TÉCNICO LISBOA

FaceID-Cloud

Face Recognition Leveraging Utility and Cloud Computing

Ricardo Daniel Marques Caldeira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairperson:	Prof. Doctor Mário Rui Fonseca dos Santos Gomes
Supervisor:	Prof. Doctor Luís Manuel Antunes Veiga
Member of the Committee:	Prof. Doctor David Manuel Martins de Matos

May 2013

Acknowledgements

I would like to thank my supervisor Luís Veiga for all the effort he has put on helping me complete this work, which proved to be a worthy and ever too interesting challenge, but not impossible.

I would also like to thank all the Distributed Systems Group students and professors from INESC-ID which helped me along the way, providing useful thoughts and ideas whenever I needed and guiding me through the (hopefully) right path.

Finally, to my fellow colleagues at IST from whom I have learned a lot and which have accompanied me during these 5 short years till the end, in no special order: Pedro Amaral, João Silva, Pedro Fonseca, Nuno Ramos, David Maia, Daniel Pinto, Francisco Raposo, Eugénio Ribeiro, Eduardo Camões, Paulo Gonçalves, Nuno Costa, Sérgio Almeida, Pedro Mota, Nuno Diegues, Hugo Rodrigues, Alexandre Dias and Bruno Santos. Thank you guys.

This work was partially supported by national funds through FCT - Fundação para a Ciência e a Tecnologia, under projects PTDC/EIA-EIA/102250/2008 and PEst-OE/EEI/LA0021/2011.

Lisboa, May 2013

Ricardo Daniel Marques Caldeira

Aos meus avós, Carmina, Leonel, Helena e José, porque ‘‘acabar o curso é o que mais importa’’,

aos meus pais, António e Ana, pelo pouco que me viram durante este trabalho e por me terem trazido aqui,

e à minha irmã Mara, por já não poder ouvir falar em tese, até ser a vez dela de estar aqui.

Resumo

A detecção e identificação de caras humanas tem sido uma àrea de estudo intenso ao longo das últimas décadas, com diversas estratégias propostas com alguns bons resultados. No entanto, é uma tarefa computacionalmente intensiva, especialmente em vídeos que podem atingir um tamanho considerável. Assim, é importante desenvolver novas soluções que melhorem o desempenho de métodos de identificação de caras. Actualmente, um dos paradigmas de tecnologia mais promissores que pode ser utilizado para atingir este fim é a Computação na Nuvem, permitindo o desenvolvimento de sistemas escaláveis e flexíveis com um acesso *on-demand* a recursos virtuais num modelo utilitário em que se paga apenas o que realmente se utiliza. O objectivo deste trabalho é estudar a melhor forma de integrar um método de identificação de caras com uma infraestrutura de nuvem, e propor um sistema que tire partido destes recursos para aumentar consideravelmente o desempenho do reconhecimento facial em grandes bases de dados de vídeos.

Abstract

Detection and identification of human faces has been an intense area of study in the past decades, with many strategies being proposed with some good results. Yet, it is a computationally intensive task, especially in videos that can reach a considerable size. Hence, it is important to develop new solutions to improve the performance of face identification methods. At the moment, one of the most promising technology paradigms that can be used to achieve this result is Cloud Computing, allowing for scalable and flexible systems to be built with an easy on-demand access to virtual resources in a pay-as-you-go utility model. The purpose of this work is to study the best way to integrate a face identification method with a cloud infrastructure, and propose a system that leverages cloud resources to greatly improve facial identification performance in large video databases.

Palavras Chave
Keywords

Palavras Chave

Identificação de Caras em Vídeos, Computação na Nuvem, Computação Utilitária, Escalabilidade, Virtualização, Escalonamento Orientado a Dados

Keywords

Video Face Identification, Cloud Computing, Utility Computing, Scalability, Virtualization, Data-Driven Scheduling

Index

List of Figures

List of Tables

Acronyms

API Application Programming Interface 51

EDSF Experimentally Determined Serial Fraction 69

HDFS Hadoop Distributed File System 35

ID Identifier 14

IT Information Technology 1

PCA Principal Component Analysis 25

SOA Service-Oriented Architecture 8

VM Virtual Machine 52

1 Introduction

Over the last couple of decades the world has been witnessing the evolution and expansion of the Information Technology (IT) area at a very fast rate, especially since the appearance of the Internet and the World Wide Web. This led to the appearance of several new paradigms that revolutionized the way information is processed, among which there is Cloud Computing. Even though a newcomer, making its debut only during the last few years, all the major IT companies like Google, Amazon and Microsoft are now fully aware of its power and usefulness for IT businesses and, indirectly, to the end-user. A great number of services typically used by today's average user are now cloud-based (e.g. Amazon online store) and the tendency is for this type of services to grow in number as companies start to acknowledge benefits of cloud computing such as on-demand allocation of resources and utility pay-as-you-go payment models. Given this facilitated access to computing and storage resources provided by emerging cloud platforms, it makes sense to design new cloud-based systems capable of presenting end-users with out-of-the-box solutions that can be used from almost any Internet-capable device.

In this context, the recognition of human faces in videos is a good example of a research area that could greatly benefit from exploiting the cloud potential, especially when dealing with huge databases of videos that can easily reach the terabytes mark. The nature of this problem implies a massive computational effort from the start, since video processing is a computationally heavy task and a face identification algorithm executing on top of it only aggravates this problem. Yet, video processing can be seen as belonging to the class of the generically dubbed ''embarrassingly parallel'' problems, meaning that it is easily divided into sub-problems that should not, considering a sequentially-framed raw video footage, have data dependencies between them. However, video technology has been evolving to a point where very different and complex codification algorithms, containers and formats are used, imposing an additional computational/developing effort to either decode and transform videos to a sequentially-framed raw format before analyzing them, or interpret these complex formats on-the-go when possible.

Although the interpretation of video formats plays a big role on the performance of video processing, it is not the object of research of this work, and so we should assume a black-box behavior from a third-party application to interpret the several different video formats available and perform a translation to a known format. Knowing this, the main focus of this research is on

how to bring the two sides of the original problem together, or, in other words, how to design and implement a parallelized version of an existing algorithm for face identification in videos on top of the distributed environment of Cloud Computing, and it is with this goal in mind that we propose a solution to the problem in the form of the system FaceID-Cloud.

1.1 Objectives and Contribution

The contribution of this work is to study an efficient way to solve the problem of integrating an existing facial recognition method with the technologies which are part of the Cloud Computing paradigm, the main goal being to greatly increase the performance of one such technique when compared with a sequential approach. The resulting system should be able to process video material using the resources provided by a cloud infrastructure, and identify the people that appear in it given a database of known individuals. Unknown faces should be analyzed and grouped by their similarity, so that an end-user can tag the group with a name and add another person to the system. From that moment on, that person will be identified and no more user input should be needed. All processed information should be adequately stored and indexed so that useful queries can be made, namely to know which people appear in a video and which videos does a person take part in. The system should be able to scale horizontally (computation- and storage-wise) and increase its performance in proportion to the on-demand addition of new nodes. Also, it should strive for efficiency in terms of communication steps and message size so as to reduce the load on the network which, from a high-level pre-analysis of the system's most probable bottlenecks, should be the most scarce resource available. Summarily, the main contributions of this work are:

- Horizontally scalable system that leverages cloud computing resources
- Dynamically sized infrastructure based on system load, suitable for pay-as-you-go-models
- Distributed system to accelerate human facial recognition on videos
- Ranged search efficient storage strategy of vectors in a key/value database

It is not in the scope of this work to develop a new face identification method or improving the core techniques of an existing one. Any modifications or improvements should only facilitate the implementation of the system and should not be made on the core of the method, as it is not the focus of this work. Thus, one of the most basic face identification methods was chosen to be used, *eigenfaces* by Turk and Pentland (Turk & Pentland 1991), so that the focus is on the middleware design and the integration of the method with cloud technologies. While not in the main objectives list, as a secondary goal it is important to design the system in such a way that it can, in the future, accept different face identification methods.

1.2 Document Structure

This document will start with a survey on previous works and state-of-the-art technologies related with this project in the Related Work (see Chapter 2), where three main subjects will

be analyzed: cloud computing, data-driven systems and face recognition. Each of these areas is closely related with this work's system, and should provide a foundation to start with. Next, the design process of the solution's architecture is presented in the Solution Design (see Chapter 3), by specifying the main architectural drivers behind it and several views of the system at different levels of detail. The text continues with a description of the implementation process, covering themes such as the technologies chosen and how the design description was materialized in the Implementation (see Chapter 4). Finally, we conclude with a description of the deployment process and an evaluation of the developed system's prototype performance on the Evaluation (see Chapter 5). The final chapter contains a wrap-up of the work, along with a discussion of what could be improved in the current solution and some final remarks in the Conclusions (see Chapter 6).

