HBase's case, these operations are:

- Put;
- Get;
- Scan;
- Delete.

The `put` method allows the insertion of a certain column value for a given row key. Hence, to insert it on the HTable, it is necessary to identify the row key, the column family,

the column qualifier, and the value that the user wants to put. This method is also used to update values. To do so, it is only necessary to perform a new `put` for the key and column – or columns – that should be updated.

The `get` method is the one used to retrieve specific values of an HTable. To achieve that, it is necessary to specify the key and the columns of the respective value that we want to retrieve.

The `scan` method works like a `get` but it has the ability to retrieve multiple rows and there is no need to give the key of the row – or rows – that the user wants to retrieve. To do so, it is necessary to associate a filter to the scan with the value of the columns that we want to use as search criteria. Moreover, these filters allow the implementation of different compare operators, allowing the search of results that fit a range of values.

The `delete` method allows the deletion of data from the HTable. To do so, in first place, it is necessary to specify the row key where we want to perform a deletion. Then the deletion must be narrowed down. Meaning, the column family and/or the column qualifier must be specified.

It is important to notice that all these operations are performed over byte array values. In such away, all values must be converted to that data type before any action.

2.5.4 *Applications in the healthcare industry*

The healthcare industry is becoming increasingly concerned about the capability that their storage systems have to tolerate data partition, since the amount of daily generated data is becoming bigger and bigger [82]. Ergo, in a generic healthcare market, there are some applications that use HBase as a storage system. Between these, it is possible to refer the systems described by Jin et al. [83] and Sobhy et al. [84]. These are both systems that use the advantages that HBase, Hadoop and HDFS provide to create a distributed, fault tolerant, low cost, and highly scalable solution to store Electronic Health Record (EHR). However, these aren't already prepared to deal with medical images records.

Another application of Hadoop's environment in the healthcare world is the storage for medical image processing softwares. Between these software applications it is possible to find Content Based Image Retrieval (CBIR) systems [85–87]. These use this kind of databases to store a set of features that are extracted from the original images.

In what concerns to the actual storage of medical images or its metadata, there are studies that propose, as a possible option, the implementation of the Hadoop's panoply as back-end [88–90]. However, there are already systems that actually describe a storage system for medical images using these components. An application that is worth to reference is the *CloudDICOM* [91]. This is an application layer constructed on top of Hadoop and HBase to access and distribute medical images through a web-based system. This study is focused

on the system's architecture, workflow, and data schema and here, both metadata and DICOM objects are stored in HBase, without concerns about security. A flaw that is possible to identify in this work is the lack of a description of the obtained results in terms of the systems' performance.

2.6 Cryptographic techniques – symmetric key encryption

Cryptography can be defined as the science of writing in code [92]. This was, reportedly, first used in 1900 B.C. in Egypt. This science has three main goals: protect from theft, alteration, and authentication purposes.

When using unreliable means of communication, it is important to use the techniques described by cryptography in order to protect the exchanged data. Nowadays, a particular unreliable channel of communication is the Internet. Since this is the communication method used to transfer information in this project, it is important to study techniques to provide protection of transferred data [92].

There are two main types of cryptographic techniques – symmetric and asymmetric key encryption [93]. Let us consider a client that needs to exchange a message with a provider without letting an attacker now which message is being exchanged. Briefly, symmetric key encryption uses the same secret key to encrypt and decrypt. This key is kept by the and cannot be revealed to other parties that are not supposed to have access to the original information in the clear, i.e., the attacker [94, 95]. On the other hand, asymmetric key encryption is based on the usage of two different keys for encryption and decryption. In this case, a public key is used to encrypt data and a private key is used to decrypt it. However, asymmetric key encryption is a relatively slow process in comparison to the symmetric key encryption [93]. Symmetric algorithms have faster implementations in hardware and software, which makes them more suitable to deal with large amounts of data [94]. In such way, the symmetric key encryption is the most promising technique in this project's context.

Symmetric key encryption algorithms can be subdivided in two main groups: stream and block ciphers [96, 97]. In a generic way, stream ciphers encrypt a single bit at a time, producing a random bit string. On the other and, block ciphers encrypt a specific number of bits at the same time. Despite the high speed of transformation presented by stream ciphers, these show a low diffusion – because a symbol of plaintext is represented by only one ciphered symbol. Besides, these are more susceptible to insertions and modifications. On the other hand, block ciphers are more powerful, more secure, and more easily implemented than stream ciphers and, thus, can be used to protected the integrity of data and authenticate at the same time, which is not possible using stream ciphers. Furthermore, block ciphers are more useful when the data being encrypted is previously known. In such way, this work will be focuses on symmetric key encryption using block ciphers.

Symmetric key encryption algorithms are based on two concepts: block and key [94]. The block cipher is the metric defining the number of bits being encrypted at the same time. The key defines the bits used to cipher the plaintext. In a generic way, a cipher is as secure as its key's number of bits.

There are several algorithms to perform symmetric key encryptions [93]: Data Encryption Standard (DES), 3DES, Advanced Encryption Standard (AES), RC4, RC6, BLOWFISH. However, the most common ones are DES and AES.

The DES was the first symmetric key encryption standard created [93, 94]. It has a fixed size for cipher blocks – 64 bits – and keys – 56 bits. Given the key's number of bits, it is possible to understand that there are 2^{56} possible keys. Thus, a disadvantage of this algorithm is that it is easily attacked using brute force.

The AES is an algorithm created to replace the DES [93]. This one needs a fixed block size of 128 bits. However, the keys' sizes can vary between three possible values – 128, 192, and 256 bits – which are bigger than the required sized on the DES. An advantage of using the AES encryption method is that it is fast, flexible, easily implementable in different platforms and in small devices. Since this algorithm's keys are bigger than the ones used in DES's, it is a more secure method.

Despite all the referred characteristics of these encryption standards, when working with databases, it is important to maintain their searchability [98, 99]. To do so, several techniques can be applied. One of the most common is the deterministic encryption. This type of encryption ensures the ciphered text will always be the same when given the same input. The AES, when used with a constant initialization vector, is a fully deterministic algorithm for the same given plaintext, block cipher and key [95]. The same applies for the DES.

2.7 Personal data protection in healthcare

As referred along the previous sections, working with cloud storage solutions has many advantages. However, these raise important issues in what concerns to data security and privacy [21]. The healthcare business is one of the areas that requires a reinforced concern about the security of stored data since it can contain sensitive information. As said before, medical images' metadata can contain information such as names, personal dates or even a personal medical history. These can be sensitive due to several reasons. For instance, it can reveal personal and business data that can become a competitive advantage or disadvantage if it becomes public [100]. Actually, the individual's privacy has been declared by Meglena Kuneva, European Consumer Commissioner, as the "new oil of the Internet and the new currency of the digital world" [101].

Therefore, some standards and guidelines have been structured to protect personal data. For instance, the United States of America developed the Health Insurance Portability and

Accountability Act (HIPAA) and the European Union has implemented the Data Protection Directive (EU directive 95/46) [102] and the Directive on Privacy and Electronic Communications (EU Directive 2002/58) [103–105].

Sections 2.7.1 and 2.7.2 further address these privacy preserving guidelines, while Section 2.7.3 exposes some of the applications that already comply with these guidelines.

2.7.1 *Health Insurance Portability and Accountability Act – HIPAA*

Since information violations were anticipated, in 1996, when the HIPAA standard [106] was created, a privacy rule was integrated on it. This rule states that entities may only use or disclose information that is de-identified. Besides, it implies that there are eighteen characteristics that allow the identification of a subject and, thus, must be protected. Between these characteristics the following can be found: names, addresses, dates related to the subject – e.g., birth dates –, contacts, or any other unique identifying numbers. This rule only applies to covered entities, in which are included health plans, healthcare clearing houses, and healthcare providers who transmit health information electronically. These entities can be institutions, organizations or persons.

2.7.2 *European directives*

Regarding the European legislation, the data protection directive is the oldest active directive in what concerns to the protection of the individual's data. According to the Article 2 of the EU Directive 95/46, personal data refers to any information relating to an identified or identifiable person. On the other hand, an identifiable person is someone who is eventually identifiable directly or indirectly. Both directives, – EU Directive 95/46 [102] and EU Directive 2002/58 [103] – state that "member states shall protect the fundamental rights and freedoms of natural persons, and in particular their right to privacy with respect to the processing of personal data", and on free movement of such data. When working with electronic communications, the confidentiality of these communications must be guaranteed.

2.7.3 *Personal data protection compliant applications*

Having in mind all these concerns about secure access to medical images, there already are some cloud storage solutions for medical images that take personal security in consideration when storing them [107]. Between these solutions it is possible to identify two main groups: solutions that protect the whole image and solutions that only protect sensitive data – in this case, the image's metadata.

Between the solutions that protect the whole image, there are many security approaches. For instance, Cao et al. [108] uses Digital Envelope (DE) and Digital Signature (DS) algorithms to protect the image, while Dzwonkowski et al. [109] uses a Cosine Number Transformation (CNT) quarterian based encryption, and Castiglione et al. [110] uses a watermarking method. About the solutions that only protect sensitive data, Xiang et al. [111] applies a chaotic encryption technique, while Ribeiro et al. [112] uses a Posterior PlayFair Searchable encryption. There is also another interesting approach, described by Silva et al. [113], that relies on the implementation of a DICOM router which is responsible for the encryption and decryption of all the exchanged messages.

In spite of the panoply of secure cloud storage solutions for medical images, there are not yet solutions that comply with the personal privacy protection guidelines and use Hadoop's environment as backend to store these images which, as said before, can bring great advantages at several levels.

3

RESEARCH METHODOLOGY

This chapter will address the research technologies and methodologies followed during the development of this master thesis, which allowed the achievement of the objectives referred on Section 1.2. Section 3.1 briefly explains how this chapter is organized and its main purpose.

3.1 INTRODUCTION

A research methodology can be defined as a way of finding a solution for a proposed problem or an investigation that leads to the understanding of a phenomenon [114]. In such way, several directions can be followed in order to reach the problem's solution. That being said, some methodologies may be more effective than others and, thus, it is important to follow a well established research methodology in order to achieve the best possible results.

In the specific case of the Information Technologies (IT), in which is included the computer science's branch, there is a famous and commonly used research methodology – the *DSR methodology* [115–117]. This is a methodology that, in a simplistic way, is based on the identification of a problem, the design of a solution and its validation. Since this fits in this thesis' work field and allows the accomplishment of the established objectives, it was adopted as research methodology during the development of this project.

In this way, Section 3.2 explains in a more extensive way the DSR methodology's assumptions and Section 3.3 shows the practical application of the DSR concepts in this thesis context.

3.2 DESIGN SCIENCE RESEARCH – DSR

The DSR methodology can be defined as a paradigm of problem solving. According to Von Alan et al. [117] "It seeks to create innovations that define the ideas, practices, technical capabilities, and products through which the analysis, design, implementation, management, and use of information systems can be effectively and efficiently accomplished".

In another words, DSR is the course of action that leads to the useful products' design and creation, and its effectiveness analysis using stringent methods [114].

This methodology is composed of two main activities [114]: *building* – creative process that leads to new products – and *evaluating* – the assessment of its utility. In terms of process, Vaishnavi and Kuechler [118] described five steps to be taken when following the DSR methodology:

- **Awareness of the problem**: the researcher becomes conscientious about the problem's existence. He may become aware of it through many ways, including reading in various areas or staying in touch with technology developments. In this stage, the researcher formulates a *proposal* to solve the problem that was found and, thus, a new investigation begins;
- **Suggestion**: the researcher must come up with a *tentative design*. This is when different researchers – or the same one – can present different ideas to solve the problem, i.e., different tentative designs;
- **Development**: the researcher presents at least one *artefact*. These artefacts can be of different natures depending on the research field. These can be, for instance, algorithms or software. The employed methods on the artefact's development do not necessarily need to be innovative. The uniqueness is in the design;
- **Evaluation**: the constructed artefact must be assessed using criteria that may or may not be described on the proposal. The tests results must be in accordance with the expectations about the product's idealized behaviour during its design. If not, the researcher must make an effort in order to describe any discrepancy.
- **Conclusion**: the results, either positive or less positive, must be consolidated and written down. This way, acquired knowledge can be applied many times and/or be further researched.

Having these concepts in mind, it was possible to apply them in a real-life context, during the course of this master thesis investigation. The following Section (3.3) makes a connection between the above described theory and the practice by describing when each of the aforementioned steps was present.

3.3 Practical application

As said before, this project had, as conductive line, the DSR investigation methodology. Hence, the methodology steps were identifiable in different phases of the project's development.

At the beginning and after some research, it was possible to identify a shortcoming in the medical imaging storage systems: the lack of a scalable, available, low-cost and secure database for medical image's storage. Hereupon, being *aware of the problem*, proceeded to a new phase. A deeper exploration made one capable of making a *suggestion* to solve the problem: a NoSQL database – HBase –, compliant with the DICOM standard that should be able to safely store medical images by applying a cryptographic layer. Having an idea of a possible solution, it was time to start its *development*. In this stage, the first step was the choice of a versatile DICOM toolkit allowing its adaptation to the new backend and still complying with the standard. Then, the HBase's cluster was set up and the toolkit was adapted to use it – both ciphered and deciphered versions. Having the prototype fully constructed, it was time to test it and *evaluate* its behaviour comparatively to the default version. In order to achieve that, performance tests were executed and, with their results, charts were constructed. To *conclude* the project, the produced charts were analysed and all the conclusions and middle steps were documented in this thesis.

4

SYSTEM ARCHITECTURE AND IMPLEMENTATION

This chapter exposes the overall architecture and implementation of the developed prototypes. Here are explained the main decisions made during the project as well as the grounds that led to those. Section 4.1 makes a brief contextualization of the chapter, and describes its structure.

4.1 INTRODUCTION

The DICOM standard is composed of several documents that regulates various aspects concerning to medical imaging. Thus, searching for an open source implementation of those rules became the best option. In this way, and after the analysis of several DICOM software applications, the one that fitted best our needs was *dcm4che* [22]. The dcm4che is an extremely complete and versatile open-source DICOM toolkit that, given its complexity, deserves a special approach. Hence, Section 4.2 explains in more detail what it is and why it was the chosen one between several others. This section also exposes the utilities' panoply offered by dcm4che, addressing with more detail the components that were indispensable to this project's development. After all this explanation, one is capable of describing the default architecture constructed with the used utilities, as well as the workflow observed in that context.

After some research and according to the knowledge acquired and documented on Section 2.4, it was possible to understand that NoSQL databases – and more specifically HBase – show a huge promise in what concerns to the future of data stores. In this way, an HBase cluster was setup and dcm4che was adapted to it. Section 4.3 concentrates on the explanation of the HBase data schema and cluster that were setup to work as the new dcm4che backend. Moreover, this section discloses how this cluster was integrated with the DICOM toolkit.

Despite all the advantages HBase can provide, it also presents some security issues. Since it is a cloud oriented database and uses an unreliable communication channel – the internet [92]. In such way, protecting the stored data with a cryptographic layer is a clever decision. In such way, Section 4.4 presents an HBase implementation protected with symmetric

key encryption. Here is discussed the protected data elements and how this implementation was integrated with the dcm4che toolkit.

4.2 dcm4che – the DICOM toolkit

The dcm4che [22] is an open source toolkit composed of a set of utilities, each one of them playing a different role. It is implemented in Java and supports deployments with JDK version 1.6 and higher. This toolkit presents a solid implementation of the DICOM standard and it is currently on its third version. The version used as basis to this master thesis was *dcm4che3.3.7* which is a re-written version of the dcm4che1.x and dcm4che2.x, providing increased flexibility and performance.

Section 4.2.1 focuses on explaining, in a more extensive way, what the dcm4che toolkit is and why it was chosen over a panoply of other available open source softwares. Thereafter, Section 4.2.2 exposes its architecture and workflow.

4.2.1 *Why* dcm4che

With the evolution of medical IT, the development of open source tools to back this field's research started growing. In such way, the dcm4che toolkit is not alone in this battle. There is a panoply of open source frameworks [119–121] to help researchers accomplish new advances in the PACS industry.

After some research, a huge amount of software applicaitons that could eventualy fit our purpose emerged. Between those, it is possible to find IQ-DICOMTEST [122], Orthanc [123], OsiriX [124], DICoogle [125–127], CDMEDIC [128], DCMTK [129], dcm4che [22], ClearCanvas [130], Conquest [131], OSPACS [132], Xebra [133], between others. In fact, in 2013, Medfloss [119], listed 11 open source projects, ranked according users' ratings. Although, each one of them presented characteristics that made them less attractive than dcm4che to this project's development. Some only presented a Graphical User Interface (GUI) with no source code available, others were just an upper layer of another software, and some of them stopped being developed some years ago. It was needed an up-to-date software with an accessible and flexible source code, an easy interface, and a well defined architecture. Thus,dcm4che was the only of the above mentioned software applications to fit all these demands.

The most attractive characteristics presented by dcm4che are:

- **Modular and flexible architecture**: it is organized in a set of modules, each one implementing a set of related functionalities, making it more flexible and easily adaptable;

- **Up-to-date**: it is currently being developed, being updated according to the eventual change of the standards;
- **Command line interface**: its interface enables an easy usage along with a better integration with benchmark scripts.

It is important to notice that the dcm4che toolkit contains a batch of utilities, each one of them implementing a DICOM service. These utilities resort to the previously mentioned modules to perform the needed operations and will also be used as basis to this project's development.