1.3 Other Publications

Part of this work can be seen in the INForum 2012 paper (Caldeira & Veiga 2012), published during the development of the system.

2 Related Work

In this section a survey on the background subjects of this work is made, providing the theoretical and technological foundations necessary to develop a solution for the problem. Since this work depends on knowledge from very different areas, a division was made into three distinct subjects that will be studied: Cloud Computing, Data-Driven Scheduling and Face Identification.

2.1 Cloud Computing

During the past few years, the IT world witnessed the birth and growth of a new paradigm, commonly called **Cloud Computing**, although it has also been named as Dynamic Computing (Rajan & Jairath 2011). It is difficult to assign a precise date for its genesis, since the term "cloud" has already been used in several contexts, describing large ATM networks in the 1990s for instance (Zhang, Cheng, & Boutaba 2010), and is also based upon some already existing technologies like distributed computing, virtualization or utility computing which have been around for several years (Vaquero, Rodero-Merino, Caceres, & Lindner 2008). However, some (Gong, Liu, Zhang, Chen, & Gong 2010; Vouk 2008) claim that the true birth of Cloud Computing happened when IBM and Google announced a partnership in this domain (IBM & Google 2007), leading to a hype around the subject and lots of popularity.

In the following subsections, a possible definition for Cloud Computing will be presented after its evolution and main characteristics (Section 2.1.1), followed by a description of some concepts related to the cloud paradigm (Sections 2.1.2.1 and 2.1.2.2) and the most significant enabling technologies (Section 2.1.2.3). Finally, some running research challenges (Section 2.1.3) are presented. For completeness, a survey and classification of some existent cloud systems is also given on Tables B.3*a* and B.3*b* in Appendix B.

2.1.1 Overview of Cloud Computing

Cloud Computing is a much younger paradigm when compared with its older ''siblings'' who have been around for years, like Grid Computing or Cluster Computing, meaning that it still needs a standardized and generally accepted definition. In an online article (Geelan 2009), twenty-one IT experts define Cloud Computing according to their vision and experience in the area, but the answers are somewhat disjoint and seem to focus on just certain aspects of the technology, lacking a global analysis of the proposals (Vaquero, Rodero-Merino, Caceres, & Lindner 2008). Some

of the experts and other authors focus on *immediate scalability*, *elasticity* and *dynamic resource provisioning* as the key characteristics for the Cloud (Markus Klemns in (Geelan 2009) and others in (Tsai, Sun, & Balasooriya 2010; Rajan & Jairath 2011)), while others disagree with this vision focusing on the type of service as the main concept (Brian de Haaf in (Geelan 2009) and others in (Tsai, Sun, & Balasooriya 2010)). Some other existing perspectives are based on the business model, characterizing Cloud Computing as a *pay-as-you-go based service* and with the potential to reduce costs by the realization of *utility computing* (Jeff Kaplan in (Geelan 2009) and others in (Bojanova & Samba 2011; Buyya, Yeo, & Venugopal 2008; Wang, Laszewski, Younge, He, Kunze, Tao, & Fu 2010)).

2.1.1.1 From Mainframes to Cloud Computing

Even though Cloud Computing is considered a new paradigm, its roots come from far back in the past. In the 1960s, large mainframes could be accessed by several thin clients to process bulk data, very much like today's users can use personal computers, tablets, smartphones and other computing devices to request a service from a cloud provider. Thus, one could say that the hype around the cloud does not have a great reason to exist, as it seems to have just brought back an old and well known concept. Steve Mills, Senior Vice-President and Group Executive at IBM, illustrated this point in a 2008 interview[1] stating that "We [IBM] have been running multitenancy for decades and decades" and that "Everything goes back to the beginning, the mainframe", when asked about his company's view on Cloud Computing. Although it is true that mainframes and the present cloud systems share some basic characteristics, there are some clear distinctions to be made in respect to computing power, storage capacity and scalability. According to (Voas & Zhang 2009), a mainframe offers finite computing power, being a physical machine, while cloud systems offer's a very high degree of scalability regarding power and capacity. Also, as opposed to the simple terminals used to access mainframes, today's computers have significant computing power and storage capacity on their own, allowing for a certain degree of local computing and caching support.

In this perspective, some authors describe the technology evolution steps from the mainframe paradigm to the cloud one (Voas & Zhang 2009; Rajan & Jairath 2011), although the descriptions do not match exactly. It started, of course, with mainframes shared by several users and accessed through terminals. However, this strategy is not financially feasible for a single person, leading to the birth of the personal computer era, defined as the second stage in the evolution by the authors. Nonetheless, there were still considerable costs for companies, since each computer needed its own application interfaces and databases. With the appearance of local networks, the third stage, the client-server model helped organizations to reduce costs by featuring a centralized database access

[1] http://news.cnet.com/8301-13953_3-9933108-80.html (accessed April 2013)

in the servers and the application interfaces in the clients. Yet, this model still has limited resources and cannot be applied globally in an efficient and effective way. The fourth stage saw the advent of local networks that could connect to other local networks, giving birth to the World Wide Web and the Internet, providing no single point of failure, no single point of information, no single owner and no single user or service provider. All these stages laid the foundation for the appearance of several paradigms based on distributed information systems, including, of course, Cloud Computing.

2.1.1.2 Economy of Scale

An important concept that helps define Cloud Computing is that of *Economy of Scale* (Vouk 2008; Zhang, Cheng, & Boutaba 2010). The rationale behind large data-centers is based on the notion of reductions to the unitary cost as the size of the facility increases, allowing companies to focus on specializing and optimizing their products. In the context of Cloud Computing, an economy of scale is usually achieved by building a data-center with a very high number of commodity-class systems, leveraging the lower price and also the lower maintenance costs of this type of hardware. In this way, network equipment, cooling systems, physical space, and other resources can be efficiently exploited, where otherwise, with fewer high-end systems, resource utilization efficiency would severely drop. Since the cost and computing power of a single system is so insignificant when compared to the whole data-center, providers can even cope with several failures without bothering to replace individual nodes. Also, to expand the data-center it suffices to acquire more commodity computers and add them to the existing bundle. By reducing their costs with data-centers, cloud providers can lower the prices for their clients, turning Cloud Computing into a competitive and feasible business.

2.1.1.3 Utility Computing

During the past century, society has become accustomed to utility services such as electric power, water, natural gas, telephone access, and many others, including Internet access more recently. The infrastructure supporting these services has evolved to an utility-based business model, where customers pay solely for what they use and do not have to own the whole necessary equipment, what would be financially infeasible. This way of thinking is now embedded in our mentality, where it does not make sense to possess an expensive power generator at home, for instance, when there are companies who can supply the same service at a lower cost based on economies of scale. All these services are shaped by a combination of several requirements like ease of use, pay-as-you-go charges, high reliability among others (Rappa 2004).

Given the concept of an utility service described above, the Utility Computing paradigm can be described as the on-demand delivery of infrastructure, applications and business processes in a security-rich, shared, scalable and standards-based computer environment over the Internet. Customers can access IT resources as easily as they get their electricity or water, and are charged in a

pay-as-you-go basis (Rappa 2004).

2.1.1.4 Service-Oriented Architecture

In the IT business world, Service-Oriented Architecture (SOA)s are gaining ever more importance as universal models to which automation and business logic conforms. Service-orientation aims to cleanly partition and consistently represent available resources, simplifying technical disparities by applying abstraction layers and providing them as services, which in turn lay the foundation for standards for representing logic and information. The main principle is that one should leverage the potential of individual services existing autonomously yet not completely isolated from each other, seeing that they must conform to a set of principles that allow them to evolve independently, while assuring some commonality and standardization. Thus, SOAs have the potential to support and promote these principles throughout the business process and automation domains of an enterprise when realized through a web-services platform (Erl 2005). Cloud Computing is closely related to SOA in that it aims to supply several different services, providing the necessary technologies and flexible platform for companies to build their SOA solutions in a cost-effective way.