Being aware of the strengths and general aspects of the dcm4che, it is now important to understand how the toolkit is architected and how it works. Thus, Section 4.2.2 explores the toolkit's utilities and the workflow and architecture that they create together.

4.2.2 *Architecture and workflow*

Section 2.3.2 focused on explaining some of the most important services that are described by the DICOM standard. Dcm4che contains a set of utilities, where each one of them, in a simple way, implements a distint DICOM service. Since the objective of this master thesis is to evaluate the performance of the toolkit using different backends, only two operations – read and write – will stay in focus. The DICOM services responsible for this are the *storage service* and the *query/retrieve service*. To accomplish that, only the following eight out of thirty nine utilities were used [134]:

- **DcmDict**: lookup DICOM attribute in DICOM dictionary;
- **DcmDump**: dump file content in textual form;
- **DcmDir**: dump, create or update DICOMDIR file;
- **StoreSCP**: DICOM object receiver;
- **StoreSCU**: DICOM object sender;
- **DCMQRSCP**: simple DICOM archive;
- **GetSCU**: makes DICOM retrieve requests – `C-GET`;
- **MoveSCU**: makes DICOM retrieve requests – `C-MOVE`.

Starting with the *storage service*, the main utilities used were the StoreSCP and the StoreSCU. StoreSCP works like a storage server. This component is responsible for receiving objects and storing them in the local disk. To start it is necessary to define two property values: the

Application Entity Title (AET) and the port. Optionally, the directory for storing received files can be provided. The AET works like a server name and it is one of the characteristics the client is going to lookup when trying to establish connection with it. The port is going to be used to establish connection with the client. An example of a command employed to start the StoreSCP can be found on the Listing 4.1 example.

```
storescp -c STORESCP:11112 --directory dicom/images
```

Listing 4.1: Start StoreSCP – example command.

The StoreSCU is the storage client. To connect with the server it needs to know the server address with which it wants to establish connection with, meaning, ip, port, and AET, as well as the image or images that the client wants to store. Listing 4.2 shows an example of a command to start a storage client.

```
storescu -b STORESCP@cloud80:11112 image.dcm
```

Listing 4.2: Start StoreSCU – example command.

At the end of a storage activity, it is important to update the DICOMDIR. The DICOMDIR is a file where all images are indexed and its locations on the filesystem are stored. It uses images' UIDs to construct a kind of an images tree, where the PatientID is the root, followed by StudyInstanceUID, SeriesInstanceUID and SOPInstanceUID levels [1] – Figure 1. In order to better understand this mapping, it is important to notice that a patient may be targeted at several studies and a study can be composed of several image series. A serie is the result of an examination. Although, these series can be composed of several slices, where each one of these slices represent an image, which ID is the SOPInstanceUID. To create and update this file, the *dcmdir* utility is used. It receives as parameter the DICOMDIR and the locations of the images to be indexed. Examples of the DICOMDIR's creation and update commands can be consulted on Listings 4.3 and 4.4, respectively.

```
dcmdir -c dicom/DICOMDIR
```

Listing 4.3: Create DICOMDIR – example command.

```
dcmdir -u dicom/DICOMDIR dicom/images
```

Listing 4.4: Update DICOMDIR – example command.

Before explaining the query/retrieve model, it is important to understand that there are two different models that can be used to execute query/retrieve protocols: the `C-GET` – executed by the GetSCU utility – and the `C-MOVE` – executed by the MoveSCU utility, the major ones being: security, flexibility and impact on the system's architecture. In terms of

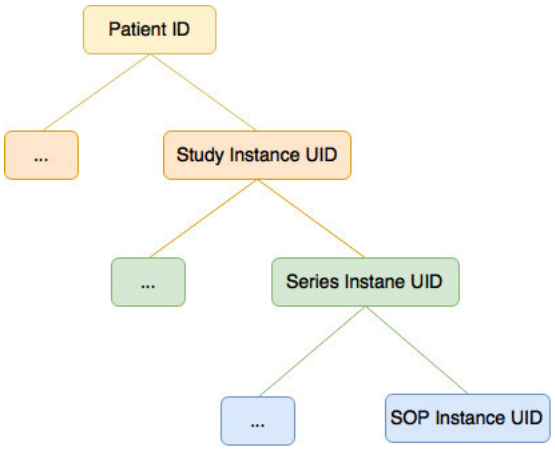


Figure 1.: DICOMDIR levels (Adapted from [1]).

security, `C-MOVE` is a better option because it needs an intermediary utility (StoreSCP) to store the received images. Meaning, the client is not an open channel to receive images. It needs to be recognized as the correct machine to receive the data by identifying the *StoreSCP* AET. In terms of flexibility, `C-MOVE` also presents some advantages. When retrieving an image, this protocol requires the establishment of two distinct connections – one to execute the query and another to retrieve the contents. In this way, the negotiation of the transfer syntax[1] being used is much more flexible. While in the `C-GET` model the transfer syntaxes are listed by the GetSCU and the DCMQRSCP needs to pick one, in the `C-MOVE` model, the transfer syntaxes are listed by the DCMQRSCP according to the query being made by the GetSCU and the client just needs to agree with it. In terms of architecture, it is now clear that the `C-MOVE` model requires the usage of the StoreSCP besides the query/retrieve client and server, while `C-GET` does not need it.

To the present case study it is unnecessary to add complexity to the system's architecture and the latency of extra connections that need to be established. Since the only thing being tested is the overhead introduced or removed by a new backend, this is not important, as the same backend can also be implemented in a `C-MOVE` model.

Now, one is in condition of understanding the architecture of the query/retrieve task. The DCMQRSCP works as the query/retrieve server. It is responsible for receiving a query, processing it, and retrieving the adequate content to the client. To start this utility, it is necessary the specification of the component AET, the port from which it will establish connections and the path to the DICOMDIR where the images are indexed, as is possible to see on Listing 4.5.

```
dcmqrscp -b DCMQRSCP:11113 --dicomdir dicom/DICOMDIR
```

1 The transfer syntax corresponds to the image's encoding/file format and the network transferring methods [135, 136]

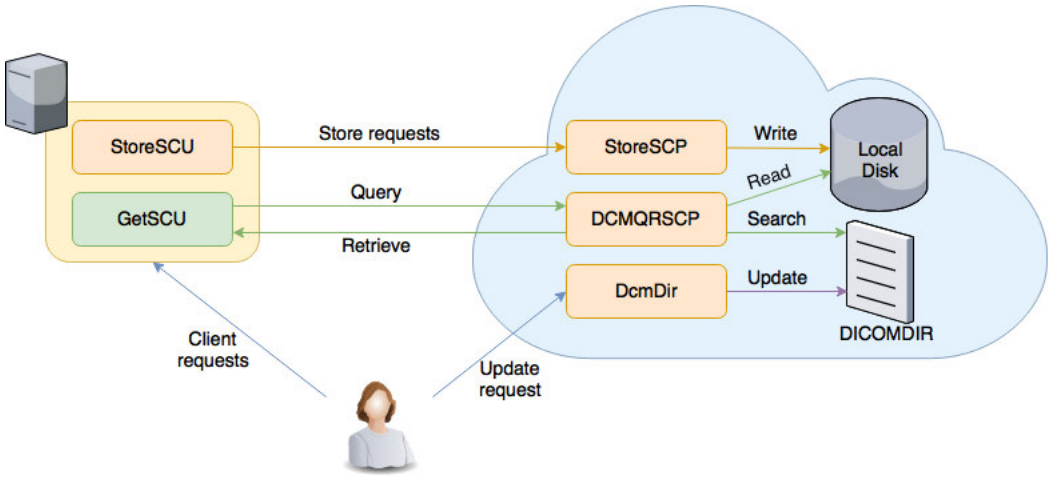


Figure 2.: Dcm4che default – architecture and workflow.

Listing 4.5: Start DCMQRSCP – example command.

In what concerns to the GetSCU, it works in a similar way to the StoreSCU. It is responsible for sending a query to the query/retrieve server and receiving the retrieved content. In order to correctly establish connection with the server, the parameters that must be specified are: server's AET, port, and ip; and the query values, as Listing 4.6 shows.

```
getscu -c DCMQRSCP@cloud81:11113 -m StudyInstanceUID=1.2.3.4
```

Listing 4.6: Start GetSCU – example command.

In what concerns to the implementation of these services on a cluster, it is important to notice that the client utilities are deployed on a different machine of the server ones. In this way, it is possible to compare with more efficiency the performance of each component. Figure 2 shows the workflow and architecture that was setup for the baseline system – default dcm4che.

4.3 DCM4CHE WITH HBASE BACKEND

HBase is a NoSQL database that presents several advantages in comparison to SQL systems and local storages, as mentioned on Section 2.4. These present improved characteristics in areas such as scalability, load distribution, data replication, and availability. This can indicate that this can be a great alternative to the existing medical images storage solutions. To this project's development, it was used HBase version 0.98 and HDFS version 2.7.

Table 1.: HTable schema with example values.

SOPInstanceUID	Patient				Image				Series				Study			
	ID	Name	Birth date	(...)	Type	Date	Hour	(...)	ID	Date	Hour	(...)	ID	Date	Hour	(...)
1.2.3.4	00001	"John"	02-05-1987	(...)	"Orig/Prim"	15-10-2013	14:32:26	(...)	1.3.5.9	15-10-2013	14:31:34	(...)	1.2.7.4	14-10-2013	10:12:43	(...)
1.2.3.5	00002	"Mary"	23-07-1965	(...)	"Orig/Prim"	29-12-2010	17:47:13	(...)	1.4.7.3	29-12-2010	17:40:12	(...)	1.5.3.6	15-11-2010	21:07:11	(...)
(...)	(...)	(...)	(...)	(...)	(...)	(...)	(...)	(...)	(...)	(...)	(...)	(...)	(...)	(...)	(...)	(...)

Section 4.3.1 focuses on detailing the HBase data schema, whereas Section 4.3.2 exposes the setup HBase cluster's architecture. In terms of implementation, Section 4.3.3 exposes the adaptations performed to the toolkit, as well as some decisions taken during its implementation's course. At last, Section 4.3.4 explains the overall system architecture and workflow, as well as how the small parts interact with each other.

4.3.1 *Data schema*

Before addressing the system's architecture and the procedure taken to save the data on the HBase cluster, it is important to understand the data schema used to store it.

As explained on Section 2.3.1, DICOM objects are composed of two main parts: a "header" – containing the image's metadata – and another part where the actual graphical image is encoded. The amount of data that is written in File Meta-Information (FMI) can take major proportions and not every attribute is important to the user or, at least, there is no interest on querying the database using some attributes, such as attributes regarding to the image dimensions. In this way, only values that might have some interest for querying were stored on the database.

As is possible to see in Table 1, 4 column families were defined[2]: *Patient*, *Image*, *Series*, *Study*. Each one of these column families is composed of a set of column qualifiers referring to the column family. The meaning of each column can be conferred on Table 2. Additionally, for every table entry it is necessary to specify a row-key (see section 2.5). Since each entry refers to one image, its identifier specified on its metadata – the SOP Instance UID – was the one used as row-key.

4.3.2 *HBase cluster architecture*

HBase, on its own, has a well defined architecture, as explained on Section 2.5. However, it can be deployed in different modes depending on the work's purpose. There are three possible deployment modes [137] – standalone and distributed, where the distributed mode can be subdivided in pseudo-distributed and fully-distributed modes.

2 Notice that a patient can have several studies associated, while a study can be composed of a group of series and a serie can contain a set of images.

Table 2.: HTable columns' description.

Column family	Column qualifier	Description
Patient	ID	Patient identifier number.
	Name	Patient name.
	BirthDate	Patient birth date.
	Gender	Patient gender.
	Weight	Patient weight.
	History	Patient medical history.
Image	Type	Indicates if the image is original or derived and if it is primary or secondary.
	Date	Date of the image's creation.
	Hour	Instant of the image's creation.
	TransferSyntax	Image's transfer syntax.
Study	UID	Study instance UID.
	Date	Date of the study's creation.
	Hour	Instant of the study's creation.
	Description	Body part being studied.
Series	UID	Series instance UID.
	Date	Date of the series' creation.
	Hour	Instant of the series' creation.
	Modality	Type of performed examination (Computed Radiography (CR), Computed Tomography (CT), Magnetic Resonance Imaging (MRI), etc).
	Manufacturer	Acquisition equipment manufacturer.
	Institution	Medical Institution where the series was acquired.
	ReferringPhysician	Name of the patient's physician.
	Description	More detailed description of the examined body part.

The standalone mode needs only one HBase node to work. It is the mode running with simpler configuration: it does not need HDFS – uses the local filesystem – and ZooKeeper is handled by HBase in a transparent way. The usage of the local filesystem instead of HDFS leads to the possibility of loosing data. When using HDFS, it makes sure not to loose files, while the local filesystem can loose data, for instance, when a file is edited and not properly closed. Additionally, this configuration does not allow data distribution across multiple servers. This is a better option at the beginning, to learn the HBase basics and get comfortable working with it. Unfortunately, it does not allow load distribution or replication, which is the opposite of this project's goal.

About the distributed modes, they both work in the similar ways. Their main difference is that the pseudo-distributed mode runs on a single node and can use the local filesystem or HDFS, while the fully-distributed mode needs to run on multiple servers and only runs against HDFS. Like the standalone mode, the pseudo-distributed mode is a satisfactory option in a preliminary stage. It is a good way of testing configurations and prototyping HBase, but still presents drawbacks in what concerns to performance testing. The most complete and more advisable for production is the fully distributed mode and, thus, it was the one used in this project.

To run a fully-distributed HBase cluster it is necessary to deploy several HBase instances in diferent servers. It needs at least one master server – HMaster – and the remaining hosts act like region servers – HRegions. In what concerns to the filesystem, in the fully-distributed mode, the usage of HDFS is demanded. In the same way as HBase, HDFS has a master – NameNode – and slaves – DataNodes –, where the NameNode runs on the same host as the HMaster and DataNodes run on the same hosts as HRegions. About the ZooKeeper, it is demanded to have a ZooKeeper running on the cluster. It is recommended to have three or more ZooKeeper instances on the cluster, allways maintaining an odd total. Since the objective of this project is not to test the system's fault tolerance, one ZooKeeper

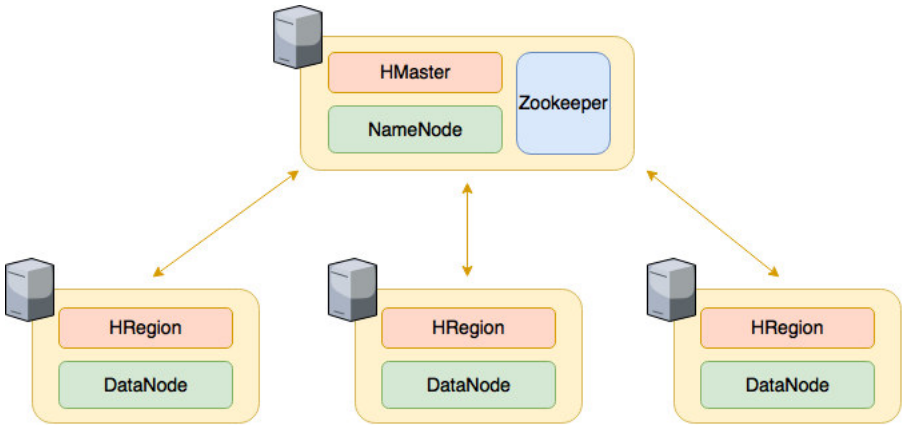


Figure 3.: Implemented HBase cluster.

instance is enough. This way, only one machine has the load of having the ZooKeeper running and the other ones are not unnecessarily burdened.

In terms of cluster, four machines were set up: one master server and three region servers. Each one of the machines was equiped with an Intel i3 CPU with four cores at 3.7 GHz, 8 GB of RAM and an SSD disk with 128GB.

Figure 3 represents the implemented cluster.

To install an HBase cluster the developer needs to define the desired architecture through several configuration files. The configuration files used to set up the cluster can be consulted on Appendix B.2.

The following section – Section 4.3.3 – explains the changes made to the DICOM toolkit in order to adapt it to this HBase cluster.

4.3.3 *Toolkit adaptations*

As said before, the purpose of this master thesis is to compare the performance of a baseline system – featuring a standard local disk as backend – versus two versions of HBase backend: one protected with symmetric cipher and one without security concerns. Thus, the dcm4che baseline version – Section 4.2 – needed to be adapted to use a new backend. Therefore, two utilities were modified: the *StoreSCP* and the *DCMQRSCP*. Only these utilities deal with read and write operations, so those were the ones in need to be adjusted.

The following subsections focus on explaining how StoreSCP and DCMQRSCP were adapted to use an HBase backend.

StoreSCP

The first change necessary on the StoreSCP was the addition of an option to the utility's command line arguments. The `-f` option must provide the HBase client configuration file as

input. This configuration file contains all the information the client needs to connect himself to the HBase cluster and interact with it. The properties included in this file are [19]:

- `hbase.zookeeper.quorum`: ZooKeeper instances' hosts;
- `hbase.rootdir`: directory where HBase will store its data;
- `hbase.cluster.distributed`: whether the HBase is deployed on standalone mode or distributed mode;
- `hbase.zookeeper.property.dataDir`: directory where the ZooKeeper's snapshots are stored;
- `hbase.master.hostname`: HMaster adress;
- `hbase.zookeeper.property.clientPort`: port that will allow the client's connection.