2.1.2 Defining Cloud Computing

Before presenting a possible definition for the cloud paradigm, it is important to describe the distinctive technical characteristics which differentiate it from earlier related paradigms. Note that there can be dependencies between the listed characteristics, in the sense that one is achieved by relying on the existence of the other, but it is relevant to make the distinction as they are significant in the classification of cloud systems.

On-demand service provisioning. One of the key features of Cloud Computing is the possibility to obtain and release computing resources on the fly, following the real necessities of an application. This avoids both having to pay for unused resources and the exact opposite, missing resources during periods of peak demand.

Service orientation. As mentioned before in Section 2.1.1.4, Cloud Computing aims to provide different services and act as a foundation for SOA solutions. The business models around the cloud are service-driven, hence a strong emphasis is placed on service management. A description of the most significant cloud service models is given in Section 2.1.2.1. In order to protect both cloud providers and customers, a Service Level Agreement (SLA) is negotiated, where the provided services and conditions are defined. SLAs oblige providers to live up to the Quality of Service (QoS) terms they committed to in the SLA and assure customers that the services they paid for will be fully provided. On the other hand, SLAs can also protect providers from having to cope with unreasonable requests. Yet, SLAs still constitute one of the hardest challenges to overcome,

as cloud providers cannot usually guarantee an agreed QoS, forcing some companies to refrain from migrating their services to the cloud. This problem will be discussed in the Challenges Section (see 2.1.3).

Scalability and Flexibility. A scalable and flexible infrastructure is central to cope with different geographical locations of the resources, disparate hardware performance and software configurations. Also, to support a dynamic service provisioning, the computing platforms should be flexible to adapt to the requirements of a potentially large number of users without violating the QoS conditions of the SLAs (Wang, Laszewski, Younge, He, Kunze, Tao, & Fu 2010).

Pay-per-use utility model. Some of the experts in (Geelan 2009) referred a strong connection between Cloud and Utility Computing, in the sense that the former is the realization of the earlier ideals of the latter. An utility service model is usually applied to Cloud Computing featuring a pay-per-use billing, where customers only pay for what they really use or for small predefined quanta (e.g. 1 hour of CPU, 1GB of storage, etc).

Ease of access. A utility-model requires a ubiquitously accessible service (following traditional public utilities like electricity or water supplies) which can be used from any connected devices over the Internet, with simple to use and well defined interfaces and with very few requirements from the client system (Wang, Laszewski, Younge, He, Kunze, Tao, & Fu 2010).

Virtualization. Virtualization techniques provide the means to achieve most of the above mentioned features, namely by partitioning hardware and thus acting as the base for flexible and scalable computing platforms (Wang, Laszewski, Younge, He, Kunze, Tao, & Fu 2010). Virtualization is, thus, one of the major underlying and enabling aspects of the Cloud Paradigm. This subject will be described in more detail in Section 2.1.2.3.

Multi-tenancy. Cloud infrastructures must be capable of serving multiple clients, or tenants, under the same hardware or software infrastructure to achieve the goal of cost effectiveness. There are always some issues in need to be addressed in order to achieve the full potential of multi-tenancy, like security isolation, customizing tenant-specific features or efficient resource sharing, which are usually handled by leveraging the virtualization support of the cloud infrastructure.

Given the set of concepts described, a possible definition for the cloud paradigm can now be given, based on the work of (Vaquero, Rodero-Merino, Caceres, & Lindner 2008): ***Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development tools and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale) and can be accessed by several users simultaneously (multi-tenant), allowing thus for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered***

by cloud providers by means of customized SLAs. Note that the Cloud concept is still changing and this definition shows how it is conceived today.

2.1.2.1 Cloud Service Models

Cloud Computing has been widely described has a service-oriented paradigm by several authors (Vaquero, Rodero-Merino, Caceres, & Lindner 2008; Tsai, Sun, & Balasooriya 2010; Rajan & Jairath 2011; Bojanova & Samba 2011; Prodan & Ostermann 2009; Gong, Liu, Zhang, Chen, & Gong 2010) and the IT community in general. It is clearly one of the major characteristics that distinguish it from similar paradigms, like Grid Computing. Although the SOA (Software Oriented Architecture) concept is common to both Cloud Computing and Grid Computing, it reaches a more practical state in the former (Gong, Liu, Zhang, Chen, & Gong 2010).

There are two key properties that enable Cloud Computing to be service-oriented in agreement with the utility philosophy, *abstraction* (Vaquero, Rodero-Merino, Caceres, & Lindner 2008; Gong, Liu, Zhang, Chen, & Gong 2010; Vouk 2008) and *accessibility* (Vouk 2008). In order to achieve the type of interaction described earlier with utility computing, it is necessary to hide all the complexity of the cloud architecture from the end-user and to present an user-friendly interface for requesting services. In respect to the first requirement, the abstraction property is best described through a layered diagram which identifies the several parts of the cloud architecture and the relations between them. Each layer is, of course, built in a hierarchical way on top of one another and represents a specific type of service to be provided based on the service provided by the subjacent layer.

Note that each layer can in fact offer a service for end-users even if it is not the top layer, distinct from the common software engineering notion of a system build from several modular layers, where often the top one alone provides some service. Virtualization, described in Section 2.1.2.3, is a key requisite to attain abstraction without exposing the underlying architecture details to the users. As for the accessibility requirement, the common practice is to provide a standardized API which respects a given protocol (e.g. SOAP, REST) and enables an easy access to the service from anywhere in the world. This easy access to cloud services is likely to be one of the main boosters of its ongoing acceptance in the non-academic community in a relatively short period of time (Weinhardt, Anandasivam, Blau, & Stöß er 2009).

A description of the several cloud service models is now presented, starting with an hierarchical view for Cloud Computing on Fig. 2.1 followed by the description of the service model associated with each layer in the hierarchy.

Data Centers. The Data Centers layer is the foundation for Cloud Computing technologies, providing raw computation, storage and network hardware resources. Usually, they are built in less populated ares with cheaper energy rates and low probability of natural disasters, being constituted by thousands of inter-connected servers (Tsai, Sun, & Balasooriya 2010).

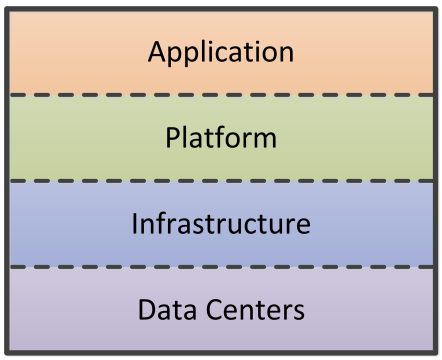


Figure 2.1: Cloud Hierarchical Layer View

Infrastructure-as-a-Service (IaaS). The Infrastructure layer focuses on leveraging the subjacent data-center's resources and provisioning them as services to consumers by virtualizing storage capacity, computational power, communications and other fundamental computational resources necessary to run software. Companies offering this kind of services are commonly called *Infrastructure Providers* (IPs) (Vaquero, Rodero-Merino, Caceres, & Lindner 2008). Even though users do not directly control the physical infrastructure, they are able to scale their virtualized resources up and down dynamically according to their needs. This is in line with the utility perspective, where consumers use and pay only for what they do consume. Also, the virtualization of resources is of great value for IPs, since they can focus on maximizing the efficiency of their infrastructure and therefore make it more profitable. An example of a commercial solution in this domain is Amazon's Elastic Compute Cloud (EC2).

Inside the IaaS domain it is still possible to define some sub-categories, although there is less agreement from the IT community in these and the naming is primarily performed by business marketing people trying to differentiate their product's characteristics. First, note that the term *storage* above does not clearly identify the type of service. Is the data organized in some way as in database or is it just stored in bulk as a backup? From this perspective *Database-as-a-service* (DBaaS) and *Storage-as-a-Service* (StaaS) emerge. The former is centered on providing a fine-grained access to data by delivering database functionality as a service, supporting multi-tenancy, automated resource management and other features of traditional database service systems (Oracle 2011). The latter refers simply to cloud storage leveraging virtualized storage resources, with no complex data querying functionalities provided as in DBaaS. Another sub-category of IaaS derives from the virtualization of communication services, being called *Communications-as-a-Service* (CaaS). In this field, communications platforms like VoIP are offered as a service, freeing companies from the financial burden of building a platform from scratch.

Platform-as-a-Service (PaaS). The second layer, Platform layer, adds an additional abstraction level to the services supplied by the Infrastructure Layer, effectively aggregating them and offering

a complete software platform to assist in application design, development, testing, deployment, monitoring. Namely, it provides programming models and APIs for cloud applications, MapReduce (Dean & Ghemawat 2008) for example. Also, problems like the sizing of the resources demanded by the execution of the services are made transparent to developers (Vaquero, Rodero-Merino, Caceres, & Lindner 2008).

Software-as-a-Service (SaaS). Finally, the Application layer presents software to end-users as an on-demand service, usually in a browser, being a model already embraced by nearly all e-business companies (Prodan & Ostermann 2009). It is an alternative to locally running applications with some clear advantages over them. First, applications hosted in the cloud do not require users to install and continuously update applications on their own computers, saving time and disk space, and enhancing accessibility and user-friendliness. This also helps software providers, since updates are automatically distributed to users, needing only to alter the few instances running in the cloud infrastructure. Also, the software developers can target their applications to work in a known and controlled environment, raising the quality and security of the products.

2.1.2.2 Cloud Deployment Models

Another way of classifying a Cloud infrastructure is according to its type of deployment. Currently, there are three main deployment models acknowledged by the IT community, *public*, *private* and *hybrid* clouds (Zhang, Cheng, & Boutaba 2010; Rajan & Jairath 2011; Bojanova & Samba 2011; Sakr, Liu, Batista, & Alomari 2011), although some other models are also considered such as *community* (Bojanova & Samba 2011; Sakr, Liu, Batista, & Alomari 2011) and *virtual private* (Wood, Shenoy, Gerber, & Ramakrishnan 2009; Zhang, Cheng, & Boutaba 2010) clouds.

Public or Hosted Clouds. In this deployment model, the cloud infrastructure is owned by organizations selling cloud services who offer their resources to the general public or large industry groups (e.g. Amazon, Google, Microsoft). Public clouds offer several benefits to service consumers since there is no initial investment on infrastructure and the related inherent risks are shifted to infrastructure providers. However, there are some drawbacks, like the lack of fine-grained control over data, network and security settings, hindering the effectiveness in some business scenarios.

Private Clouds. Although the cloud ecosystem has evolved around public clouds, organizations are showing interest in open source cloud computing tools which let them build private clouds using their own or leased infrastructures. Thus, private cloud deployments' primary goal is not to provide services over the Internet, but to give users from the organization a flexible and agile private infrastructure to run service workloads within their administrative domains. This model offers the highest degree of control over the performance, reliability and security of the resources, and improves resource reuse and efficiency, but the advantages of a public cloud are lost, since the organization itself has to perform maintenance and entails up-front costs in case of not owning the

infrastructure. It embodies, to some extent, the late adoption of Grid Computing principles by corporate IT.

Hybrid Clouds. Both public and private cloud approaches still have some limitations, namely the lack of control and security in the former and the difficulty of expanding or contracting the resources on-demand in the latter. A common solution is an hybrid deployment model, where unique clouds are merged, bound by technology that enables interoperability amongst them, taking the best features of each deployment model. In an hybrid cloud, an organization builds a private cloud over its own infrastructure, maintaining the advantage of controlling aspects related to performance and security, but with the possibility of exploiting the additional resources from public clouds on an as-needed basis (e.g. when there is a demand peak on the infrastructure of an online retailer in holiday season). On the downside, designing a hybrid cloud requires a careful decision on the best split between public and private cloud components and the cost of enabling interoperability between the two models.

Community Clouds. Community clouds are somewhat of a middle way between public and private clouds. They are not owned by a single organization as in a private cloud, but are also not completely open to the general public as commercial services. The cloud infrastructure is usually provisioned for exclusive use by a specific and limited community of organizations with shared concerns or interests.

Virtual Private Clouds. Another way of dealing with public and private clouds limitations is called Virtual Private Cloud (VPC). In this deployment model, a platform is built on top of a public cloud infrastructure in order to emulate a private cloud, leveraging Virtual Private Network (VPN) technology. In this way, users are allowed to design their own topology and security settings, having a higher level of control than allowed by simple public clouds, while retaining easy resource scalability. VPC is essentially a more holistic design since it not only virtualizes servers and applications, but also the underlying communication network as well, facilitating the task of migrating services from proprietary infrastructures to the cloud.

2.1.2.3 Enabling Technologies

There is a set of well-understood technologies which constitute the foundation for cloud infrastructures. Without enabling technologies like *virtualization*, *web services*, *distributed storage systems*, *programming models*, just to mention a few, most of the cloud paradigm features would not be attainable. A brief description of these subjects will now be presented.

Virtualization Technology. Virtualization is one of the key components that render cloud computing service models possible. Each layer in the architecture can be provided on-demand as a service through virtualized resources, which not only provide users with access to controlled en-

vironment and performance isolation, but allow infrastructure providers to achieve flexibility and scalability of their hardware resources. Also, virtual machines act as a sandbox environment, enforcing a high security level and isolation between instances. In the context of Cloud Computing, virtualization is achieved through special purpose applications called *Virtual Machine Monitors* (VMM), or *hypervisors*, which simulate an hardware environment for deploying guest operating systems, while controlling every aspect regarding resources and interactions. Among the most widely used VMMs are the Xen Hypervisor[2], KVM[3] and VMware ESX[4].

Web Services. A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. The generalized acceptance of Web services in the e-commerce business, namely in the service providing industry, has greatly contributed for the growth and recognition of the Cloud Computing paradigm. Cloud services are usually exposed as Web services, being easily accessible from anywhere in the world and leveraging the standard mechanisms provided to interact with different types of clients. An example of this technology is Amazon Web Services[5].

Distributed Storage System. The underlying distributed storage system where the various cloud layers are built upon is, of course, another important technology necessary for Cloud Computing. Built on top of large distributed data centers, it provides scalability and flexibility to cloud platforms, allowing data to be migrated, merged and managed transparently to end-users for whatever data formats (Wang, Tao, Kunze, Castellanos, Kramer, & Karl 2008). An example of this technology is the Google File System (Ghemawat, Gobioff, & Leung 2003) and Amazon S3[6]. A cloud storage model should not only focus on how to store and manipulate large amounts of data, but also on how to access it in a semantic way. Since the goal of a distributed storage system is to abstract the actual physical location of data, this access should be performed by logical name and not by the physical name. For example, in Amazon S3 each object is assigned a key and is accessed through a pure Identifier (ID), which does not identify a physical location.

Programming Model. In order to comply with the ease of use requirement from the utility computing vision, some Cloud programming models should be proposed for users to adapt more easily to the paradigm, as long as they are not too complex or too innovative. One of the most known

[2]http://xen.org/
[3]http://www.linux-kvm.org/
[4]http://www.vmware.com/
[5]http://aws.amazon.com/
[6]http://aws.amazon.com/s3/

programming models is MapReduce (Dean & Ghemawat 2008), which allows the processing and generation of massive data sets across large data centers.

2.1.3 Challenges

Despite the growing popularity around Cloud Computing, there are still several challenges that need to be overcome. These problems do not prevent the application of cloud technologies – they are being used today – but introduce difficulties when trying to take full advantage of its characteristics.

Cloud Provider Interoperability. Most cloud providers spend time and money developing an API for their cloud infrastructure, giving users a simplified and documented way to access cloud services. However, there is no consensus on a standardized API between providers, as each has already invested many resources in their solution. This situation forces companies to invest on and develop new applications for one specific cloud platform, as it is not financially feasible to cover the whole spectrum of existing platforms. Customer lock-in may be attractive to cloud providers, but users are vulnerable to price increases, to reliability problems or even to providers going out of business (Armbrust, Joseph, Katz, & Patterson 2009).

Coupling Between Components. It is common practice for cloud providers to possess the whole suite of platforms necessary for clients to build full applications. Yet, this usually means that there are dependencies between these modules, where if one is to be used, it needs another one on which it depends to function correctly. Thus, one is stuck with a full suite of computing services from the same provider with a high cost to change, exactly as with the APIs.

SLA Support. Service Level Agreements are the major obstacle for the wide adoption of cloud computing today. Cloud providers are still struggling to achieve the level of guarantees needed for companies to use cloud infrastructures for serious business deployment. Until providers are capable of signing the SLAs and comply with their requirements, most large organizations will not rely on the cloud for mission-critical computing needs since it cannot provide the necessary customization and service guarantees. Also, the business is very dynamic, while SLAs are quite static, meaning that they are not able to adapt to changes in business needs (Tsai, Sun, & Balasooriya 2010).