The values each one of these properties took can be found on appendix B.1.

In the default version, every time a C-STORE action is required, a new class – `DefaultStoreSCP` – is instantiated and, thereafter, every operation is performed according with that class's methods. To bypass this process, a new class was written – `HBaseStore` – which implements operations on the default HBase version. This class contains the same methods as the `DefaultStoreSCP` class, although adapted to the new backend. The default class contains one main method – `store()` – which is responsible for a cascade of opperations conducting to a correct storage of the image in the filesystem. Thus, the new adapted class follows the same guideline. Since the actual image still needs to be stored in the local disk, the code of the default method was maintained and some was added in order to perform the HBase-related operations.

In what concerns to the `HBaseStore` the main instructions performed were (see listings on Appendix B.3):

1. Create a new table if it does not exist;
2. Extract the tags' values from the FMI;
3. Create a new `Put` with every value that needs to be inserted on the table;
4. For protected attributes it is necessary to set an `Attribute` in the `Put`;
5. Execute the `Put` over the table.

Every value inserted on the HTable – whether it is a column family, column qualifier, value or attribute – needs to be converted to bytes. In what concerns to dates and times, they

were previously convertd to a `long` format, by getting the date's or time's milliseconds[3], and then converted to bytes. It is important to notice that the extracted times from the FMI are retrieved as a `string` in the format `'HHMMSS'`. Thus, it was necessary to parse the tag content to a workable format. To do so, it was used a parser that converted this string into the `long` format. This can be consulted on Appendix B.4.

Having all these instructions completed, the process of storing the metadata in the HBase is accomplished. It is now important to understand how the whole images are retrieved to the client having only metadata stored there.

When a C-STORE request is performed on an image, before its metadata is sent to the HBase, the image is stored in the local disk in a known location (given by the user through the command line when the StoreSCP is deployed). To enable the identification of the correct image when a query is performed, these images are stored with their SOPInstanceUID as file name. This makes the retrieve process easier since this tag value is the row key in the HTable.

DCMQRSCP

In a analogue way to the StoreSCP, the first change made to the DCMQRSCP utility, was the addition of a command line option (`-f`) to provide the HBase client configuration file, which can be consulted on Appendix B.1.

The *DCMQRSCP* utility contains an important method used to perform the match search for each query – `calculateMatches()`. It receives as parameter an object `Attributes` containing the tag values inserted in the query. Then, from these attributes, independently of the queried ones, are extracted the PatientID, StudyInstanceUID, SeriesInstanceUID, and SOPInstanceUID. These are the only values verified because they are the only ones stored in the DICOMDIR (see DICOMDIR tree – Figure 1). Knowing which were the queried values, the DICOMDIR is scanned by level until the correct match(es) is (are) found. At the end an `InstanceLocator` list is returned. Each one of these objects contains information about one match that must be retrieved to the client. Furthermore, it contains the location of the image, identified by its Uniform Resource Identifier (URI).

Having in mind the default matches calculation process, it is possible to understand that, to adapt the toolkit to the HBase backend, it needs to iterate over an HTable and not over a DICOMDIR. In such way, the `calculateMatches()` method was rewritten, resulting on the `calculateHBaseMatches()`. Thus, the following instructions were written (see listings on Appendix B.5):

1. Read the queried attributes;

3 The date's millisecons corresponds to the milliseconds occurred since the Unix time's beginning, which was on Thursday, 1 January 1970.

2. If the row-key attribute is given, a get operation is performed. If not, a scan with a `SingleColumnValue` filter is performed;

3. An `InstanceLocator` instance is created for each match and it is then added to the list to be returned.

It is important to notice that a `SingleColumnValueFilter` is being used to scan the HTable. Therefore, the developed system is only capable of supporting queries with a single restriction. This means that to each `scan` can only be associated one filter with one restriction. Moreover, this type of filter does not allow the execution of `scan` operations with multiple restrictions. This will implicate that the system can only handle queries, for instance, to retrieve all images of patients with weight=70kg. To avoid this constraint, a `FilterList` could be used [19].

However, since the purpose of this work is to test the basic query cases, the implementation of these filters was not a priority. Remember that the baseline version only supported queries with the four values stored in the DICOMDIR file. Since this prototype's database stores other values – such as names, dates or institutions – these parameters can also be queried. Thus, this is a new feature added to the default ones.

There is also another detail important to refer. The `InstanceLocator` object needs four values to be instantiated: SOP Class UID, SOP Instance UID, Transfer Syntax UID, and the image's URI. However, the SOP Class UID is not stored in the HBase and the Transfer Syntax UID is not in the `get` or `scan` result. This implies the images are read from the filesystem to extract the needed values and, only then, the `InstanceLocator` instance can be created. This is a time consuming operation which can have some impact on the performance results.

Section 4.3.4 explains how the adapted toolkit interacts with the set up cluster and which is the resultant system architecture.

4.3.4 *Architecture and workflow*

Having in mind the defined HBase data schema, the set up cluster, and the changes made to the toolkit, one is now in condition of understanding how these parts interact altogether and how the components are distributed over the cluster.

Besides the HBase cluster's four machines, two more were used: one for the StoreSCU and GetSCU, and one for the StoreSCP and DCMQRSCP. The first two components work as client applications that make store and query/retrieve requests to StoreSCP and DCMQRSCP. Nonetheless, the SCP component are the ones responsible for issuing store and retrieve requests to HBase, making them the HBase clients. Figure 4 represents the system's components and the architecture that they construct together.

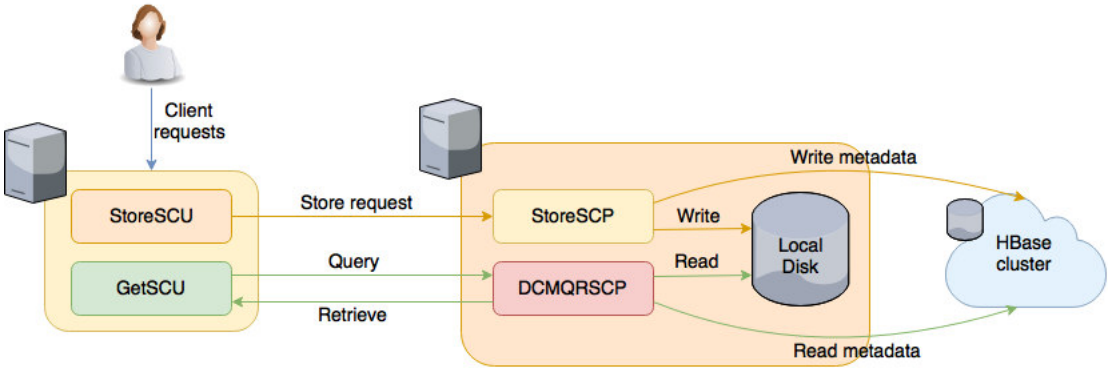


Figure 4.: Dcm4che with HBase backend – architecture and workflow.

Between the components represented on Figure 4, in a store job, the following workflow is established:

1. StoreSCU sends a store request for a set of images to the StoreSCP;
2. StoreSCP processes the images;
3. StoreSCP writes the images on the local disk and sends the attributes' values to the HBase cluster.

In what concerts to query/retrieve jobs, the following workflow is established:

1. GetSCU sends a query to the DCMQRSCP;
2. DCMQRSCP processes the query and communicates it to the HBase cluster;
3. HBase retrieves the SOPInstancesUIDs that matched the query to the DCMQRSCP;
4. DCMQRSCP searches for the corresponding images on the filesystem and retrieves them to the GetSCU.

4.4 DCM4CHE WITH PROTECTED HBASE BACKEND

HBase presents great advantages in what concerns to the storage and retrieval of contents. However it still presents some security issues since it uses an unreliable mean of communication. When working with medical data, the safety measures must be increased since these constitute a sensitive point in what concerns to personal privacy violations. In this way, a symmetric key encryption system was associated with the system described on the previous section – Section 4.3.

Having this in mind, before explaining how the implementation was made, it is important to understand the modifications made to the data schema and which attributes were protected – Section 4.4.1.

In terms of architecture and workflow, there is no change in comparison to the system presented on Section 4.3 – dcm4che with HBase backend. This happens because the cryptographic layer was embedded in the toolkit's implementation. However, a simple overview of the established workflow is made on Section 4.4.3. Section 4.4.2 focuses on explaining the changes made to the toolkit in order to alter its backend to an HBase protected version.

4.4.1 *Protected data*

To develop a dcm4che prototype with a protected HBase as backend, in comparison to the previously described version – dcm4che with a simple HBase backend –, the only change needed, in terms of schema, was the decision of which attributes should be protected and the development of the encryption and decryption system. In such way, the HBase data schema was maintained (see Figures 1 and 2), however, a column attribute was added in order to mark the attributes that need to be protected.

According to the directives described on Section 2.7, all attributes that can lead to an individual's identification must be protected. In such way, in our understanding and according to the list given in Department of Health & Human Services - USA [106], the attributes chosen to be protected were:

- Patient:Name;
- Patient:BirthDate;
- Patient:Weight;
- Patient:History;
- Series:RefPhysician;
- Series:Institution.

The following section – Section 4.4.2 – shows how the toolkit was adapted in order to support a protected version of HBase.

4.4.2 *Toolkit adaptations*

In order to protected the attributes indicated on Section 4.4.1 as susceptible to reveal the individual's identity, the first step was the development of two classes:

- `EncryptionService`: responsible for the data encryption;
- `DecryptionService`: responsible for the data decryption.

To execute both of the above mentioned services, two parameters need to be specified:

- `IvParameterSpec`: initial vector;
- `SecretKeySpec`: secret key.

Both initial vector and secret key are hard-coded and take as input a 16 byte array, which is enough for a performance testing environment. However, in a further development these parameters should be given from an external input, e.g., a file.

Regarding the type of cipher, it was used an `"AES/CBC/PKCS5PADDING"`. The AES [138] is the symmetric encryption standard currently implemented by the National Institute of Standards and Technology (NIST). In this case study a block cipher of 128 bits was used [139]. Cipher Block Chaining (CBC) [140] is an encryption method which states that for each block of plain text it is ciphered with the previously ciphered text block. The `"PKCS5PADDING"` string indicates the padding used in the encryption/decryption service. Public Key Cryptography Standards (PKCS) 5 is a password-based encryption service which has associated a padding of 64 bits.

Three operations are being tested: `put`, `get`, and `scan`. To execute these operations the decryption service is not needed. However, for testing purposes and for a future work, the decryption service class was created. This class was used to verify, before starting the benchmarking tests, if the encryption service was working correctly. These encryption and decryption classes can be consulted on Appendixes B.7.1 and B.7.2, respectively.

Having the encryption service working, it is now time to adapt the basic operations `put` and `scan`. To do so, two new classes were created:

- `SymHBaseStore`: extends the class `HBaseStore` (mentioned on section 4.3) and overrides the `createTableInterface` method, which returns a `SymColTable` instance.
- `SymColTable`: overrides the `put` and `getScanner` methods of the `TableInterface` with new ones, adapted to encrypt the chosen values.

The above mentioned classes can be found on Appendixes B.8.1 and B.8.2, respectively.

In a generic way, the `SymColTable` methods check if the `put` or `scan` being executed have the attribute marking the values as sensitive ones. If this attribute is found, a new `put` or `scan` is created with an encrypted value and, then, the operation is executed over the `TableInterface`. This means the encryption jobs are being executed on the StoreSCP and DCMQRSCP.

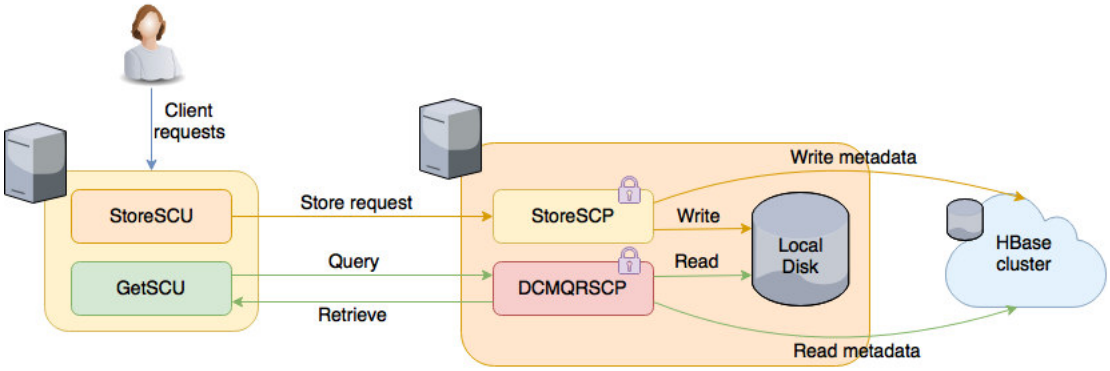


Figure 5.: Dcm4che with protected HBase backend – architecture and workflow.

Knowing how this dcm4che version was implemented, one is now in condition of understanding the architecture and workflow of this system. These topics are approached on the following section.

4.4.3 *Architecture and wokflow*

Knowing how the system is implemented, it is now important to know how its parts interact with each other and which workflow is established.

Since the way data is inserted on the HTable and the way it is searched didn't change (in comparison to the dcm4che with simple HBase backend) the architecture and workflow of the system is maintained – Figure 5. The only difference is the encryption job on the SCPs. In such way, instead of communicating to HBase all metadata in plaintext, the protected tags are communicated in a ciphered way.

5

BENCHMARKING

This chapter focuses on explaining the most important concepts in what concerns to the performance evaluation methods and how their metrics were acquired. Furthermore, here is exposed the used data set and how it was generated. Section 5.1 contextualizes the chapter and describes its overall structure.

5.1 INTRODUCTION

Acquiring performance metrics and comparing them, whether on computer science or any other industry, is commonly called *benchmarking* [141]. The DSR methodology states that after the prototype systems' development it is important to assess their performance. In such way, this chapter focuses on explaining how the developed prototypes were benchmarked. It is important to notice that to execute benchmarking tests, it is necessary to have some kind of application simulating clients and executing controlled requests over the system.

In the first place, it is necessary to supply the benchmark with data to work. In such way, a `python` script was written in order to automatically generate a dataset composed of several medical images. Section 5.2 focuses on explaining how this script works and, thus, in what consists the used dataset.

Thereafter, Section 5.3 explains how a system's performance can be assessed. Here are described the most important performance indicators, as well as the specific metrics acquired during the developed prototypes' testing.

Section 5.4 exposes how the referred metrics were acquired during the tests' execution.

Finally, Section 5.5 focuses on explaining how the scripts used to simulate clients were constructed and what processes are started in each test.

5.2 DATA SET

To run benchmarking tests with medical images it was important to have a solid dataset to supply to clients. In such way, a base image was used, which was then replicated to create

a dataset with a customizable size. The base image used was a slice of a helical chest CT scan, whose content dump can be consulted on Appendix A.2. The image's size is 531KB.

In order to create a diversified dataset and become closer to a real-world system, the image was cloned and the stored tag values were modified to a random value. To achieve that, a `python` script was written. This script made use of the *DCMTK toolkit*'s utility *dcmodify* to change some tags' values. The usage of this toolkit was necessary because dcm4che does not contain a utility capable of performing this operation. Besides, this has a similar usage to dcm4che, meaning it has a command line interface and is, thus, easily employable in such scripts as this one.

Regarding the script's execution, some parameters need to be provided. First of all, it needs a preferences file that contains:

- The base image's path;
- The path to the dcmodify utility binary file;
- The location where the replicas will be stored.

These parameters were specified in a file because they are less likely to change from run to run. More unstable parameters need to be given through the command line. These parameters are:

- The preferences' file path;
- The number of wanted replicas;
- The number of images with patient's weight = 70kg.

As said before, images' metadata is composed of several tags. Although, for this case study, it was only important to diversify the ones being inserted on the HTable. In such way, hereafter is described how each one of the new values were chosen.

In what concerns to the image's UIDs – Study Instance UID, Series Instance UID, and SOP Instance UID – were modified using the options provided by the dcmodify utility – `gst`, `gse`, and `gin`, respectively – to maintain valid values for these tags and, thus, maintain a valid image.

About the patient's weight, it takes random values between 20kg and 120kg. However, it was decided that, in order to have control over the retrieved image's number when performing `scan` operations, it would be useful to decide the number of images with the same weight inserted on the database. In such way, as previously mentioned, the developer needs to specify the number of images with patient weight=70kg that will be created. Hence, in first place, the script generates all required images with weight=70kg and only then starts creating images with a random patient weight.

In what concerns to the remaining tag values, every value is randomly picked from a group of possible values, with the exception of the patient name, referring physician name and institution. These last three values are result of a concatenation of the strings `"PATIENT"`, `"PHYSICIAN"`, and `"INSTITUTION"` with a sequential number. Pieces of the script used to generate the data set can be consulted on Appendix B.6.

Having the dataset created, it is now possible to start executing the performance tests. The following section – Section 5.3 – focuses on explaining how a system's performance can be assessed, which kind of parameters should be analysed and what metrics were acquired during the benchmarking job.

5.3 Performance evaluation

A system, on the user's perspective, is considered a well performing system when the user is not able to notice any kind of delay on the system's response. This performance can be measured [142, 143]. There are some metrics that can be acquired and be used to assess quantitatively a system's performance. These metrics are also known as Key Performance Indicators (KPIs) and are subdivided in two main groups. Hereafter are described each one of these groups and what metrics are considered in each one:

- **Service-oriented** – measure the service's quality provided to the user:
  - **Availability**: amount of time that the system is available;
  - **Response time**: time took to answer to a client's request;
- **Efficiency-oriented** – assesses the system's infrastructure use:
  - **Throughput**: number of operations made per time unit;
  - **Capacity**: percentage of available resources usage.