Security. Most organizations will not have their sensitive corporate data on the cloud, whether because a public cloud system is more exposed to attacks, or internal laws preventing customer and copyrighted data outside national boundaries, or even the concern that a country's government can get access to their data via the court system. These are all well justified reasons, but there are no fundamental obstacles on making a cloud environment as secure as a private data center. Technologies such as encrypted storage, VPC and firewalls are capable of providing the necessary security protections needed by the customer organizations, although in some countries the usage of some kinds of encryption can be highly regulated and even illegal (e.g. China).

2.1.4 Cloud Computing in the Proposed System Context

This area has, of course, a major influence in the system to be designed, as it should be implemented on top of a cloud platform and leverage its resources. This means that all the advantages of the cloud paradigm will be made available, such as elastic resource provisioning and scalability for example, but also some of the problems. The system is envisioned to be built on top of an IaaS-oriented private cloud infrastructure, but nonetheless able to expand into a hybrid infrastructure if needed or even a full public cloud approach.

2.2 Data-Driven Scheduling

Nowadays, web and scientific applications need to deal with massive amounts of data. Social networks, video-sharing websites, search engines, physics simulators, they all have to handle an ever increasing volume of data per day. Given this scenario, data placements decisions are gaining more and more importance when designing a robust workflow planner, where computational jobs have usually been the most used methodology. Workflow planners need to become data-aware in order to tackle this emergent growth of data-intensive applications, taking into account problems like data placement, storage constraints or efficient resource utilization (Kosar, Livny, Street, & Wi 2004; Kosar & Balman 2009).

2.2.1 The Problem

CPU-centric schedulers have been the preferred method of running applications, namely in clusters/grids, as the focus of tasks is usually to leverage all the existing computing power to achieve some result in the shortest time possible, the CPU-time and network access being the dominant bottlenecks. However, the growth and spread of data-intensive applications is outpacing the corresponding increase in the ability of computational systems to transport and process data. Also, techniques which once worked well for CPU-intensive workloads in a local environment can suffer orders of magnitude losses in throughput when applied to data-intensive workloads in remote environments. This situation has been forcing a new perspective, where data access is viewed as the main bottleneck, turning the scheduling of data placement activities in a crucial problem to be solved and carefully coordinated with CPU allocation activities (Bent, Denehy, Livny, Arpaci-Dusseau, & Arpaci-Dusseau 2009; Sakr, Liu, Batista, & Alomari 2011).

2.2.2 Early Work on Data-Driven Systems

Even though the size of today' datasets were not a reality outside scientific projects until recently, there were already some concerns about the role of data in computational tasks a few decades ago. In (Treleaven, Brownbridge, & Hopkins 1982), data-flow (data-driven) was perceived as a possible fifth-generation architecture for computer systems, opposing the von Neumann principles

which are based on control-flow (CPU driven). This work was focused on the availability and flow of data inside a single machine, where operands would trigger the operation to be performed on them when all the required inputs are available for that operation. In a data-flow architecture each instruction can be seen as having a computing element allocated to it continuously waiting for the arguments to arrive, lying dormant whilst they do not.

Thus, *data-driven* is defined as the term to denote computation organizations where instructions passively wait for some combination of their arguments to become available. This strategy has the advantage that an instruction is executed as soon as its operands are available, leveraging a high degree of implicit paralelism. On the other hand, it can be a restrictive and time wasting model, as an instruction will not execute until it has all its arguments. The introduction of some non-data-driven instructions is possible to overcome this problem and provide some degree of explicit control.

2.2.3 Computational Jobs vs. Data Placement Jobs

There is no efficient solution based exclusively on computational job scheduling or data placement job scheduling when developing a data-intensive application, so it is necessary to bring them together in order to take full advantage of their best features. According to (Kosar, Livny, Street, & Wi 2004), in the world of distributed and parallel computing data placement activities are regarded as second class citizens, meaning that they cannot be queued, scheduled, monitored, managed and check-pointed, just as a computational activity. However, there has to be a different treatment for data placement jobs, as they have very different semantics and characteristics compared to computational jobs. For example, if a large file transfer fails, a full retransmission is not desirable. It should be possible to restart the transfer from the point where it failed or to try again using another protocol, for example, and a traditional computational job scheduler may not be able to deal with such cases. Thus, there should be different schedulers for each type of job, leveraging the different semantics of each one.

2.2.4 Definition

It is possible to establish a definition using the notion of data-driven architecture stated in Section 2.2.2 and the properties described in Section 2.2.3, adapted to the problems of today's data-intensive distributed applications. The principles are essentially the same, only the computing environment and the data dimension are different. In this perspective, ***a data-driven distributed system is an application where the availability of data drives the execution of the individual tasks, so that data placement activities are considered first-class citizens and thus can be scheduled, queued, monitored, managed and check-pointed as if they were computational jobs.*** This means that instead of having computational jobs triggering data

requests from a remote location across a throughput-limited network, it is possible to schedule data placement activities which will trigger the tasks needed to be performed on that data.

2.2.5 Data Aware Batch Scheduling

In data-intensive distributed systems, it is often necessary to start a high number of similar tasks, usually called a workload, which will process high volumes of data autonomously. By *similar* it should be understood as capable of running in the same system without manual intervention (e.g. same input/output formats). This process is commonly denoted by *batch scheduling*, being, for example, extensively used in scientific applications where problems are usually highly parallelizable, although a good scheduling must be performed in order to efficiently make use of the available resources, namely computing units, main memory, storage and network. As stated in previous sections, scheduling is widely performed with computation jobs as the main driver, but the constant growth of data is becoming a relevant problem, which means that batch schedulers should take data into consideration when planning. In this section, some relevant topics concerning the properties a data-aware batch scheduler should possess will be presented.

2.2.5.1 Efficient Resource Utilization

One of the main concerns of a data aware batch scheduler is to optimize the utilization of the available computing, storage and network resources, making sure that neither the storage space nor the network connections get overloaded, and the load is evenly distributed between computing units. In practice, this is strongly connected with the way parallelism and concurrency are used to maximize the overall throughput of the system, as showed by (Kosar & Balman 2009) (parallelism refers to the transfer of a single file using multiple streams while concurrency refers to the transfer of multiple files concurrently). The empirical results of this work showed that the effect of both methods in the efficiency of the system is different if it is tested on a wide area network (WAN) or on a local area network (LAN). In a WAN, the transfer rate of the system increases as expected, but in a LAN it comes to a threshold and additional streams and transfers causes it to decrease. Also, it was observed that CPU utilization at the clients increases with both parallelism and concurrency, while on the server only the concurrency level increases the CPU utilization. With an increased parallelism level, the server CPU utilization starts dropping and keeps this behaviour as long as the parallelism level is increased. It is interesting to note that concurrency and parallelism have completely different effects on the CPU utilization at the server side, probably due to the amortization of the select() system call overhead when monitoring several streams for the same file. Thus, it is important for a scheduler to conciliate both parallelism and concurrency, as the usage of each strategy by itself is not very effective, and is also important to adjust them dynamically, as it helps maximizing the transfer rates and lowers the CPU utilization on the servers.

2.2.5.2 Dedicated Storage Space Management

When scheduling data transfers to a host, it is important that the necessary space is available at the destination. In an ideal scenario, the destination storage system is capable of space allocations prior to the transfer itself, therefore assuring the scheduler that it can proceed with the transfer. However, this feature is not supported by some storage systems, so the scheduler has to keep track of the size of the data entering or exiting each host. Also, in a storage constrained environment it should apply some strategy that maximizes the most desirable quality of the data (e.g. size, age or some user or system wide utility measure) and takes into account all the jobs executing in the same host and their input and output sizes.

Assuming multiple uniform pipelined jobs to be executed in one host, four possible strategies are described in (Bent, Denehy, Livny, Arpaci-Dusseau, & Arpaci-Dusseau 2009) in order to maximize the throughput of the workload (i.e. to minimize the total time to completion). In these strategies, a set of pipelined jobs is called a *batch*, whereas the read-shared data between jobs is called *batch data*. Also, each pipeline has access to its *private data* that is not shared with other pipelines. Finally, each storage unit allocated to hold a particular data is called a *volume*. A description of the four strategies is now presented, starting with the one with less storage limitations to the one with the most constrained environment:

All Strategy. In this strategy, there are no storage constraints so all the space needed by the workload fits in the available storage. Thus, the planning is straightforward as no possible schedule can result in adverse effects. Also, since the storage space for all jobs executing in a given host can be readily allocated, there are no limits on the concurrency level between the jobs and no refetching of data is needed.