The work described on this master thesis focuses on two main metrics – latency (response time) and throughput. However, in order to support the obtained results, some capacity indicators were also acquired. Between these, it is possible to find: CPU, disk, memory, and network usage, as well as percentage of performed reads and writes.

Having these concepts in, it is possible to perceive that, to test the prototypes and acquire these metrics, it is necessary to implement ways of acquiring them. In such way, Section 5.4 describes how the system's metrics were acquired.

5.4 Metrics acquisition

The metrics' acquisition is a fundamental step in software performance test. In the present case, both service and efficiency metrics were acquired. About the capacity efficiency met-

rics, they were acquired through the start of a `dstat` process. This process writes to a `csv` file the state of the disk, network, memory, and I/O operations. This process is started in every machine of the cluster through the command presented on Listing 5.1.

```
nohup dstat -t -c -d -m -n -r --output cloud80.csv --noheaders > /dev/null &
```

Listing 5.1: `dstat` process start – example command.

In what concerns to latency and throughput, some timers were added to the source code of the SCU utilities to register the time each operation took to be executed. These timers are posteriorly written to a text file for further processing.

Regarding GetSCU, the timers' results account for the amount of time occurred between the issuing of the retrieve request and the image write in the client's local disk. However, it is important to notice that for each retrieved image, the instant of the request issuing is registered. In the StoreSCU case, the timers measure the time occurred during the image's send job.

Knowing the amount of images retrieved or sent in each test, as well as the tests' duration, it is possible to calculate each job's latency and throughput.

Section 5.5 focuses on explaining how clients were simulated and, thus, how the used scripts are structured.

5.5 Benchmarking scripts

In order to run performance tests, it is necessary to use an application tool simulating clients issuing requests. In such way, `bash` scripts were written.

In a simple way, these scripts execute the following operations:

- Prepare the machine for a new test run:
  - Remove previous results files;
  - Create new results files;
  - Generate the dataset;
  - Start a process to monitor the state of the machine – `dstat`;
  - Start the SCP;
- Start the client(s);
- Stop the `dstat` process;
- Collect the data.

This is the overall workflow of the benchmarking scripts. Although, some small deviations can be identified between `put`, `get` and `scan` scripts due to the system's requirements.

6

RESULTS

This chapter focuses on exposing and exploring the results obtained from the systems' benchmarking. Here are explained the tests performed, their results and some conclusions that can be taken from analysing the results.

6.1 INTRODUCTION

From some tests execution over the developed prototypes, many sets of results were obtained. With those results, some charts were constructed in order to make the results' analysis easier. These charts were constructed using `python` scripts.

This chapter is subdivided into four main sections. The first three focus on results obtained in each developed system. Section 6.2 focuses on tests run over the baseline system, Section 6.3 on tests run on the dcm4che with an unprotected HBase backend, and Section 6.4 focuses on the results obtained from the tests run over the dcm4che with a protected HBase backend.

Finally, Section 6.5 makes a direct comparison between each system's performance results for each tested operation. This has the purpose of making it easier to identify the overhead between systems.

6.2 DEFAULT DCM4CHE

The first system that is important to analyse is the baseline, i.e., the system that already exists on the market and uses the local disk and filesystem as backend. In such way, tests over two functionalities – `C-STORE` and `C-GET` – were performed.

Each one of these tests is explained on a distinct section. Section 6.2.1 focuses on explaining the `C-STORE` tests, the obtained results and what conclusions can be taken of the results' analysis. The same happens with the `C-GET` test, in Section 6.2.2.

Table 3.: Default dcm4che – C-STORE latency and throughput.

Metric	Average	Standard deviation
Latency (ms)	25.157	15.618
Throughput (ops/ms)	0.040	0.0002

Table 4.: Baseline dcm4che – DICOMDIR creation and update latency and throughput.

Metric	Average	Standard deviation
Latency (ms)	0.019	0.001
Throughput (ops/ms)	52.532	1.839

6.2.1 *C-STORE*

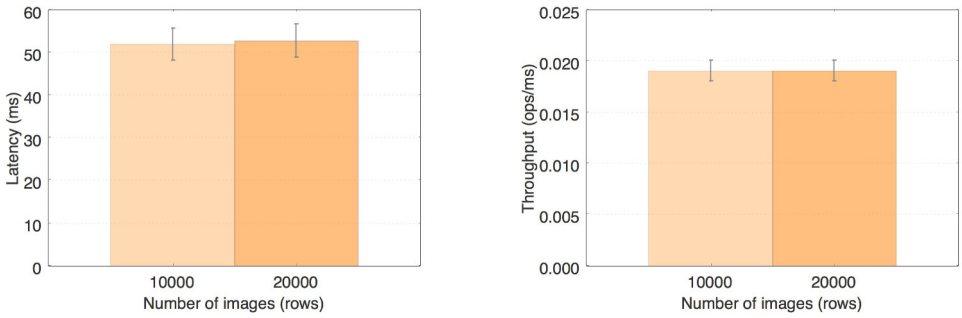
In order to test the baseline system when storing medical images, a test was ran and repeated 9 times, performing a total of 10 runs. Each run of the test stored 20 000 images on the server. The operation's throughput and latency were calculated by averaging the acquired values. These metrics are represented on Table 3.

From the analysis of the results presented on Table 3, it is possible to see that the C-STORE registered latency was 25.157ms, with 15.618ms standard deviation. In terms of throughput, for this operation, was registered a 0.040ops/ms throughput, with 0.0002ops/ms standard deviation.

It is important to notice that having the images stored, the system is still not prepared to answer to queries. To allow the query/retrieve workflow, the creation and update of the DICOMDIR file is needed. Without this step, the GetSCU will not consider the images that are not registered on this file. However, the time required to update the DICOMDIR is not considered in the C-STORE results because it is necessary to use a different dcm4che utility to update it. Table 4 shows the latency and throughput of adding a new image to the database in the worst case scenario, which consists of the addition of four new records: Patient ID, Study Instance UID, Series Instance UID, and SOP Instance UID. At the end of each C-STORE test, the DICOMDIR was updated. Hence, the results presented on Table 4 were calculated on the basis of ten tests, each of them registering 20 000 new images. This table shows that this operation's registered latency was 0.019ms, with 0.001ms standard deviation. Regarding its throughput, the obtained value was 52.535ops/ms, with 1.839ops/ms standard deviation.

6.2.2 *C-GET*

In order to test the query/retrieve service's performance, 10 000 random C-GET requests were executed over two DICOMDIR sizes – 10 000 and 20 000 images. The system was



(a) Latency.

(b) Throughput.

Figure 6.: Default dcm4che – `C-GET` latency and throughput.

queried by every image in the database with a given Study Instance UID. It is important to notice that this system can only be queried by the tags mapped on the DICOMDIR, i.e., Patient ID, Study Instance UID, Series Instance UID, and SOP Instance UID.

From the query/retrieve job acquired times, the first 100 were not considered in order to prevent the inclusion of results influenced by the server's warm up. The obtained latency and throughput values, which are presented on Figure 6, were calculated by averaging the acquired times.

From the analysis of the system's latency – represented on Figure 6a – it is possible to see that in a bigger database the query/retrieve service takes slightly more time to answer a request. However, this is a 0.816ms increase – from 52.758ms to 51.942ms – which is not significant. The standard deviation in the database with 10 000 entries is 3.780ms, while in the database with 20 000 rows it is 3.872ms.

This small increase on the `C-GET` latency might happen because the images are being mapped on a file. Since the number of records in this file is bigger, the time for the system to seek the queried value in the entire database increases.

On the other hand, the throughput – represented on Figure 6b – does not show different values on different sized databases. The registered throughput value on both databases was 0.019ops/ms, with 0.001ops/ms standard deviation. This empowers the idea that there is no significant difference on the system's response time, independently of the number of records registered on the DICOMDIR file.

It is important to notice that to execute the each `C-GET` it is necessary to restart the client's Java Virtual Machine (JVM) – GetSCU – because the dcm4che does not support querying a list of values. In the `C-STORE` case it was opted to use the same client to perform the images' storage because it is a more realistic scenario.

Table 5.: Dcm4che with unprotected HBase backend – `put` throughput and latency.

Metric	Average	Standard deviation
Latency (ms)	25.245	11.105
Throughput (ops/ms)	0.040	0.0003

6.3 DCM4CHE WITH HBASE BACKEND

In order to test the dcm4che version with an unprotected HBase backend, tests over three types of operations were run – `put`, `get`, and `scan`. Each one of these tests is addressed in Section 6.3.1, 6.3.2, and 6.3.3, respectively. These sections expose the tests run, which were the obtained results and which conclusions can be taken from the results' analysis.

6.3.1 *Put*

In order to test the system's behaviour when performing `put` operations, a test was made where 20 000 images/rows were inserted in the HTable, which was empty at the beginning. Here, as in the default dcm4che, was only used one StoreSCP client to store all images.

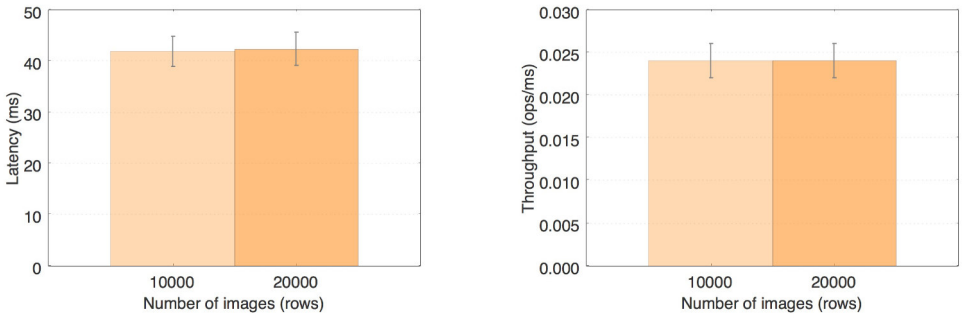
The test was run 10 times, in order to have a more solid set of results, since the initial state of the database is different for each image insertion. Afterwards, latency and throughput values were determined using the average of the values acquired during those `puts`. The registered latency and throughput values can be consulted on Table 5.

From Table 5 results' analysis, it is possible to see that the `put` operation in this dcm4che version the registered latency was 25.245ms with 11.105ms standard deviation. In terms of throughput, was obtained a total of 0.040ops/ms, with 0.0003ops/ms standard deviation.

In this case, since the database used is HBase, the creation and update of the DICOMDIR is completely unnecessary. This happens because the query/retrieve workflow was deviated from this file and redirected to HBase.

6.3.2 *Get*

One of the search operations that can be performed over HBase is the `get`. To perform this operation it is necessary to query the database using the image's SOP Instance UID, which is the HBase's row key. In order to test the performance of this functionality in the dcm4che environment, two tests were made. Each one of these tests performed 10 000 random `gets`, where the first 100 of each were dumped. However, one of the tests was performed over a database populated with 10 000 rows and the other one over a database populated with 20 000 rows.



(a) Latency.

(b) Throughput.

Figure 7.: `Get` results on dcm4che with simple HBase backend.

It is important to notice that the dcm4che does not allow querying for a list of values. Meaning, it is not possible to execute all 10 000 `gets` within a single client. In such way, to perform each `get` it is necessary to start a new GetSCU client.

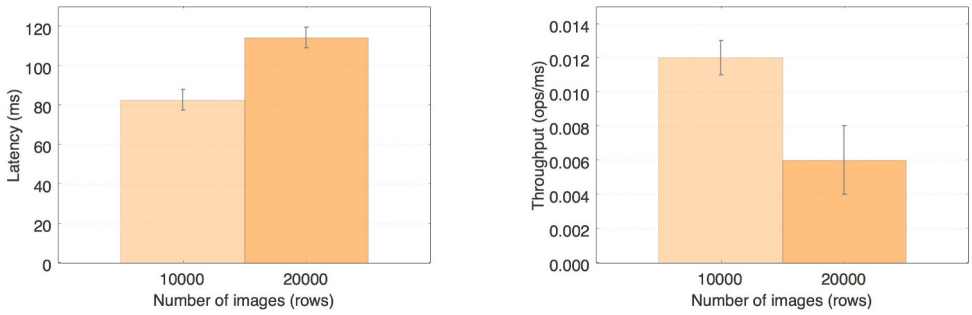
From the analysis of the obtained throughput and latency values, which are presented on Figure 7, it is possible to conclude that the database's size increase has almost no impact on the `get` latency – Figure 7a. From one database size to another, the `get` latency only increases 0.562ms – from 41.840ms to 42.402ms. It is also possible to see that the latency's standard deviation is 2.935ms for the database with 10 000, and 3.233ms for the database with 20 000 rows.

This same conclusion is reinforced by the non-existent difference between the throughput obtained on the 10 000 and 20 000 rows databases – presented on figure 7b – , which value is 0.024ops/ms with 0.002ops/ms standard deviation.

6.3.3 *Scan*

To test the `scan` operation provided by the HBase in the dcm4che environment, two tests of 10 000 `scans` were made, where the first 100 results were removed from each of them. One of these tests was performed over a database populated with 10 000 rows and the other one was made over a database populated with 20 000 rows. In this way, it is possible to study the influence that the database size has in the operation's performance.

To run this test, the database was queried for images of patients weighting 70kg. Notice this is a new feature that was added to the system. In the baseline dcm4che version, the system was only capable of answering to queries by Patient ID, SOP Instance UID, Series Instance UID, and Study Instance UID because these are the only values stored on the DICOMDIR. Despite the acceptance of query parameters like the patient's weight, the baseline system does not even verify their existence on the query. Remember that the number of



(a) Latency.

(b) Throughput.

Figure 8.: `Scan` results on dcm4che with simple HBase backend.

matches was controlled when the dataset was generated and, in this case, it was chosen a number of matches equal to one.

Observing the obtained results – represented on Figure 8 – it is possible to see that the operation's latency suffers a major increase – from 82.502ms to 114.257ms which makes a 31,755ms difference – when the database size is increased. For the database with 10 000 entries, the registered standard deviation was 5.235ms, while in the database 20 000 rows one was 5.487ms.

In the same way, since the GetSCU is restarted to peform a new request, the throughput shows a major decrease when the database size is increased – decreases from 0.009ops/ms (with 0.001ops/ms standard deviation) to 0.012ops/ms (with 0.0004ops/ms standard deviation). This happens because when an operation takes more time to be performed, the system cannot perform the same number of operations per time unit. Notice that this is only true when performing operations in a non-concurrent way, like in this case. These results are expected because the number of rows to be scanned increases and thus, the HBase needs to do a full table `scan` in order to find every match satisfying the query.

6.4 dcm4che with protected HBase backend

The dcm4che version with the ciphered HBase backend was also tested. Here, the types of tests run were the same as in the unprotected version – `put`, `get`, and `scan`. Thus, in this section, the results are also organized in three main subsections, each one addressing a test type. Section 6.4.1 addresses the `put` tests, the results obtained, and some conclusions taken from analysing those results. The same happens with the `get` and `scan` tests on Sections 6.4.2 and 6.4.3, respectively.

Table 6.: Dcm4che with protected HBase backend – `put` latency and throughput.

Metric	Average	Standard deviation
Latency (ms)	28.227	19.349
Throughput (ops/ms)	0.035	0.0005

6.4.1 *Put*

Having tested the performance of the baseline system and the system with a simple HBase backend it is now important to test the behaviour of the system protected with symmetric cipher. To do so, a test was performed by inserting 20 000 images in the database. Afterwards, this test was repeated 9 times, performing a total of 10 runs, in order to have a more consistent set of data.

With the obtained results, two metrics were acquired – throughput and latency. The values obtained for each one of those are indicated on Table 6.

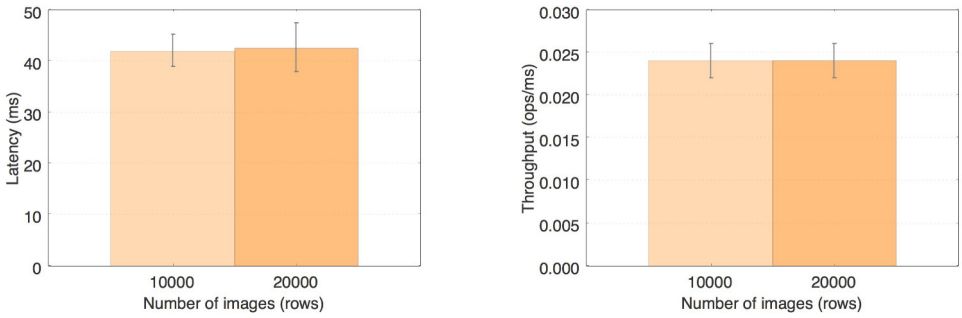
From the analysis of Table 6 it is possible to see that in this `put` operation were registered 28.227ms of total latency, with 19.349ms standard deviation. In what concerns to the throughput, was registered a 0.035ops/ms total value, with 0.0005ops/ms standard deviation.

6.4.2 *Get*

In order to test the `get` performance for this dcm4che prototype, the same tests as in the other versions were executed. Meaning, two tests of 10 000 random `get` operations were performed, where one was made over a database previously populated with 10 000 rows and the other one was run over a database with 20 000 rows. The first 100 results were discarded from each test as ramp up time.

By analysing the obtained throughput and latency results – represented on Figure 9 – it is possible to see that the `get` latency on the 10 000 rows database – Figure 9a – is 41.996ms, which is lower than the value registered on the 20 000 rows database – 42.630ms. However, this is not a significant difference because it only performs a difference of 0.634ms. Analysing the standard deviation, it is possible to see that in the smaller database were registered 3.099ms, while in the bigger one were registered 4.693ms.

Evaluating the throughput results, represented on Figure 9b, it is possible to see that there is no difference between these metrics' values for the 10 000 and 20 000 database sizes. This result strengthens the fact that the latency difference between each tested case is not significant.



(a) Latency.

(b) Throughput.

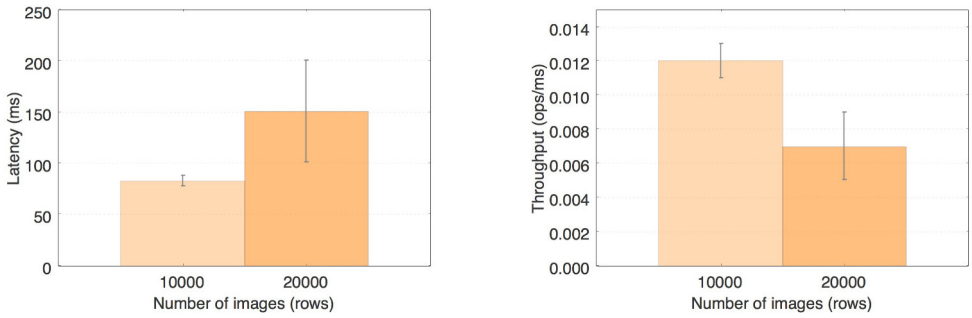
Figure 9.: `Get` results on dcm4che with protected HBase backend.

6.4.3 *Scan*

To test the `scan`'s behaviour in the dcm4che with a protected HBase backend, the same tests were run as in the unprotected version. Meaning, two tests of 10 000 `scans` were made: one on a database with 10 000 rows and another one on a database with 20 000 rows. In a similar way as in the previous tests, the first 100 results for each test were not included on the calculations. To perform this operation, the database was queried for all images were the patient's weight was 70kg. Notice that in the database only one image was stored with this weight, thus, the number of matches was exactly one.

Figure 10 represents the obtained latency and throughput values. From the analysis of the latency – Figure 10a – it is possible to see that when the database size increases, the latency also increases from 82.836ms – with 5.499ms standard deviation – to 150.813ms – with 49.986ms standard deviation–, which performs a total latency difference of 63.977ms. Regarding the standard deviation values, it is possible to see that the tests run on the database with 20 000 rows have a much higher standard deviation. A possible justification for this, is that when the tests over the bigger database were run, the network was much loaded. Again, these latency results are expected because the number of rows to be scanned is bigger.

In the same way, and since the clients are not concurrent, the system's throughput decreases when the size of the database increases. The registered throughput on the 10 000 rows database was 0.012ops/ms – with 0.001ops/ms standard deviation –, while in the 20 000 rows one was 0.007ops/ms – with 0.002ops/ms standard deviation.



(a) Latency.

(b) Throughput.

Figure 10.: `Scan` results on dcm4che with protected HBase backend.

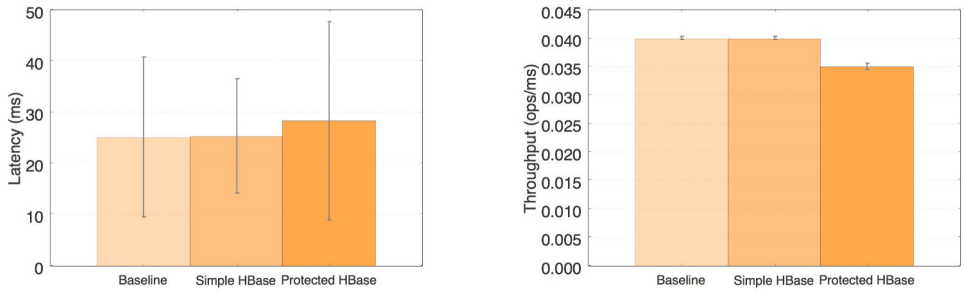
6.5 COMPARISON BETWEEN SYSTEMS

Having in mind how each system performs on its own, now it is important to establish a comparison between each system's performance and, thus, take some conclusions about the trade-offs between the tested systems.

6.5.1 *C-STORE and put*

Figure 11 makes a comparison between each system's store service. Meaning, a comparison is established between the baseline system's `C-STORE` and the `put` of the dcm4che with an unprotected and protected HBase backend. Remember that to test this operation, 20 000 `puts` were made in 10 different tests and the shown results refer to the average of the obtained values

Starting with the latency, shown on Figure 11a, it is possible to see that the system with the best latency is the baseline. Nevertheless, the system with an unprotected HBase backend presents a very similar `put` latency in comparison to the baseline one. This is, the baseline system presents a total latency of 25.157ms with 15.618ms standard deviation, while the dcm4che with an unprotected HBase backend presents 25.245ms with 11.105ms standard deviation. These two values are only 0.088ms apart, which is not a significant difference. It is important to notice that this `C-STORE` results do not include the time required to create and update the DICOMDIR, which is essential to allow the query/retrieve job. This is an operation that adds a 0.019ms latency to the process. The system with a protected HBase backend shows, naturally, an higher latency in comparison to the other two systems – 28.227ms with 19.349ms standard deviation. This is an expected result because the encryption process has an additional overhead in the time it takes to process a query. However, the latency only increases 2.982ms in comparison to the dcm4che with



(a) Latency (b) Throughput

Figure 11.: Comparison of storage performance for each system.

the unprotected HBase backend. Given the total latency, this is not a big difference, which allows the conclusion that the encryption process does not have a significant impact on the operation's latency.

In what concerns to the systems' storage throughput, represented on Figure 11b, it is possible to see that it has the exact opposite behaviour in comparison to the latency. The throughput corresponds to the number of operations a systems is capable to perform per time unit. In such way, it is expected from systems with lower latency to have higher throughput. As is possible to see on Figure 11b, the baseline system is the one with higher throughput – 0.040ops/ms with 0.0002ops/ms standard deviation –, as well as the dcm4che version with an unprotected HBase backend – 0.040ops/ms with 0.0003ops/ms. The system with the protected HBase backend shows a throughput 0.0002ops/ms lower in comparison to the other prototypes – 0.035ops/ms with 0.0005ops/ms standard deviation – because the encryption job includes some latency on the operation. However, as explained before, this is a very small difference, which supports the conclusion that the data encryption does not have a significant impact on the system's performance.

6.5.2 *C-GET and get*

This section establishes a comparison between the default dcm4che's `C-GET` and the two prototypes' `get` operation performance. These two operations are being compared because these are both retrieve mechanisms. Remember that the obtained results are the product of two types of tests: one in a database with 10 000 rows and another one in a database with 20 000 rows. In each test 10 000 random requests were executed, where the first 100 runs were excluded from each test as ramp up time. A comparison between the obtained results is made on Figure 12.

Before starting the results analysis, it is important to notice that when performing `get` operations there is no sensitive data being exchanged. To perform this operation only the row key is needed, which is an unprotected value – the SOP Instance UID.

Having this in mind it is possible to understand that the latency – Figure 12a – and throughput – Figure 12b – values on the protected and unprotected HBase version are equal, as expected, because the operations that are being performed are exactly the same. In a database with 10 000 entries was registered a latency of approximately 42ms with 3ms of standard deviation, while in a database with 20 000 entries was registered a latency of approximately 42.5ms with a standard deviation of 4.693ms in the protected case, and 3.233ms in the unprotected.

Furthermore, it is also possible to see that, in comparison to the prototypes' `get`, the baseline dcm4che's `C-GET` shows higher latency and lower throughput. In a database with 10 000 records, was registered a 51.942ms latency with 3.780ms standard deviation. In the database with 20 000 records was registered a latency of 52.758ms with 3.872ms standard deviation. These values are approximately 10ms greater than the ones obtained using both HBase versions as backend.

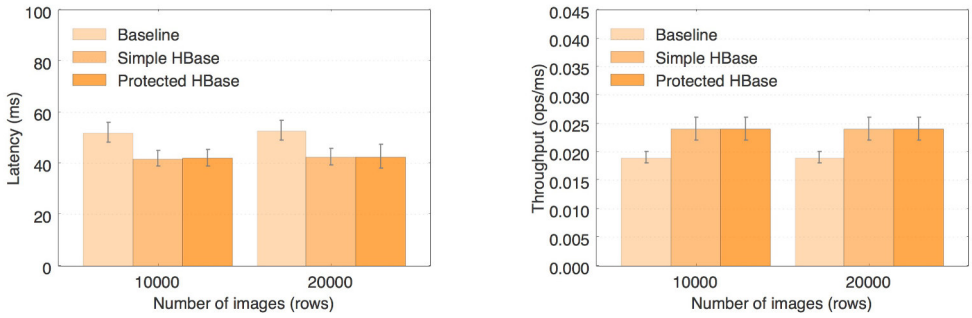
Regarding the throughput, the recorded value was 0.019ops/ms with 0.001ops/ms standard deviation for both database sizes. It is possible to see that this value is 0.005ops/ms lower than the one registered in both prototype systems for the `get` operation.

These results show that the latency of performing a single `get` request is lower than when performing a local search for the image on the DICOMDIR, despite the communication overhead between the DCMQRSCP and the database. This happens because the HTable's entries in each HRegion are sorted by key and the HBase applies bloom filters to check if the queried entry is eventually present on the set or definitely not in it. Besides, since the `get` operation is made by unique keys, when a match is found, the search stops. On the other hand, when executing queries over a DICOMDIR, the file needs to be fully scanned for a match. The application of bloom filters on HBase reduces the search space, making it more efficient than the DICOMDIR's full scan.

6.5.3 *C-GET and scan*

This section focuses on comparing the baseline's `C-GET` with the two prototype systems' `scan` – dcm4che with an unprotected and a protected HBase backend. One more time, it is important to compare the `C-GET`'s performance with the `scan` because, unlike the two developed prototypes, the default dcm4che only presents one way of performing searches.

Before starting the results analysis, it important to recall that the obtained results are the product of two types of tests – one on a 10 000 rows database and another one on a 20 000



(a) Latency (b) Throughput

Figure 12.: Comparison between the performance of baseline dcm4che's `C-GET` and the `get` of the dcm4che with an unprotected and protected HBase backend.

rows – that consisted of 10 000 requests, where the first 100 values were discarded as ramp up time.

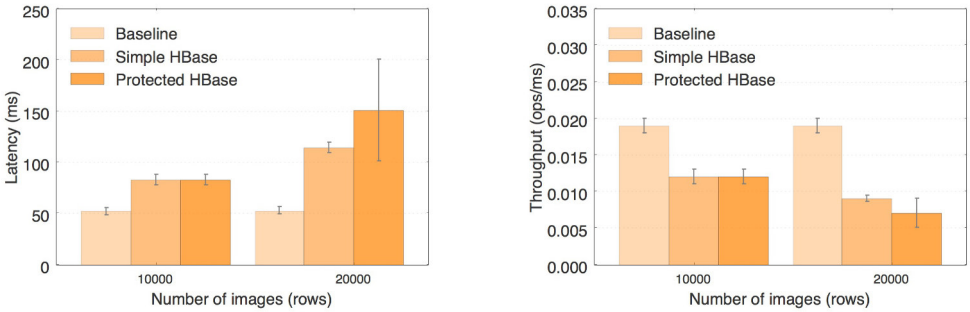
Figure 13 establishes a direct comparison between the `scan`'s performance on the unprotected and protected systems, as well as between those and the baseline dcm4che's `C-GET`.

By analysing this figure, it is possible to see that, in the 10 000 rows database, the difference between the unprotected and protected HBase `scan`'s latency and throughput is almost imperceptible. The unprotected version shows a latency value of 82.502ms with 5.235ms standard deviation, while the protected one shows a total latency of 82.836ms with 5.499ms standard deviation. In such way, it is possible to see that for this database size the latency included by the encryption job is insignificant – 0.334ms. However, when looking to the 20 000 rows database, the difference between each prototype's latency is much bigger. When dealing with protected data in such database, maintaining the HBase backend, the system increases its latency from 114.257ms – with 5.487ms standard deviation – to 150.813ms – with 49.986ms standard deviation –, performing a total difference of 36.286ms. Given the results on the database with 10 000 entries, it is possible to infer that this is not the expected result. In this database, the latency values should be higher than in the smaller one but still similar. This discrepancy may be present due to the high standard deviation that the tests run on the dcm4che with a protected HBase backend show – 49.986ms. This is clearly an uncommon value for a standard deviation, which suggests that something external affected the test's results. For example, an unusually loaded network.

Regarding the `scan`'s throughput on both prototypes, it is possible to see that it maintains a 0.012ops/ms throughput with 0.001ops/ms standard deviation. However, on the database with 20 000 entries it decreases from 0.009ops/ms – with 0.0004ops/ms standard deviation – to 0.007ops/ms – with 0.002ops/ms standard deviation. This happens for the reasons previously described.

Comparing the `scan` and the `C-GET` performances, it is visible that the the `C-GET` has a much lower latency and higher throughput. Having a database with 10 000 rows, the baseline systems shows a 51.942ms latency with 3.780ms standard deviation, while the dcm4che with an unprotected backend presents a latency value of 82.502ms with 5.235ms standard deviation. This means that the baseline systems has a 30.56ms lower latency than the prototype system. On the bigger database, this difference is even more notorious. The `C-GET`'s latency is 52.758ms with 3.872ms standard deviation, which is 61.499ms lower than the unprotected prototype system's. In terms of throughput, the baseline system reached a throughput value of 0.019ops/ms with 0.001ops/ms standard deviation in both databases, which is greater than the one achieved with the dcm4che with an unprotected HBase backend.

Despite this difference, it is important to notice that the `C-GET` does not allow the query of much parameters. Using the default dcm4che it is only possible to execute queries using four parameters: Patient ID, Study Instance UID, Series Instance UID, and SOP Instance UID. On the other side, the prototype systems allow the query by every tag value stored on the HTable. These parameters are specified on Section 4.3.1 and perform a total of 22 possible query values. This allows the user to execute queries using more intuitive parameters, such as names or modalities, becoming more useful to physicians and researchers.



(a) Latency (b) Throughput

Figure 13.: Comparison between the performance of baseline dcm4che's `C-GET` and the `scan` of the dcm4che with an unprotected and a protected HBase backend.

7

CONCLUSIONS AND FUTURE WORK

This dissertation ends with this chapter, which focuses on the main conclusions that can be taken from the development of the described work about the safe storage of medical images in NoSQL databases.

The project was based on NoSQL databases concepts, more specifically HBase, as well as the associated standards with the storage and communication of medical images. Moreover, some knowledge of cryptographic concepts was required, as well as some performance evaluation notions.

Having this said, this chapter is divided in two main sections. Section 7.1 focuses on gathering the main overall conclusions that can be taken form the presented results, as well as this work's main contributions. Section 7.2 makes some suggestions about some future work that can be made over this project.

7.1 FINAL CONSIDERATIONS

This master thesis main goal was the development of a secure and highly available storage solution for medical images, using NoSQL databases.

This project's motivation emerged from the high usage that is being made of SQL databases in the medical industry. These systems present several issues essentially in what concerns to availability and scalability. Nowadays, due to the high rate of data volume creation, the presence of these characteristics in the storage systems used in the medical industry is crucial.

During this project's development, and according to the DSR methodology, some steps were followed. After some research, the lack of a scalable, available, low-cost, and secure imaging system was evident. In such way, a solution for the identified problem emerged: the development of a storage system compliant with the medical imaging standards – the DICOM – capable of storing medical imaging data without violating the individual's data privacy. Having made this decision, the identification of an open-source DICOM toolkit that implemented this standard and allowed the adaptation of its backend became crucial. Along these lines, the chosen toolkit was the dcm4che.

After some study of the dcm4che's structure and functioning, it was altered to use two different new backends: an unprotected and a protected version of HBase. Having this major step accomplished, some performance evaluation tests were performed. These tests evaluated the performance of storage operations – `C-STORE` and `put` –, as well as retrieve operations – `C-GET`, `get`, and `scan`. From these tests the latency and throughput were acquired and analysed.

By analysing the obtained results it is possible to identify some trade-offs between systems:

- **`C-STORE` and `put`**: the storage operations' throughput and latency are equivalent both on the baseline dcm4che and dcm4che with a simple HBase backend. The dcm4che with personal data protection compliance shows an higher latency and lower throughput in comparison to the other two systems due to the time required to encrypt the data. However, this is not a significant difference. Moreover, the two prototype systems update de database by themselves in the storage process, which does not happen in the baseline dcm4che's case.
- **`C-GET` and `get`**: the retrieve process with the `get` operation on the dcm4che with an unprotected or protected HBase backend have a much better performance than the default dcm4che, in terms of latency and throughput. Between the two prototypes, there is no performance difference in these metrics because there is no encryption job being made when this operation is executed.
- **`C-GET` and `scan`**: when performing a retrieve process using a `scan` in the two dcm4che prototypes, the system shows a higher latency and lower throughput in comparison to the default dcm4che. However, the newly developed systems are capable of performing a wider range of queries. Meaning, the baseline system can only be queried for four parameters, while the two new systems can support queries with 22 different tag values. Between the two systems with HBase backends, the encrypted one shows a higher latency and lower throughput due to the encryption step performed before executing the `scan` on the database.