AllBatch Strategy. In the case where all batch data fits in the host storage, but not all private volumes can be allocated, an AllBatch Strategy can be applied, as long as there is at least one pipeline that can simultaneously allocate two volumes for its input and output. This strategy limits the system concurrency and may cause computing capacity underutilization if the number of pipelines that can allocate their private volumes is fewer than the number of available CPUs at the host.

Slice Strategy. When the system is even more storage-constrained compared to the previous two strategies, a Slice Strategy can be used whenever an horizontal slice of the workload can be allocated simultaneously. A slice requires storage space for at least one batch volume (which is used at the same level by all pipelines in the host) and one private volume for each pipeline plus an additional private volume so that at least one of the jobs can access both its input and output storage and allow the workload to make some progress. This strategy ensures that no batch refetch is necessary, since the entire horizontal slice will execute before the workload continues to the next level.

Minimal Strategy. The most constrained approach is the Minimal Strategy, where at least one job needs to have have access to its batch volume and its two private volumes (input and output). This strategy suffers from several problems, like the need to refetch batch volumes and the underutilization of CPU since only one job is executing. It is a similar approach to the AllBatch Strategy, with the difference that only one batch volume can be allocated and refetching needs to be performed.

2.2.5.3 Data Placement

In a data-intensive distributed system, a workflow planner must either take into consideration where data is located and send jobs to a site "close" to it, or it should include stage-in and stage-out steps in the workflow, in order to make sure that data arrives at the execution site before the computation starts. Both strategies are not mutually exclusive, of course.

There are some similarities between data-aware workflow planning and the way microprocessors plan the execution of instructions, where pipelining strategies are employed. In the latter, data retrieval from memory is usually the bottleneck due to slow memory access speed and the latency in the bus and CPU. When applying a pipeline strategy at the instruction level, memory access instructions are buffered and ordered in order to improve the overall throughput. The same strategy can be applied to a distributed workflow planner, where workloads can be viewed as a large pipeline and the bottleneck is the access to remote data due to network latency, bandwidth and communication overhead. Pipelining techniques are used to overlap different types of jobs and to execute them concurrently, yet maintaining the task sequence dependencies, which is what a workflow planner needs to achieve. By following the pipeline strategy, it can order and schedule data placement tasks in a distributed system independently of the computation tasks in order to exploit a greater level of parallelism and reduce CPU underutilization when waiting for data to arrive, while enforcing data dependencies.

2.2.6 Data-Driven Scheduling in the Proposed System Context

The target system of this work will have to deal with large amounts of data, potentially having to process gigabytes of video material. This large amount of data cannot be pushed aside to the background, it must be given priority as it will condition the execution of the system. This must lead the system's architecture to be data-oriented and not computational-oriented. Data tasks should be seen as the basic work unit and guide the computation according to the placement and availability of the data.

2.3 Face Identification

The ability to recognize and distinguish one face from thousands of other different faces has been imprinted on the human brain visual cortex for millennia, so that now people usually perform

this task subconsciously without realizing the difficult process around it, and what it takes for their brain to accomplish an identification with a very high precision. Nonetheless, however easy it seems for a human to recognize a face quite independently of the context, computers still struggle for high confidence intervals when presented with the same face in several different environments. During the last 40 years, the problem of face identification by computers has been the subject of extensive research, with several techniques being suggested which, sometimes, can match or exceed humans (O'Toole, Phillips, Jiang, Ayyad, Penard, & Abdi 2005). During this last decade, the great improvements in computing power have brought new possibilities, enabling techniques that 20 or 30 years ago were simply not feasible, CPU- and memory-wise.

In this section, the problem of face identification by computers will be described, starting with an overview and definition of the problem and goals (Section 2.3.1), followed by a description of how humans identify faces (Section 2.3.2). Next, the description of some existing methods to perform recognition (Section 2.3.3). Finally, some issues and approaches specifically related to identifying faces in videos are described (Section 2.3.4).

2.3.1 Defining Human Face Identification

Human face identification is a natural activity to the majority of humans, being an essential capability that has evolved and reached a high accuracy percentage. Despite being natural to humans, it is necessary to define clearly what facial identification means for a computer. It is part of a broader subject, which is known as *human face perception* in (Hongxun, Wen, Mingbao, & Lizhuang 2000). A general notion of this area is given, stating that it is a kind of intelligent behavior for computer to catch and deal with the information on human faces, including human face detection, human face tracking, human face pose detection, human face recognition, facial expression recognition, lipreading and others. Although face identification is a specific sub-area of human face perception, it partially encompasses several of the others areas presented.

Simply put, it tries to solve the problem of, given a still or video image, detect and classify an unknown number (if any) of faces using a database of known faces. It is worth noting that although it is relatively simple to clearly identify the problem and what has to be done to solve it, in (Turk & Pentland 1991) it is shown that developing a computational model for face recognition that efficiently and completely solves the problem is immensely difficult, as faces are a complex, multidimensional, and meaningful visual stimuli. Also, faces are a natural class of objects which are not easily mappable with the usual models utilized in other computer vision research areas.

2.3.2 Face Identification by Humans

One of the goals of automatic face identification systems is to reach and eventually surpass human performance on that same task, although they are not required to follow the same strategies

humans have developed over time. As a starting point, this section describes the strategies the human visual system uses to perform face identification, which might be useful when developing automatic systems.

Humans make use of a lot of sensory information to perform recognition (visual, auditory, olfactory, tactile, etc) and in some cases can even rely on the context to identify a person. Yet, most of this information is not easily available to computers, which are frequently only presented with 2D-images. However, computers have a clear advantage on the amount of information that can be stored and processed, even if they cannot use it so effectively. A person is typically able to store, or "remember", a limited set of faces in the order of thousands, while a computer system can potentially store an infinite number.

Some relevant topics regarding the way humans recognize faces are now presented, based on the works of (Chellappa, Wilson, & Sirohey 1995; Zhao, Chellappa, Phillips, & Rosenfeld 2003; Sinha, Balas, Ostrovsky, & Russell 2006).

2.3.2.1 Feature-based vs. Holistic Analysis

Humans are believed to rely heavily on two crucial methods when analyzing a face, *feature-based* and *holistic-based* analysis. These approaches are by no means mutually exclusive, so they are used together when identifying a face. Nevertheless, the question remains as to which of this approaches most influences a recognition.

A feature based approach is performed in a bottom-up fashion, where individual features (e.g. eyes, eyebrows, mouth, etc) are extracted from a face and assigned to a known person which has similar features. On the other hand, an holistic, or configural, approach implies the apprehension and classification of the face as a whole, using elements such as the shape or the global geometric relation between individual features to help in the recognition. Studies suggest the possibility that a global description like this can serve as a front end for a feature-based analysis, being at least as important as the latter. Yet, in the case that dominant features are present, such as big ears or a crooked nose, holistic descriptions may be skipped (Chellappa, Wilson, & Sirohey 1995).

A simple example of the importance of holistic analysis is when features on the top half of a face are combined with the bottom half of another face. What happens is that the two distinct identities are very difficult to recognize as whole. However, when the two halves are separated, presumably disrupting the holistic process, the individual identities of each half are easily recognizable. This also suggests that some features alone are sometimes sufficient for facial recognition (Sinha, Balas, Ostrovsky, & Russell 2006).

Another strong evidence of the importance of a holistic analysis is that an inverted face is much harder to recognize than an normal face. In (Bartlett & Searcy 1993) this is exemplified using the

"Thatcher Illusion", where the eyes and mouth of an expressing face are inverted. The result in the face with the normal orientation looks grotesque, but on the inverted version it appears quite normal and it takes some close inspection to notice the change.

Although feature processing is important for facial recognition, given these results it is suggested that facial recognition by the human visual system is dependent on holistic processes involving an interdependency between features and configural information (Sinha, Balas, Ostrovsky, & Russell 2006).

2.3.2.2 Importance of Facial Features

Even though all features of a human face can be used for identification, there are some who appear to be more significant to this task, without which it seems to be harder to identify someone. Among these most important features are included the face outline, eyes and mouth. The nose is usually not considered with much significance in frontal images, but it has a greater impact in a profile view of a face.

Another feature whose importance is typically undervalued is the eyebrow. It is typically a mean to convey emotions and other nonverbal signals, which the human visual system may be already biased to detect and interpret, and so it may be that this bias is extended to the task of facial identification. It is also a very "stable" feature, as they are hardly occluded as the eyes, tend to contrast with the surrounding skin and are a relatively large facial feature, surviving image degradations. Some interesting results on this subject show that the upper part of the face is more easily recognized than the lower part, presumably by the presence of the eyes and eyebrows (Sinha, Balas, Ostrovsky, & Russell 2006).