According to the objectives defined on Chapter 1, the availability of medical images was increased by storing their metadata on a distributed server, as proposed. Furthermore, the query capabilities of the used DICOM toolkit were improved, as well as the storage ones by discarding the need to create and update the DICOMDIR separately from the storage job.

7.2 Future work

The prototypes presented in this master thesis, as initially proposed, already increased the availability of the medical images' database. Nevertheless, some improvements can be

made in order to increase it even more. For instance, the actual medical images should be stored in a distributed filesystem, such as HDFS. Having them stored on the HDFS, a link to their location should also be stored on the HBase.

In terms of performance, it would be interesting to study more extensively the behaviour of the HBase's `scan` operation in order to understand how its efficiency can be improved. It would be attractive to reduce the discrepancy between the time required to execute a query using different parameters.

In terms of security, there are also some improvements to be made. The symmetric cipher is a well known technique and well established on the market. However, there are some novel techniques currently emerging on the market, showing great promise in terms of security guarantees. An example is multi-party computation and secret sharing, as described by Pontes et al. [144]. In this model, it is considered that the cloud provider is untrusted and, thus, the data being stored is divided and protected in different parties. This is made in such way that it is not possible to acquire any information about the original, unless every provider cross information with the others. Pontes et al. [144] show that there is a loss in terms of performance when using this type of security protocols, yet, the provided security guarantees are huge.

This master thesis focused on the development of a NoSQL data store for medical images metadata. However, as mentioned on Chapter 2, medical institutions still make use of RDBMS systems. Hence, it would be interesting to these institutions to implement a layer that converts SQL queries to NoSQL operations, enabling the usage of NoSQL databases without changing the existent SQL interfaces.

BIBLIOGRAPHY

[1] Medical Connections. DICOMDIR, July 2008. URL https://www.medicalconnections.co.uk/kb/DICOMDIR. Accessed on: 09-12-2015.

[2] Margaret Rouse and Alex DelVecchio. PACS (picture archiving and communication system) definition, Junho 2015. URL http://searchhealthit.techtarget.com/definition/picture-archiving-and-communication-system-PACS. Accessed on: 04-11-2015.

[3] Luís Bastião Silva, Louis Beroud, Carlos Costa, José Luís Oliveira, et al. Medical imaging archiving: A comparison between several NoSQL solutions. In *Biomedical and Health Informatics (BHI), EEE-EMBS International Conference*, pages 65–68. IEEE, 2014.

[4] Simón J. Rascovsky, Jorge A. Delgado, Alexander Sanz, Víctor D. Calvo, and Gabriel Castrillón. Informatics in radiology: use of CouchDB for document-based storage of DICOM objects. *Radiographics*, 32(3):913–927, 2012.

[5] Neal Leavitt. Will NoSQL databases live up to their promise? *Computer*, 43(2):12–14, 2010.

[6] Luís Bastião Silva, Carlos Costa, Augusto Silva, José Luís Oliveira, et al. A PACS gateway to the cloud. In *Information Systems and Technologies (CISTI), 6th Iberian Conference*, pages 1–6. IEEE, 2011.

[7] P. M. A. van Ooijen, P. J. M. ten Bhomer, and M. Oudkerk. PACS storage requirements — influence of changes in imaging modalities. In *International Congress Series*, volume 1281, pages 888–893. Elsevier, 2005.

[8] Apache HBase. Reference guide, January 2016. URL http://hbase.apache.org/book.html#configuration. Accessed on: 30-01-2016.

[9] Google cloud platform. Cloud BigTable. URL https://cloud.google.com/bigtable/.

[10] CouchDB. Data where you need it. URL http://couchdb.apache.org.

[11] Mongodb. MongoDB. URL https://www.mongodb.com.

[12] OpenStack. OpenStack Swift. URL http://docs.openstack.org/developer/swift/.

[13] Planet Cassandra. What is NoSQL?, 2015. URL http://www.planetcassandra.org/what-is-nosql/. Accessed on: 05-11-2015.

[14] Jing Han, E. Haihong, Guan Le, and Jian Du. Survey on NoSQL database. In *Pervasive computing and applications (ICPCA), 6th international conference*, pages 363–366. IEEE, 2011.

[15] Yishan Li and Sathiamoorthy Manoharan. A performance comparison of SQL and NoSQL databases. In *Communications, Computers and Signal Processing (PACRIM), Pacific Rim Conference*, pages 15–19. IEEE, 2013.

[16] Jaroslav Pokorny. NoSQL databases: a step to database scalability in web environment. *International Journal of Web Information Systems*, 9(1):69–82, 2013.

[17] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[18] Teddyma. Learn cassandra. 2015.

[19] Lars George. *HBase: the definitive guide*. O'Reilly Media, Inc., 2011.

[20] Xuhui Liu, Jizhong Han, Yunqin Zhong, Chengde Han, and Xubin He. Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS. In *Cluster Computing and Workshops, International Conference*, pages 1–8. IEEE, 2009.

[21] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[22] dcm4che. Open source clinical image and object management, . URL http://www.dcm4che.org.

[23] Robert H. Choplin, J. M. Boehme 2nd, and C. D. Maynard. Picture archiving and communication systems: an overview. *Radiographics*, 12(1):127–129, 1992.

[24] Samuel J. Dwyer III. A personalized view of the history of PACS in the USA. In *Medical Imaging*, pages 2–9. International Society for Optics and Photonics, 2000.

[25] NEMA. About DICOM. URL http://dicom.nema.org/Dicom/about-DICOM.html. Accessed on: 10-11-2015.

[26] André J. Duerinckx and E. J. Pisa. Filmless picture archiving and communication in diagnostic radiology. In *Picture Archiving and Communications Systems for Medical Applications*, pages 9–18. International Society for Optics and Photonics, 1982.

[27] Samuel J. Dwyer III, Judith M. S. Prewitt, and André J. Duerinckx. Digital picture archiving and communication systems in medicine. IEEE, 1983.

[28] Robert E. Hoyt, Melanie Sutton, and Ann Yoshihashi. *Medical Informatics: Practical Guide for the Healthcare Professional*. Lulu.com, 2008.

[29] Erwin Bellon, Michel Feron, Tom Deprez, Reinoud Reynders, and Bart Van den Bosch. Trends in PACS architecture. *European journal of radiology*, 78(2):199–204, 2011.

[30] Actualmed. What is PACS server? why do I need one?, October 2010. URL http://www.actualmed.com/blog/en/2010/10/20/servidor-pacs-dicom-server/. Accessed on: 16-11-2015.

[31] Keith J. Dreyer, David S. Hirschorn, James H. Thrall, and Amit Mehta. *PACS: a guide to the digital revolution*. Springer Science & Business Media, 2006.

[32] Nicola H. Strickland. PACS (picture archiving and communication systems): filmless radiology. *Archives of disease in childhood*, 83(1):82–86, 2000.

[33] R. N. J. Graham, R. W. Perriss, and A. F. Scarsbrook. DICOM demystified: a review of digital file formats and their use in radiological practice. *Clinical radiology*, 60(11): 1133–1140, 2005.

[34] Peter Mildenberger, Marco Eichelberg, and Eric Martin. Introduction to the DICOM standard. *European radiology*, 12(4):920–927, 2002.

[35] ISO 12052. Health informatics – digital imaging and communication in medicine (DICOM) including workflow and data management. Standard, International Organization for Standardization, 2006.

[36] Mark O. Gueld, Michael Kohnen, Daniel Keysers, Henning Schubert, Berthold B. Wein, Joerg Bredno, and Thomas M. Lehmann. Quality of DICOM header information for image categorization. In *Medical Imaging*, pages 280–287. International Society for Optics and Photonics, 2002.

[37] B. Ramakrishnan and N. Sriraam. Internet transmission of DICOM images with effective low bandwidth utilization. *Digital Signal Processing*, 16(6):825–831, 2006.

[38] David Power, Eugenia Politou, Mark Slaymaker, Steve Harris, and Andrew Simpson. A relational approach to the capture of DICOM files for grid-enabled medical imaging databases. In *Proceedings of the ACM symposium on Applied Computing*, pages 272–279. ACM, 2004.

[39] NEMA. The data set. URL http://dicom.nema.org/dicom/2013/output/chtml/part05/chapter_7.html.

[40] Bernard Gibaud. The DICOM standard: a brief overview. In *Molecular imaging: computer reconstruction and practice*, pages 229–238. Springer, 2008.

[41] Neologica. The main DICOM services, . URL https://www.neologica.it/html/Tutorial/DICOMServices. Accessed on: 16-11-2015.

[42] NEMA. DICOM information model: Service-Object Pair Class, . URL ftp://dicom.nema.org/medical/DICOM/2013/output/chtml/part04/sect_6.5.html. Accessed on: 30-06-2016.

[43] Neologica. The dicom storage commitment service, . URL https://www.neologica.it/html/Tutorial/DICOMStorageCommitment. Accessed on: 13-11-2015.

[44] Neologica. The DICOM query/retrive service. URL https://www.neologica.it/por/Tutorial/DICOMQueryRetrieve. Accessed on: 16-11-2015.

[45] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[46] Santhosh Kumar Gajendran. A survey on NoSQL databases. Technical report, 2012.

[47] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[48] Edgar F. Codd. Relational database: a practical foundation for productivity. *Communications of the ACM*, 25(2):109–117, 1982.

[49] Clare Churcher. Relational database overview. *Beginning SQL Queries: From Novice to Professional*, pages 1–15, 2008.

[50] Edgar F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems (TODS)*, 4(4):397–434, 1979.

[51] Dan Pritchett. BASE: An ACID alternative. *Queue*, 6(3):48–55, 2008.

[52] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N. Gray, Patricia P. Griffiths, W. Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System R: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[53] Oracle. Defying conventional wisdom. *Profit Magazine*, May 2007.

[54] DB-Engines. DB-Engines Ranking of Relational DBMS, January 2016. URL http://db-engines.com/en/ranking/relational+dbms. Accessed on: 25-01-2016.

[55] Adam Lith and Jakob Mattsson. Investigating storage solutions for large data - a comparison of well performing and scalable data storage solutions for real time extraction and batch insertion of data. 2010.

[56] Carlo Strozzi. NoSQL-a relational database management system. *Lainattu*, 5, 1998.

[57] Eric Evans. Nosql: What's in a name. *Eric Evans's Weblog*, 2009.

[58] Shashank Tiwari. *Professional NoSQL*. John Wiley & Sons, 2011.

[59] Eric Brewer. A certain freedom: thoughts on the cap theorem. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 335–335. ACM, 2010.

[60] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. *Cluster-based scalable network services*, volume 31. ACM, 1997.

[61] Eric Brewer. CAP twelve years later: How the rules have changed. *Computer*, 45(2): 23–29, 2012.

[62] Armando Fox, Eric Brewer, et al. Harvest, yield, and scalable tolerant systems. In *Hot Topics in Operating Systems. Proceedings of the Seventh Workshop*, pages 174–178. IEEE, 1999.

[63] Eric A. Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[64] Michael Stonebraker. Errors in database systems, eventual consistency, and the cap theorem. *Communications of the ACM, Blog ACM*, 2010.

[65] Will Stott, Andy Ryan, Ian J. Jacobs, Usha Menon, Conrad Bessant, and Christopher Jones. Source code for biology and medicine. *Source code for biology and medicine*, 3:11, 2008.

[66] Julie Lerman. Data points - what the heck are document databases?, November 2011. URL `https://msdn.microsoft.com/en-us/magazine/hh547103.aspx`. Accessed on: 30-01-2016.

[67] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

[68] George F. Estabrook and Robert C. Brill. The theory of the TAXIR accessioner. *Mathematical Biosciences*, 5(3):327–340, 1969.

[69] Stephen Weyl, James Fries, Gio Wiederhold, and Frank Germano. A modular self-describing clinical databank system. *Computers and Biomedical Research*, 8(3):279–293, 1975.

[70] MonetDB. A short history about us. URL https://www.monetdb.org/AboutUs. Accessed on: 30-01-2016.

[71] TechTarget. Columnar database, September 2010. URL http://searchdatamanagement.techtarget.com/definition/columnar-database. Accessed on: 30-01-2016.

[72] Apache HBase, November 2016. URL http://hbase.apache.org/.

[73] Thomas Kiencke. Hadoop Distributed File System (HDFS), 2013.

[74] Amitanand S. Aiyer, Mikhail Bautin, Guoqiang Jerry Chen, Pritam Damania, Prakash Khemani, Kannan Muthukkaruppan, Karthik Ranganathan, Nicolas Spiegelberg, Liyin Tang, and Madhuwanti Vaidya. Storage infrastructure behind facebook messages: Using HBase at scale. *IEEE Data Eng. Bull.*, 35(2):4–13, 2012.

[75] Apache ZooKeeper. Zookeeper: A distributed coordination service for distributed applications, August 2014. URL http://zookeeper.apache.org/doc/trunk/zookeeperOver.html. Accessed on: 30-01-2016.

[76] Dhruba Borthakur. HDFS architecture guide. *Hadoop Apache Project*, 2008.

[77] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *Mass Storage Systems and Technologies, 26th Symposium*, pages 1–10. IEEE, 2010.

[78] Pooja S. Honnutagi. The Hadoop distributed file system. *International Journal of Computer Science & Information Technologies*, 5(5):6238–6243, 2014.

[79] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[80] HBase Apache. Data model. URL http://hbase.apache.org/0.94/book/datamodel.html. Accessed on: 31-01-2016.

[81] Amandeep Khurana. Introduction to HBase schema design. *Login*, 37(5):29–36, October 2012.

[82] Seyyed Mojtaba Banaei, Hossein Kardan Moghaddam, et al. Hadoop and its role in modern image processing. *Open Journal of Marine Science*, 4(04):239, 2014.

[83] Yang Jin, Tang Deyu, and Zhou Yi. A distributed storage model for EHR based on HBase. In *Information Management, Innovation Management and Industrial Engineering (ICIII) International Conference*, volume 2, pages 369–372. IEEE, 2011.

[84] Dalia Sobhy, Yasser El-Sonbaty, and Mohamad Abou Elnasr. MedCloud: healthcare cloud computing system. In *Internet Technology And Secured Transactions, International Conference*, pages 161–166. IEEE, 2012.

[85] Qing-An Yao, Hong Zheng, Zhong-Yu Xu, Qiong Wu, Zi-Wei Li, and Lifen Yun. Massive medical images retrieval system based on Hadoop. *Journal of Multimedia*, 9(2): 216–222, 2014.

[86] Said Jai-Andaloussi, Abdeljalil Elabdouli, Abdelmajid Chaffai, Nabil Madrane, and Abderrahim Sekkaki. Medical content based image retrieval by using the hadoop framework. In *Telecommunications (ICT), 20th International Conference*, pages 1–5. IEEE, 2013.

[87] Dimitrios Markonis, Roger Schaer, Ivan Eggel, Henning Müller, and Adrien Depeursinge. Using MapReduce for large-scale medical image analysis. *arXiv*, 2015.

[88] Xiaomao Fan, Chenguang He, Yunpeng Cai, and Ye Li. Hcloud: A novel application-oriented cloud platform for preventive healthcare. In *Cloud Computing Technology and Science (CloudCom), 4th International Conference*, pages 705–710. IEEE, 2012.

[89] Shaozhen Ye, Hanbing Wei, and Yunbin Chen. Design for medical imaging services platform based on cloud computing technologies. In *Cloud Computing and Big Data (CloudCom-Asia), International Conference*, pages 455–460. IEEE, 2013.

[90] Zarina Zafar, Shahid Islam, Muhammad Shehzad Aslam, and Muhammad Sohaib. Cloud computing services for the healthcare industry. *International Journal of Multidisciplinary Sciences and Engineering*, 5(7):25–29, July 2014.

[91] Li Jun Liu and Qing Song Huang. CloudDICOM: A large-scale online storage and sharing system for DICOM images. In *Advanced Materials Research*, volume 756, pages 2037–2041. Trans Tech Publ, 2013.

[92] T. Rajani Devi. Importance of cryptography in network security. In *Communication Systems and Network Technologies, International Conference*, pages 462–467. IEEE, 2013.

[93] Monika Agrawal and Pradeep Mishra. A comparative survey on symmetric key encryption techniques. *International Journal on Computer Science and Engineering*, 4(5):877, 2012.

[94] Hans Delfs and Helmut Knebl. Symmetric-key encryption. In *Introduction to Cryptography*, pages 11–31. Springer, 2007.

[95] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *ACM Symposium on Operating Systems Principles*, 2011.

[96] Suhaila Omer Sharif and S. P. Mansoor. Performance analysis of stream and block cipher algorithms. In *3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, volume 1, pages V1–522. IEEE, 2010.

[97] Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In *International Conference on Information Security*, pages 171–186. Springer, 2006.

[98] Georgios Amanatidis, Alexandra Boldyreva, and Adam O'Neill. Provably-secure schemes for basic query support in outsourced databases. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 14–30. Springer, 2007.

[99] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. In *Annual International Cryptology Conference*, pages 535–552. Springer, 2007.

[100] H. Takabi, J. B. D. Joshi, and G. J. Ahn. Security and privacy challenges in cloud computing environments. *IEEE Security Privacy*, 2010.

[101] World Economic Forum. Personal data: The emergence of a new asset class. Technical report, January 2011.

[102] European Parliament and of the Council of the European Union, 1995.

[103] European Parliament and the Council of the European Union. Directive 2002/58/EC – Directive on privacy and electronic communications. Official Journal of the European Communities, July 2002.

[104] Marian Quigley. *Encyclopedia of information ethics and security*. IGI Global, 2007.

[105] Eropean Commission. Protecting your personal data, June 2016. URL `http://ec.europa.eu/justice/data-protection/individuals/index_en.htm`. Accessed on: 16-06-2016.

[106] Department of Health & Human Services - USA. Protecting personal health information in research: Understanding the HIPAA privacy rule, 2003.

[107] M. Anuja and C. Jeyamala. A survey on security issues and solutions for storage and exchange of medical images in cloud. *International Journal of Emerging Trends in Electrical and Electronics*, 11(6), 2015.

[108] F. Cao, H. K. Huang, and X. Q. Zhou. Medical image security in a hipaa mandated pacs environment. *Computerized Medical Imaging and Graphics*, 27(2):185–196, 2003.

[109] Mariusz Dzwonkowski, Michal Papaj, and Roman Rykaczewski. A new quaternion-based encryption method for DICOM images. *Transactions on Image Processing*, 24(11): 4614–4622, 2015.

[110] Arcangelo Castiglione, Raffaele Pizzolante, Alfredo De Santis, Bruno Carpentieri, Aniello Castiglione, and Francesco Palmieri. Cloud-based adaptive compression and secure management services for 3D healthcare data. *Future Generation Computer Systems*, 43:120–134, 2015.

[111] Tao Xiang, Jia Hu, and Jianglin Sun. Outsourcing chaotic selective image encryption to the cloud with steganography. *Digital Signal Processing*, 43:28–37, 2015.

[112] Luís S. Ribeiro, Carlos Viana-Ferreira, José Luís Oliveira, and Carlos Costa. XDS-I outsourcing proxy: ensuring confidentiality while preserving interoperability. *Journal of biomedical and health informatics*, 18(4):1404–1412, 2014.

[113] Luís A Bastião Silva, Carlos Costa, and José Luís Oliveira. DICOM relay over the cloud. *International journal of computer assisted radiology and surgery*, 8(3):323–333, 2013.

[114] N. J. Manson. Is operations research really research? *ORiON*, 22(2):155–180, 2006.

[115] Martin S. Olivier. *Information Technology Research — A Practical Guide for Computer Science and Informatics*. Van Schaik, 3rd edition, 2009.

[116] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[117] R. Hevner Von Alan, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, 28(1):75–105, 2004.

[118] Vijay Vaishnavi and William Kuechler. Design research in information systems. 2004.

[119] Medfloss. PACS server. URL http://www.medfloss.org/taxonomy/term/84. Accessed on: 24-11-2015.

[120] I do Imaging. List programs, . URL http://www.idoimaging.com/programs. Accessed on: 19-11-2015.

[121] Stanisław Wideł, Andrzej Wideł, and Dominik Spinczyk. Overview of available open source PACS frameworks. *Studia Informatica*, 37, 2016.

[122] IQ-DICOMTEST. URL http://www.image-systems.biz/en/products/free-dicom-tools/iq-dicomtest.html. Accessed on: 02-12-2015.

[123] Orthanc. URL http://www.orthanc-server.com.

[124] OsiriX. URL http://www.osirix-viewer.com.

[125] Frederico Valente, Carlos Costa, and Augusto Silva. Dicoogle, a PACS featuring profiled content based image retrieval. *PloS one*, 8(5):e61888, 2013.

[126] Carlos Costa, Carlos Ferreira, Luís Bastião, Luís Ribeiro, Augusto Silva, and José Luís Oliveira. Dicoogle-an open source peer-to-peer PACS. *Journal of digital imaging*, 24(5): 848–856, 2011.

[127] Carlos Viana-Ferreira, Carlos Costa, and José Luís Oliveira. Dicoogle relay - a cloud communications bridge for medical imaging. In *Computer-Based Medical Systems (CBMS), 25th International Symposium*, pages 1–6. IEEE, 2012.

[128] CDMEDIC PACS. URL http://cdmedicpacsweb.sourceforge.net/CDMEDIC_PACS_WEB.html.

[129] Offis. DCMTK - DICOM toolkit, February 2014. URL http://dicom.offis.de/dcmtk.php.en. Accessed on: 19-11-2015.

[130] ClearCanvas. URL https://www.clearcanvas.ca.

[131] I do Imaging. ConQuest, . URL http://www.idoimaging.com/program/183. Accessed on: 09-12-2015.

[132] OSPACS. URL https://ospacs.codeplex.com.

[133] Xebra. URL http://sourceforge.net/projects/xebra/. Accessed on: 08-12-2015.

[134] dcm4che. dcm4che-3.x DICOM toolkit, . URL https://github.com/dcm4che/dcm4che.

[135] NEMA. Transfer syntax, . URL http://dicom.nema.org/dicom/2013/output/chtml/part05/chapter_10.html. Accessed on: 16-12-2015.

[136] NEMA. A transfer syntax specifications (normative), . URL http://dicom.nema.org/dicom/2013/output/chtml/part05/chapter_A.html. Accessed on: 16-12-2015.

[137] Apache HBase. Apache HBase reference guide, October 2016. URL https://hbase.apache.org/book.html.

[138] Douglas Selent. Advanced encryption standard. *Rivier Academic Journal*, 6(2):1–14, 2010.

[139] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. Advanced encryption standard. 2009.

[140] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362–399, 2000.

[141] Tim Stapenhurst. *The benchmarking book: a how-to-guide to best practice for managers and practitioners*. Elsevier, 1st edition, 2009.

[142] Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 2009.

[143] John Klein, Ian Gorton, Neil Ernst, Patrick Donohoe, Kim Pham, and Chrisjan Matser. Performance evaluation of NoSQL databases: a case study. In *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems*, pages 5–10. ACM, 2015.

[144] Rogério Pontes, Francisco Maia, João Paulo, and Ricardo Vilaça. SafeRegions: Performance evaluation of multi-party protocols on HBase. In *35th Symposium on Reliable Distributed Systems Workshops (SRDSW)*, pages 31–36. IEEE, 2016.

A

DICOM SUPPORT MATERIAL

A.1 DICOM DICTIONARY

```
# Tag    VR  Name       VM  Version
#
(0000,0000) UL  CommandGroupLength  1 DICOM_2009
(0000,0002) UI  AffectedSOPClassUID 1 DICOM_2009
(...)
(0002,0000) UL  FileMetaInformationGroupLength  1 DICOM_2009
(0002,0001) OB  FileMetaInformationVersion  1 DICOM_2009
(0002,0002) UI  MediaStorageSOPClassUID 1 DICOM_2009
(0002,0003) UI  MediaStorageSOPInstanceUID  1 DICOM_2009
(0002,0010) UI  TransferSyntaxUID 1 DICOM_2009
(...)
(0004,1130) CS  FileSetID 1 DICOM_2009
(0004,1141) CS  FileSetDescriptorFileID 1-8 DICOM_2009
(...)
(0004,1500) CS  ReferencedFileID  1-8 DICOM_2009
(...)
(0008,0008) CS  ImageType 2-n DICOM_2009
(0008,0012) DA  InstanceCreationDate  1 DICOM_2009
(0008,0013) TM  InstanceCreationTime  1 DICOM_2009
(0008,0014) UI  InstanceCreatorUID  1 DICOM_2009
(0008,0016) UI  SOPClassUID 1 DICOM_2009
(0008,0018) UI  SOPInstanceUID  1 DICOM_2009
(0008,001A) UI  RelatedGeneralSOPClassUID 1-n DICOM_2009
(0008,001B) UI  OriginalSpecializedSOPClassUID  1 DICOM_2009
(0008,0020) DA  StudyDate 1 DICOM_2009
(0008,0021) DA  SeriesDate  1 DICOM_2009
(0008,0022) DA  AcquisitionDate 1 DICOM_2009
(0008,0023) DA  ContentDate 1 DICOM_2009
(0008,002A) DT  AcquisitionDateTime 1 DICOM_2009
(0008,0030) TM  StudyTime 1 DICOM_2009
(0008,0031) TM  SeriesTime  1 DICOM_2009
(0008,0032) TM  AcquisitionTime 1 DICOM_2009
(0008,0033) TM  ContentTime 1 DICOM_2009
```

```
(0008,0050) SH  AccessionNumber 1 DICOM_2009
(...)
(0008,0060) CS  Modality  1 DICOM_2009
(...)
(0008,0070) LO  Manufacturer  1 DICOM_2009
(0008,0080) LO  InstitutionName 1 DICOM_2009
(0008,0081) ST  InstitutionAddress  1 DICOM_2009
(0008,0082) SQ  InstitutionCodeSequence 1 DICOM_2009
(0008,0090) PN  ReferringPhysicianName  1 DICOM_2009
(...)
(0008,1030) LO  StudyDescription  1 DICOM_2009
(0008,1032) SQ  ProcedureCodeSequence 1 DICOM_2009
(0008,103E) LO  SeriesDescription 1 DICOM_2009
(0008,103F) SQ  SeriesDescriptionCodeSequence 1 DICOM_2009
(0008,1040) LO  InstitutionalDepartmentName 1 DICOM_2009
(...)
(0010,0010) PN  PatientName 1 DICOM_2009
(0010,0020) LO  PatientID 1 DICOM_2009
(...)
(0010,0030) DA  PatientBirthDate  1 DICOM_2009
(0010,0032) TM  PatientBirthTime  1 DICOM_2009
(0010,0040) CS  PatientSex  1 DICOM_2009
(0010,0050) SQ  PatientInsurancePlanCodeSequence  1 DICOM_2009
(...)
(0010,1010) AS  PatientAge  1 DICOM_2009
(0010,1020) DS  PatientSize 1 DICOM_2009
(0010,1030) DS  PatientWeight 1 DICOM_2009
(...)
(0010,21B0) LT  AdditionalPatientHistory  1 DICOM_2009
(...)
(0018,0010) LO  ContrastBolusAgent  1 DICOM_2009
(0018,0012) SQ  ContrastBolusAgentSequence  1 DICOM_2009
(0018,0014) SQ  ContrastBolusAdministrationRouteSequence  1 DICOM_2009
(0018,0015) CS  BodyPartExamined  1 DICOM_2009
(...)
(0020,000D) UI  StudyInstanceUID  1 DICOM_2009
(0020,000E) UI  SeriesInstanceUID 1 DICOM_2009
(0020,0010) SH  StudyID 1 DICOM_2009
(0020,0011) IS  SeriesNumber  1 DICOM_2009
(0020,0012) IS  AcquisitionNumber 1 DICOM_2009
(...)
```

Listing A.1: Used DICOM dictionary.

A.2 DICOM IMAGE DUMP

```
0002,0002  Media Storage SOP Class UID: 1.2.840.10008.5.1.4.1.1.2
0002,0003  Media Storage SOP Inst UID:
   1.2.276.0.7230010.3.1.4.0.2001.1461079064.595125
0002,0010  Transfer Syntax UID: 1.2.840.10008.1.2
0002,0012  Implementation Class UID: 1.2.40.0.13.1.1
0002,0013  Implementation Version Name: dcm4che-3.3.7
0002,0016  Source Application Entity Title: DCMQRSCP
0008,0005  Specific Character Set: ISO_IR 100
0008,0008  Image Type: ORIGINAL\PRIMARY\AXIAL
0008,0016  SOP Class UID: 1.2.840.10008.5.1.4.1.1.2
0008,0018  SOP Instance UID: 1.2.276.0.7230010.3.1.4.0.2001.1461079064.595125
0008,0020  Study Date: 20010105
0008,0021  Series Date: 20010105
0008,0022  Acquisition Date: 20010105
0008,0023  Image Date: 20010105
0008,0030  Study Time: 083501
0008,0031  Series Time: 083709
0008,0032  Acquisition Time: 083848
0008,0033  Image Time: 083852
0008,0050  Accession Number: 0000000001
0008,0060  Modality: CT
0008,0070  Manufacturer: GE MEDICAL SYSTEMS
0008,0080  Institution Name: Toronto Hosp, West Div.
0008,0090  Referring Physician's Name: PHYSICIAN
0008,1010  Station Name: TWD1_OC0
0008,1030  Study Description: CHEST
0008,103E  Series Description: HELICAL CHEST
0008,1060  Name of Physician(s) Reading Study: RADIOLOGIST
0008,1070  Operator's Name: BK
0008,1090  Manufacturer's Model Name: HiSpeed CT/i
(...)
0010,0010  Patient's Name: PATIENT1
0010,0020  Patient ID: 0000001
0010,0030  Patient's Birth Date: 19700101
0010,0040  Patient's Sex: M
0010,1010  Patient's Age: 089Y
0010,1030  Patient's Weight: 0.000000
0010,21B0  Additional Patient History: WIGHT LOSS ,COUGH
(...)
0018,0022  Scan Options: HELICAL MODE
0018,0050  Slice Thickness: 5.000000
0018,0060  kVp:  120
0018,0088  Spacing Between Slices: 6.500000
0018,0090  Data Collection Diameter: 480.000000
0018,1020  Software Versions(s): 05
```

```
0018,1100  Reconstruction Diameter: 346.000000
0018,1110  Distance Source to Detector:  1099.3100585938
0018,1111  Distance Source to Patient: 630.000000
0018,1120  Gantry/Detector Tilt: 0.000000
0018,1130  Table Height: 167.100006
0018,1140  Rotation Direction: CW
0018,1150  Exposure Time:  800
0018,1151  X-ray Tube Current:  200
0018,1152  Exposure:  200
0018,1160  Filter Type: BODY FILTER
0018,1190  Focal Spot(s): 0.700000
0018,1210  Convolution Kernel: STANDARD
0018,5100  Patient Position: FFS
(...)
0020,000D  Study Instance UID:
   1.2.840.113619.2.30.1.1762295590.1623.978668949.886
0020,000E  Series Instance UID:
   1.2.840.113619.2.30.1.1762295590.1623.978668949.890
0020,0010  Study ID: 40933
0020,0011  Series Number:  2
0020,0012  Acquisition Number:  1
0020,0013  Image Number:  1
0020,0032  Image Position (Patient):  -161.399994\ -148.800003\4.700000
0020,0037  Image Orientation (Patient):
   1.000000\0.000000\0.000000\0.000000\1.000000\0.000000
0020,0052  Frame of Reference UID:
   1.2.840.113619.2.30.1.1762295590.1623.978668949.886.8493.0.12
(...)
0028,0004  Photometric Interpretation: MONOCHROME2
0028,0010  Rows: 512
0028,0011  Columns: 512
0028,0030  Pixel Spacing:  0.675781\0.675781
0028,0100  Bits Allocated: 16
0028,0101  Bits Stored: 16
0028,0102  High Bit: 15
0028,0103  Pixel Representation: 1
0028,0120  Pixel Padding Value: 32768
0028,1050  Window Center: 40
0028,1051  Window Width:  400
0028,1052  Rescale Intercept:  -1024
0028,1053  Rescale Slope:  1
(...)
```

Listing A.2: Used image's dump.

B

LISTINGS

B.1 HBASE CLIENT CONFIGURATION FILE

```
<configuration>
  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>cloud80</value>
  </property>
  <property>
    <name>hbase.rootdir</name>
    <value>hdfs://cloud80:8020/hbase</value>
  </property>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/usr/local/hbase-0.98.20-hadoop2/zookeeper-dir</value>
  </property>
  <property>
    <name>hbase.master.hostname</name>
    <value>cloud80</value>
  </property>
  <property>
    <name>base.zookeeper.property.clientPort</name>
    <value>2181</value>
  </property>
</configuration>
```

Listing B.1: HBase client configuration file.

B.2 CLUSTER CONFIGURATION FILES

B.2.1 *Master server*

HBase

```
<configuration>
  <property>
      <name>hbase.rootdir</name> <!--propriedade para definir qual o
          filesystem-->
      <value>hdfs://cloud80:8020/hbase</value>
    </property>
    <property>
      <name>hbase.cluster.distributed</name>
      <value>true</value>
    </property>
    <property>
      <name>hbase.zookeeper.property.dataDir</name>
      <value>/usr/local/hbase-0.98.20-hadoop2/zookeeper-dir</value>
    </property>
    <property>
      <name>hbase.zookeeper.quorum</name>
      <value>localhost</value>
    </property>
    <property>
      <name>hbase.zookeeper.property.clientPort</name>
      <value>2181</value>
    </property>
    <property>
      <name>hbase.master.hostname</name>
      <value>cloud80</value>
    </property>
</configuration>
```

Listing B.2: *hbase-site.xml* file.

```
cloud81
cloud82
cloud83
```

Listing B.3: *regionservers* file.

HDFS

```
<configuration>
  <property>
      <name>fs.default.name</name>
      <value>hdfs://cloud80:8020</value>
    </property>
</configuration>
```

Listing B.4: *core-site.xml* file.

```
<configuration>
  <property>
      <name>dfs.replication</name>
      <value>4</value>
  </property>
  <property>
      <name>dfs.name.dir</name>
      <value>/usr/local/hadoop-2.7.2/dfs/name</value>
  </property>
  <property>
      <name>dfs.data.dir</name>
      <value>/usr/local/hadoop-2.7.2/dfs/data</value>
  </property>
</configuration>
```

Listing B.5: *hdfs-site.xml* file.

```
cloud81
cloud82
cloud83
```

Listing B.6: *slaves* file.