2.3.2.3 Face Identification as a Dedicated Brain Process

Humans are capable of identifying all kinds of objects from the world, but faces are believed to possess a dedicated process in the brain just to deal with them. Some arguments on that there is indeed a dedicated process are:

- Faces are more easily remembered in an upright orientation when compared to other objects. As stated before, an inverted face is harder to identify than a normal one, presumably by disrupting the holistic analysis process.
- People who suffer from prosopagnosia find it very difficult to recognize faces, even those they were familiar with, but typically they are able to recognize other objects without much difficulty. They can perceive the individual features, but are unable to see them together in order to identify the subject.

- Newborns are believed to be more attracted by faces and face-like patterns than other objects with no patterns or mixed features, suggesting a genetically predisposition to process faces in order to detect and identify parents or caregivers.

2.3.3 Automatic Face Recognition Methods

There has been extensive investigation on how to best recognize a face given a learning database. Researchers from the most different areas have shown interest on the face recognition problem, leading to a vast and differentiated literature and to the creation of several methods, each having some advantage over the others while loosing in some aspect. In order to achieve a high-level classification of these kind of methods, and since the investigation on this area tends to follow the way the human visual system works, the already described categorization on how humans recognize a face will be "borrowed", namely *feature-based matching methods* and *holistic-based matching methods*. Also, as some methods employ strategies from both approaches, a third category will be used to classify these hybrid systems, named *hybrid-based methods*. This classification is based on the work of (Zhao, Chellappa, Phillips, & Rosenfeld 2003).

2.3.3.1 Facial Feature-based Matching Methods

One of the proposed ways to distinguish between two faces is through the geometric relations among the several existing facial features, relying on a feature set which is very small when compared to the number of image pixels. The general idea is that by extracting and storing a set of known features from a face image in some useful representation, it is possible to compare this set with that of another image to see if they are sufficiently similar to belong to the same person. Thus, it is necessary that the chosen set of features to extract be sufficient for discrimination even at low-resolution images, so a careful study on the importance of each feature for discrimination must be done. This is typically applied in a supervised learning fashion, so that a learning database has its features extracted and any new face is considered against those extracted features. Only a general description of purely feature-based matching methods is given, whereas further sub-categorization can be found in Table B.1 of Appendix B.

A big advantage that feature-based methods have over holistic-based is their ability to adapt to variations in illumination, scale and rotation up to a certain degree, which was the central argument in favor of feature-based methods in (Cox & Yianilos 1996). An example of this is the normalization step in (Brunelli & Poggio 1993), which achieves scale and rotation invariance by setting the interocular distance and the direction of the eye-to-eye axis, and achieves robustness against illumination gradients by dividing each pixel by the average intensity. Yet, this type of methods are more likely to have a bad performance when some features are occluded or facial hair

and facial expressions are present (Etemad & Chellappa 1997).

An important step in these types of methods is the extraction of the features. Derived from the configuration properties of a face, constraints such as bilateral symmetry or the fact that every face has two eyes can be exploited in order to facilitate feature extraction. An early work on this subject was performed by Kanade (Kanade 1973), where an algorithm for automatic feature extraction is suggested. It starts by calculating a binary representation of the image and then runs a pipelined analysis program which will try to extract the position of individual features from the binary representation, like the top of the head or the eyes. From this analysis, a feature vector is obtained containing ratios of distances between features, which can then be used to compare between two faces.

The next step after the feature extraction is the recognition of the face, which is performed by building a selection model from the training data. It is important that the model does not adapt specifically to selecting on the training data, but that it generalizes to unseen patterns (Cox & Yianilos 1996). The classification of a face is performed by evaluating the distance between the feature vector from the face to be classified to all the vectors in the learning database, using a nearest neighbour approach or other classification rule (Bayes for example (Brunelli & Poggio 1993)). Different similarity matching methods can be used, but the most common is the Euclidean distance. Some examples of other methods are the mixture-distance (Cox & Yianilos 1996), cosine-distance, Mahalanobis-distance (Navarrete & Ruiz-del solar 2002) or weighted-mean absolute square distance (Etemad & Chellappa 1997).

2.3.3.2 Holistic-based Matching Methods

Another common approach to perform facial identification is to use the whole face region as an input to the analysis process and apply a classifier to the image information and not to facial features information. Unlike feature-based methods, which extract predefined features from the face resulting in small sized vectors, in a holistic-based approach the high-dimensionality of the data is one of the major problems, as every pixel in the image is considered as a separate dimension, bearing implications in computation efficiency when processing the data. Also, low-dimensional representations are highly desirable for large databases as they are usually smaller to store and transfer through the network (Etemad & Chellappa 1997). Thus, several methods for dimensionality reduction and data representation were proposed and applied in holistic-based systems. Some examples can be seen in Table B.1 in Appendix B.

Given that the system to be implemented for the dissertation project is intended to implement an holistic-based method using the eigenfaces approach of Turk and Pentland (Turk & Pentland 1991), more emphasis will be given to the description of this work, namely its utilization of Principal

Component Analysis (PCA) to achieve dimensionality reduction and the classification strategies used to perform identification.

Eigenfaces Method. This method categorizes facial recognition as a two-dimensional problem, partially ignoring the 3-D nature of the human face that would typically require a more detailed model for its analysis. Some constraints can be applied by taking advantage of the fact that faces are usually in its normal upright position and can be described with a small set of of 2-D characteristic views. As this is an holistic-based method, it is susceptible to variations in illumination, rotation and scale, as described above, so the goal is to develop a computational model which is fast, reasonably simple and accurate in constrained environments such as an office or a household. The foundation of the eigenfaces method lies in an information theory approach of coding and decoding face images, emphasizing the significant local and global "features" of the face. Although they share the same name, these "features", or characteristics, may not be directly related with the intuitive notion of facial features existing in feature-based methods. Similarly, yet distinctly, they represent the characteristics of the *image itself* which enable a better discrimination of the given face. Hence its susceptibility to variations on the background or the light. The general idea is that the relevant characteristics should be extracted from the image, encoded in some efficient format and compared to the encoded information of other faces in the database. To achieve this result, the authors proposed the use of an encoded representation technique developed by Sirovich and Kirby (Sirovich & Kirby 1987) based on PCA, which can economically represent an image set using a best coordinate system. In the image set, each pixel is considered a dimension by itself, so an image corresponds to one point in a high-dimensional coordinate system, which implies a large computation effort to process. By utilizing PCA, the number of dimensions in an image set can be reduced to a representation they called *eigenpictures*, while Turk and Pentland called them *eigenfaces*, and thus the name of the method.

A brief description of the mathematical reasoning behind an eigenface is provided for the sake of completeness, while referring to the work of Sirovich and Kirby (Sirovich & Kirby 1987) for a full explanation. In order to efficiently encode and process a distribution of faces, the most important characteristics in terms of data variation must be calculated. These are the components which will better represent the data and allow for the most accurate reconstruction. In practice, this resumes to finding the eigenvectors, or eigenfaces, of the covariance matrix of the set of images. Note that each image in the database can be reconstructed from a linear combination of the eigenvectors, thus an encoded representation of an image corresponds to the vector containing the coefficients of that linear combination, or, in the nomenclature of Turk and Pentland, the *weights* of each image. There can be as much eigenvectors as images in a data set of size M, but the goal of PCA is to choose a subset containing only the "best" M' eigenfaces, those with the higher eigenvalues and that

best represent the data variation. The number of eigenvectors to store is typically much smaller than the total number of images ($M' << M$) and can be set with an heuristic. Each eigenface represents a coordinate in the new much smaller coordinate system, spanning a subspace of the original coordinate system which the authors named *face space*.

After calculating the eigenfaces for an image set, the process of recognizing a new face can be resumed as: project the image onto each of the eigenfaces to calculate its weights; determine if there is indeed a face in the image by checking if the weights are sufficiently close to the face space; classify the weight pattern as a known or unknown person by comparing with the weights stored in the database according to some distance measure (Euclidean distance in this case); and optionally, include the new face in the database and retrain the system.

Apart from the task of identifying a face, the eigenfaces strategy can also be used to detect faces in images. Faces present much more inter-class than intra-class difference, meaning that it is easier to tell whether an object is a face than to differentiate between two faces. Knowing this, one can identify faces as the regions of the given images that, when projected, bear most similarity to the face space.