B.2.2 *Region servers*

HBase

```
<configuration>
    <property>
      <name>hbase.rootdir</name>
      <value>hdfs://cloud80:8020/hbase</value>
    </property>
    <property>
      <name>hbase.cluster.distributed</name>
      <value>true</value>
```

```
    </property>
    <property>
        <name>hbase.zookeeper.quorum</name>
        <value>cloud80</value>
    </property>
</configuration>
```

Listing B.7: *hbase-site.xml* file.

```
localhost
```

Listing B.8: *regionservers* file.

HDFS

```
<configuration>
    <property>
      <name>fs.default.name</name>
      <value>hdfs://cloud80:8020</value>
    </property>
</configuration>
```

Listing B.9: *core-site.xml* file.

```
<configuration>
    <property>
      <name>dfs.replication</name>
      <value>4</value>
    </property>

    <property>
      <name>dfs.name.dir</name>
      <value>/usr/local/hadoop-2.7.2/dfs/name</value>
    </property>

    <property>
      <name>dfs.data.dir</name>
      <value>/usr/local/hadoop-2.7.2/dfs/data</value>
    </property>
</configuration>
```

Listing B.10: *hdfs-site.xml* file.

```
localhost
```

Listing B.11: *slaves* file.

B.3 STORING DATA ON HBASE

1. Create a new table if it does not exist:

```
String tableName = "DicomTable";

TableName tbname = TableName.valueOf(tableName);
HTableDescriptor table = new HTableDescriptor(tbname);

HColumnDescriptor patientFamily = new HColumnDescriptor("Patient");
HColumnDescriptor imageFamily = new HColumnDescriptor("Image");
HColumnDescriptor seriesFamily = new HColumnDescriptor("Series");
HColumnDescriptor studyFamily = new HColumnDescriptor("Study");

table.addFamily(patientFamily);
table.addFamily(imageFamily);
table.addFamily(seriesFamily);
table.addFamily(studyFamily);

tableInterface = createTableInterface(conf, tableName);
```

Listing B.12: Create table code.

2. Extract the tags' values from the FMI:

```
if (fmi.contains(Tag.PatientID)) {
  patientID = fmi.getString(Tag.PatientID);
}
```

Listing B.13: Tags' values extraction – example code.

3. Create a new `Put` with every value that needs to be inserted on the table:

```
String key = SOPInstanceUID;

byte[] patientCf = "Patient".getBytes();
byte[] imageCf = "Image".getBytes();
byte[] studyCf = "Study".getBytes();
byte[] seriesCf = "Series".getBytes();

Put put = new Put(key.getBytes());
```

```
if (!patientID.equals(null)) {
  put.add(patientCf, "ID".getBytes(), patientID.getBytes());
}
```

Listing B.14: Put construction – example code

4. For protected attributes it is necessary to set an `Attribute` in the `Put`:

```
put.setAttribute("protected:" + "Patient" + ":Name", "".getBytes());
```

Listing B.15: Addition of a column attribute to a protected one – example code.

5. Execute the `Put` over the table:

```
tableInterface.put(put);
```

Listing B.16: `Put` execution

B.4 TIME PARSER

```
private long ParseTime(String timeTag) {
    int len = timeTag.length();
    Long mills;
    if (len == 4) {
      timeTag = new StringBuffer(timeTag).insert(timeTag.length() - 2, ":").
          toString();
      timeTag = new StringBuffer(timeTag).insert(timeTag.length(), ":00").
          toString();
    } else if (len == 6) {
      timeTag = new StringBuffer(timeTag).insert(timeTag.length() - 2, ":").
          toString();
      timeTag = new StringBuffer(timeTag).insert(timeTag.length() - 5, ":").
          toString();
    } else if (len > 6) {
      timeTag = new StringBuffer(timeTag).insert(3, ":").toString();
      timeTag = new StringBuffer(timeTag).insert(5, ":00").toString();
    } // if none of these conditions are verified, it means that the time
      // format is invalid
    else if (len < 4 || len ==5){
      Date date = new Date();
      return date.getTime();
    }
    SimpleDateFormat formatter = new SimpleDateFormat("hh:mm:ss");
```

```
    Date date = new Date();
    try {
      date = (Date) formatter.parse(timeTag);
    } catch (ParseException e) {
      e.printStackTrace();
    }
    mills = date.getTime();
    return mills;
  }
```

Listing B.17: Parser used do convert times from the DICOM file's format to `long`.

B.5 RETRIEVING DATA FROM HBASE

1. Read the queried attributes:

```
    if (keys.contains(Tag.PatientID)){
        patientID = keys.getString(Tag.PatientID);
      }
      if (keys.contains(Tag.PatientName)){
        patientName = keys.getString(Tag.PatientName);
      }
```

Listing B.18: Qeried attributes read – example code.

2. If the row-key attribute is given, a get operation is performed. If not, a scan with a `SingleColumnValue` filter is performed:

```
    if (SOPInstanceUID != null){
        Get get = new Get (SOPInstanceUID.getBytes());
          get.setAttribute("protected: "+"Patient" + ":BirthDate", "".
              getBytes());
          get.setAttribute("protected: " + "Patient" + ":Name", "".
              getBytes());

        Result res = tableInterface.get(get);
        String value = new String(res.getRow());
      // (...)
    }else{
        Scan scan = new Scan();
        if (patientBirthDate != null){
          filter = new SingleColumnValueFilter("Patient".getBytes(), "
              BirthDate".getBytes(), CompareFilter.CompareOp.EQUAL, Longs
              .toByteArray(patientBirthDate));
          scan.setAttribute("protected: "+"Patient" + ":BirthDate", "".
              getBytes());
```

```
        }

        // (...)
      }

      scan.setFilter(filter);
      ResultScanner scanner = tableInterface.getScanner(scan);
      for (Result res = scanner.next(); res != null; res = scanner.next()
          ){
        String value = new String(res.getRow());

        // (...)
      }
```

Listing B.19: get and scan execution – example code.

3. An InstanceLocator instance is created for each match and it is then added to the list to be returned:

```
    DicomInputStream dis = new DicomInputStream (new File(imagesFolder +
        "/" + value));
      DatasetWithFMI dataWithFMI = dis.readDatasetWithFMI();
      String uri = dis.getURI();
      Attributes dataset = dataWithFMI.getDataset();
      String cuid = dataset.getString(Tag.SOPClassUID);
      String tsuid = dataset.getString(Tag.TransferSyntaxUID);
        InstanceLocator resInstance = new InstanceLocator(cuid, value,
            tsuid, uri);
        list.add((T) resInstance);
```

Listing B.20: Creation of the InstanceLocator list – example code.

B.6 DATA SET'S CREATION SCRIPT

```
def execute_command(command):
    bash_command = command.split(" ")
    subprocess.call(bash_command)

def generate_dataset (num_replicas, num_fixed_weight, image_path,
    replicas_folder, dcmodify):

    for i in os.listdir(path=replicas_folder):
        os.remove(replicas_folder + i)
    for i in range(num_replicas):
```

```
  shutil.copy2(image_path, replicas_folder + str(i+1) + '.dcm')
PatientName_tag = "(0010,0010)"
PatientAge_tag = "(0010,1010)"
PatientWeight_tag = "(0010,1030)"
# (...)
StudyDescription_tag = "(0008,1030)"
Modality_tag = "(0008,0060)"
# (...)
SeriesDate_tag = "(0008,0021)"
SeriesHour_tag = "(0008,0031)"
SeriesDescription_tag = "(0008,103E)"
ReferingPysician_tag = "(0008,0090)"

SOPInstanceUID_m = dcmodify + " -gin "
StudyInstanceUID_m = dcmodify + " -gst "
SeriesInstanceUID_m = dcmodify + " -gse "
PatientID_m = dcmodify + " -m " + PatientID_tag + "="
PatientName_m = dcmodify + " -m " + PatientName_tag + "="
PatientAge_m = dcmodify + " -m " + PatientAge_tag + "="
PatientWeight_m = dcmodify + " -m " + PatientWeight_tag + "="
# (...)
Modality_m = dcmodify + " -m " + Modality_tag + "="
# (...)
SeriesDate_m = dcmodify + " -m " + SeriesDate_tag + "="
SeriesHour_m = dcmodify + " -m " + SeriesHour_tag + "="
SeriesDescription_m = dcmodify + " -m " + SeriesDescription_tag + "="

gender = ["M", "F"]
description = ["CHEST", "HELICALCHEST", "PELVIS", "HEAD"]
modality = ["CT", "CR", "PET", "MRI"]
# (...)

counter = 1
for j in range(num_replicas):
    i = str(j+1) + ".dcm"
    print (i)
    execute_command(SeriesInstanceUID_m + replicas_folder + i)
    execute_command(StudyInstanceUID_m + replicas_folder + i)
    execute_command(SOPInstanceUID_m + replicas_folder + i)
  # (...)
    execute_command(PatientName_m + "PATIENT" + str(counter) + " " +
        replicas_folder + i)
    execute_command(PatientAge_m + str(random.randint(10, 80)) + " " +
        replicas_folder + i)
    # (...)

    if counter < (num_fixed_weight+1) :
```

```
            execute_command(PatientWeight_m + str(70) + " " + replicas_folder
                + i)
        else :
            value = random.randint(20,120)
            if value == 70:
                value = value + 1
            execute_command(PatientWeight_m + str(value) + " " +
                replicas_folder + i)

        execute_command(Modality_m + random.choice(modality) + " " +
            replicas_folder + i)
        # (...)
        execute_command(SeriesDate_m + str(random.randint(1998, 2016)) + str(
            random.randint(1, 12)).zfill(2) +
                        str(random.randint(1, 28)).zfill(2) + " " +
                            replicas_folder + i)
        execute_command(SeriesHour_m + str(random.randint(8, 23)).zfill(2) +
            str(random.randint(0, 59)).zfill(2) +
                        str(random.randint(0, 59)).zfill(2) + " " +
                            replicas_folder + i)
        execute_command(SeriesDescription_m + random.choice(description) + " "
             + replicas_folder + i)
        # (...)
        counter += 1

    for i in os.listdir(path=replicas_folder):
        if i.endswith(".bak"):
            os.remove(replicas_folder+i)

if __name__ == '__main__':
    cl_parser = argparse.ArgumentParser(description='Give preferences.xml')
    cl_parser.add_argument('file', help='preferences file', type=str)
    cl_parser.add_argument('num_fixed_weight', help='number of images with
        weight=70kg', type=int)
    cl_parser.add_argument('num_images', help='number of copies', type=int)
    args = cl_parser.parse_args()
    preferences_file = args.file
    num_fixed_weight = args.num_fixed_weight
    num_replicas = args.num_images
    tree = ET.parse(preferences_file)
    base_image = tree.find("baseImage").text
    replicas_folder = tree.find("replicasFolder").text
    dcmodify = tree.find("dcmodify").text
    generate_dataset(num_replicas, num_fixed_weight, base_image,
        replicas_folder, dcmodify)
```

Listing B.21: Data set's creation script – example code.

B.7 CRYPTOGRAPHIC SERVICES

B.7.1 *Encryption service class*

```
public class EncryptionService {

  private byte[] key;
  private byte[] initVector;

  public byte[] getKey() {
    return key;
  }
  public void setKey(byte[] key) {
    this.key = key;
  }
  public byte[] getInitVector() {
    return initVector;
  }
  public void setInitVector(byte[] initVector) {
    this.initVector = initVector;
  }

  public EncryptionService(){
    super();
  }

  public EncryptionService(byte[] key, byte[] initVector){
    this.key = key;
    this.initVector = initVector;
  }

  public byte[] cipher(byte[] value) throws UnsupportedEncodingException,
      NoSuchAlgorithmException,
    NoSuchPaddingException, InvalidKeyException,
        InvalidAlgorithmParameterException,
    IllegalBlockSizeException, BadPaddingException{

    IvParameterSpec iv = new IvParameterSpec(initVector);
        SecretKeySpec skeySpec = new SecretKeySpec(key, "AES");
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");
        cipher.init(Cipher.ENCRYPT_MODE, skeySpec, iv);
        byte[] encrypted = cipher.doFinal(value);
        return encrypted;
  }
}
```

Listing B.22: EncryptionService class.

B.7.2 *Decryption service class*

```
public class DecryptionService {

  private byte[] key;
  private byte[] initVector;

  public byte[] getKey() {
    return key;
  }
  public void setKey(byte[] key) {
    this.key = key;
  }
  public byte[] getInitVector() {
    return initVector;
  }
  public void setInitVector(byte[] initVector) {
    this.initVector = initVector;
  }

  public DecryptionService(){
    super();
  }

  public DecryptionService(byte[] key, byte[] initVector){
    this.initVector = initVector;
    this.key = key;
  }

  public byte[] decipher(byte[] value) throws UnsupportedEncodingException,
      NoSuchAlgorithmException,
    NoSuchPaddingException, InvalidKeyException,
        InvalidAlgorithmParameterException,
    IllegalBlockSizeException, BadPaddingException{

    IvParameterSpec iv = new IvParameterSpec(initVector);
        SecretKeySpec skeySpec = new SecretKeySpec(key, "AES");
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");
        cipher.init(Cipher.DECRYPT_MODE, skeySpec, iv);
        byte[] original = cipher.doFinal(value);
        return original;
```

```
  }
}
```

Listing B.23: DecryptionService class.

B.8 Classes used to execute operations with protected data on HBase

B.8.1 *Extension to the `HBaseStore` class*

```
public class SymHBaseStore extends HBaseStore{

  public SymHBaseStore(String file, String storageDir) throws
      MasterNotRunningException, ZooKeeperConnectionException, IOException {
    super(file, storageDir);
  }

  @Override
  public HTableInterface createTableInterface(Configuration conf, String
      tableName) throws Exception {
    return new SymColTable(conf, tableName);
  }
}
```

Listing B.24: SymHBaseStore class.

B.8.2 *Extension to the `HTable` class*

```
public class SymColTable extends HTable {
  static final Log LOG = LogFactory.getLog(SymColTable.class.getName());

  private final static byte[] keyFile =  new byte[16];
  private final static byte[] initVector = new byte[16];
  private final EncryptionService encService;
  private final DecryptionService decService;

  public SymColTable(Configuration conf, String tableName)
      throws InvalidKeyException, IOException, NoSuchAlgorithmException,
      NoSuchPaddingException, FileNotFoundException,
      InvalidAlgorithmParameterException {
    super(conf, tableName);
    encService = new EncryptionService(keyFile, initVector);
    decService = new DecryptionService(keyFile, initVector);
```

```
}

@Override
public ResultScanner getScanner(Scan scan) throws IOException{
  SingleColumnValueFilter scanFilter = (SingleColumnValueFilter) scan.
      getFilter();
  byte[] value = scanFilter.getComparator().getValue();
  byte[] cq = scanFilter.getQualifier();
  byte[] cf = scanFilter.getFamily();
  CompareOp compareOp= scanFilter.getOperator();
  Scan encScan = new Scan();
  if (scan.getAttribute("protected:" + new String(cf) + ":" + new String (cq
      )) != null){
    try {
      encScan.setFilter(new SingleColumnValueFilter(cf, cq, compareOp,
          encService.cipher(value)));
    } catch (InvalidKeyException e) {
      e.printStackTrace();
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    } catch (NoSuchPaddingException e) {
      e.printStackTrace();
    } catch (InvalidAlgorithmParameterException e) {
      e.printStackTrace();
    } catch (IllegalBlockSizeException e) {
      e.printStackTrace();
    } catch (BadPaddingException e) {
      e.printStackTrace();
    }
  }else{
    encScan.setFilter(new SingleColumnValueFilter(cf, cq, compareOp, value))
        ;
  }
  return super.getScanner(encScan);
}

@Override
public void put(Put put) {
  try {
    Put encPut = new Put(put.getRow());
    CellScanner cs = put.cellScanner();
    while (cs.advance()) {
      Cell cell = cs.current();
      String cf = new String(CellUtil.cloneFamily(cell));
      String cq = new String(CellUtil.cloneQualifier(cell));

      if (put.getAttribute("protected:" + cf + ":" + cq) != null) {
```

```
          byte[] value = CellUtil.cloneValue(cell);
          byte[] encValue = encService.cipher(value);
          encPut.add(CellUtil.cloneFamily(cell),
              CellUtil.cloneQualifier(cell), encValue);
        } else {
          encPut.add(CellUtil.cloneFamily(cell),
              CellUtil.cloneQualifier(cell),
              CellUtil.cloneValue(cell));
        }
      }
      super.put(encPut);
    } catch (IOException ex) {
      LOG.info(ex);
      throw new IllegalArgumentException(ex);
    } catch (IllegalBlockSizeException ex) {
      LOG.info(ex);
      throw new IllegalArgumentException(ex);
    } catch (BadPaddingException ex) {
      LOG.info(ex);
      throw new IllegalArgumentException(ex);
    } catch (InvalidKeyException e) {
      e.printStackTrace();
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    } catch (NoSuchPaddingException e) {
      e.printStackTrace();
    } catch (InvalidAlgorithmParameterException e) {
      e.printStackTrace();
    }
  }
}
```

Listing B.25: `SymColTable` class.

Project funded by FEDER funds on the scope of the “Programa Operacional Competitividade e Internacionalização - COMPETE 202” and by national funds through the FCT - Fundação para a Ciência e a Tecnologia on the scope of the project “POCI-01-0145-FEDER-006961”.