2.3.3.3 Hybrid Matching Methods

In a hybrid approach, the best of the two methods described above (holistic and feature-based) is conciliated to create a system that not only processes faces as a whole, but also takes facial features into consideration to create a more robust system. An example of a system in this category will be presented.

In an extension to the original eigenfaces system (Turk & Pentland 1991), Pentland et al. (Pentland, Moghaddam, & Starner 1994) developed a system that incorporates both holistic and feature-based methods. In this work it was pointed that the basic eigenfaces method is not prepared for real-world applications, as it was only tested on a database of a few hundred images under constrained conditions. The scalability of the system using just the normal eigenfaces strategy was tested on a large database with 3000 people, with more than one image for each individual and several different views. Nonetheless, the system still depends on the problems that affect holistic approaches (i.e. illumination, glasses, etc). The authors then extended the eigenface technique to the description and coding of facial features, yielding eigeneyes, eigennoses and eigenmouths. The evaluation showed that an hybrid stategy can outperform both holistic and feature-based strategies used exclusively.

In order to detect facial features, a similar approach to the whole face detection previously used (Turk & Pentland 1991) was applied, making use of a metric similar to the distance-from-face-space

to extract them. Refer to (Pentland, Moghaddam, & Starner 1994) for the complete description of the feature extraction.

Possessing both eigenfaces and eigenfeatures, a modular approach can be employed that makes use of both information. Several strategies were studied on the best way of merging the information, such as cumulative scores taking equal contributions by each facial feature; weighting schemes based on psychophysical data; and a sequential classifier, where eigenfaces are used to narrow the scope of database search to a region of the face space, followed by the use of eigenfeatures to perform the final classification (Pentland, Moghaddam, & Starner 1994).

2.3.4 Video Face Identification

So far, all the identified methods for facial recognition were developed with still images in mind. The next step in the evolution of face identification systems is to detect and identify faces in videos, which provide not only a whole new set of challenges, but also properties that still images do not possess and that can be useful if exploited (e.g. motion). Among the most relevant challenges there are the low-quality of the videos, either by outdoor captures or other bad conditions for video capture, and the small size of face regions when compared with still-images. Video face identification can be seen as an extension of still-image face identification, as the earliest attempts were focused on first detecting and segmenting faces from a video and then applying still-image techniques. Yet, in (Zhao, Chellappa, Phillips, & Rosenfeld 2003) it is claimed that true video-based face recognition uses both spatial and temporal information.

An improvement over the previous strategy is to add tracking capabilities to the system (Zhao, Chellappa, Phillips, & Rosenfeld 2003). A tracking-enabled system can analyze and estimate the motion of a face and recover some 3-D spacial information about it, such as rotations and translations. This enables the synthesization of a virtual frontal view from the various frames by estimating pose and depth, which can then be processed by a regular still-image based technique. Also, since there can be a high number of frames in a video, the recognition rate can be improved by applying a voting strategy, where the recognition results from each frame will vote for the most likely virtual face representation.

According to (Zhao, Chellappa, Phillips, & Rosenfeld 2003), the next step in this area is the implementation of multimodal systems, which make use of several information sources to reach an identification. The reasoning behind this concept is that humans make use of more information than the face, like voice or body motion, and the usage of multimodal cues can offer a solution which would not be achieved by using face images alone.

Finally, recent approaches in video-based face identification try to make use of one of the most important characteristics of video over still-images, the temporal semantics hidden in the video face

sequence. By exploiting this property and its relation with the spatial information considered in face tracking, a system can effectively capture the dynamics of pose changes and reduce occlusion effects. The research tendency is for the appearance of more systems based on this strategy. Some examples of systems in the three described categories can be found in Table B.2 in Appendix B.

Despite sharing some characteristics with still-images, videos usually add some other challenges to the problem. For example, scenes with fast movements or simply a night capture are prone to result in blurry videos, which are not ideal for to recognize as the edge outline is not clear. Also, the progress on video compression algorithms can difficult the processing of videos, effectively pushing the performance down and turning some algorithms unfeasible.

Summary

The three topics surveyed in this chapter are closely related to the type of system that will result from this work, and the description of the advantages and disadvantages of each can be of help in the design of the system. The first area of study is the Cloud Computing paradigm, a relatively young set of technologies that were merged and have been growing in popularity during the last decade to create cost and resource effective solutions that can adapt rapidly to the needs of its users. The development of new cloud based solutions that leverage this dynamic and large pool of resources is important, enabling new systems that were not possible before, such as human face recognition in large video databases. In the second area of study, a vision of systems from a data-oriented perspective is analyzed, opposing the more common computation-oriented perspective. In a system like the one being designed in this work, which is greatly dependent on the data flow and location, it is important to know where are the main constraints and the possible bottlenecks derived from the manipulation of big data. Finally, the last section surveys the face recognition area. Even though this work is focused on raw speedup of a face identification algorithm, it is important to know more about the subject, which are the key techniques used, how they can be modified to fit the new system and the most probable problems that can be met.

The next chapter will describe the system design process and the solution's architecture, leveraging the knowledge gathered in this chapter.

3 Solution Design

In this chapter, the system's design process and its architecture are described in detail, starting by analyzing the system's architectural drivers and what pre-existing conditions influenced the decisions made which affected the final architecture and the way the system operates. Next, the system's architecture is presented in the form of several diagrams, each representing a different view of the system and focusing on different aspects. Finally, integration process of the face recognition algorithm in the system is described, followed by the definition of the data model utilized.

3.1 Architectural Drivers

The main goal of this system was already mentioned in Chapter 1 - to speed up automatic face recognition in videos leveraging cloud computing resources. The proposed solution to this problem comes in the form of a software architecture, where it is important to clearly define what drives its design, what other factors must weigh the most when taking decisions to reach the goal, the constraints and the scope of the problem to be solved. These are the commonly called architectural drivers, which encompass the most important design principles, constraints and functional and non-functional requirements.

3.1.1 Main Principles

These principles constitute the main factors in the system design, acting as general goals of the system and limiting the number of choices if a decision must be made:

1. Performance - raw performance in terms of time vs. request size is critical in this system, so it should answer requests as fast as possible given the problem size and available resources.
2. Scalability - the system should scale gracefully without major performance loss in terms of number of resources utilized and number of videos to process.
3. Flexibility - the system should allow for different types of requests, with different quantities of resources in a seamless way to the end user.

3.1.2 Functional Requirements

These type of requirements state what the system should be able to do for an end-user, in terms of the features available:

1. The system should allow a user to submit a new video to be processed.
2. The system should allow for more than one user to submit a new video simultaneously.
3. The system should allow users to request information about processed videos or people.
4. The system should allow a user to provide an identification to an unknown person.

3.1.3 Non-functional Requirements

Finally, the non-functional requirements (or technical requirements) which impose some technical characteristics the system must have:

1. The system should support variations on the video input, such as size, resolution, aspect ratio, codecs, containers and framerate.
2. The system should be efficient in terms of exchanged messages since the network is a limited resource and videos take most of the available bandwidth. The components should not send message bursts with intervals of less than one second and should try to perform operations in batch if possible.
3. The system should allow for several videos to be processed simultaneously, and, if possible, without deterioration on the overall performance.
4. The system should allow the introduction of a new face recognition algorithm without modifying the remaining of the code.
5. The system should allow the usage of a different database and/or distributed file system with reduced need to modify the existing code.
6. The system should allow the usage of a different cloud computing provider without modifying the existing code.

3.2 Software Architecture

In the previous section, a description of the main architectural drivers was provided, which will reflect on the content of this section, where the whole system architecture will be described. This is one of the most important steps on the development of the system and should be clearly specified before implementing, as it will direct the rest of the work. Yet, during the development of the system the whole architecture evolved continually, on several situations, as more problems arose. The diagrams described here are the final result of this evolution and some of them are deeply linked with the implementation, due to critical issues regarding performance and scalability.

The section starts with a description of all the system's components, followed by a modular view of each one, enabling a high-level definition of the different code modules and their role in each component. After the modular view diagrams, a component view description of the whole system is given in order to identify communication channels and dependencies between components. Next, the

deployment scheme of the system in the cloud platform is defined, as well as the types of messages traded between components. Finally, it continues with a description of how the face recognition algorithm and its subcomponents are integrated in the rest of the system, and a high-level definition of the data model.

3.2.1 System Components Description

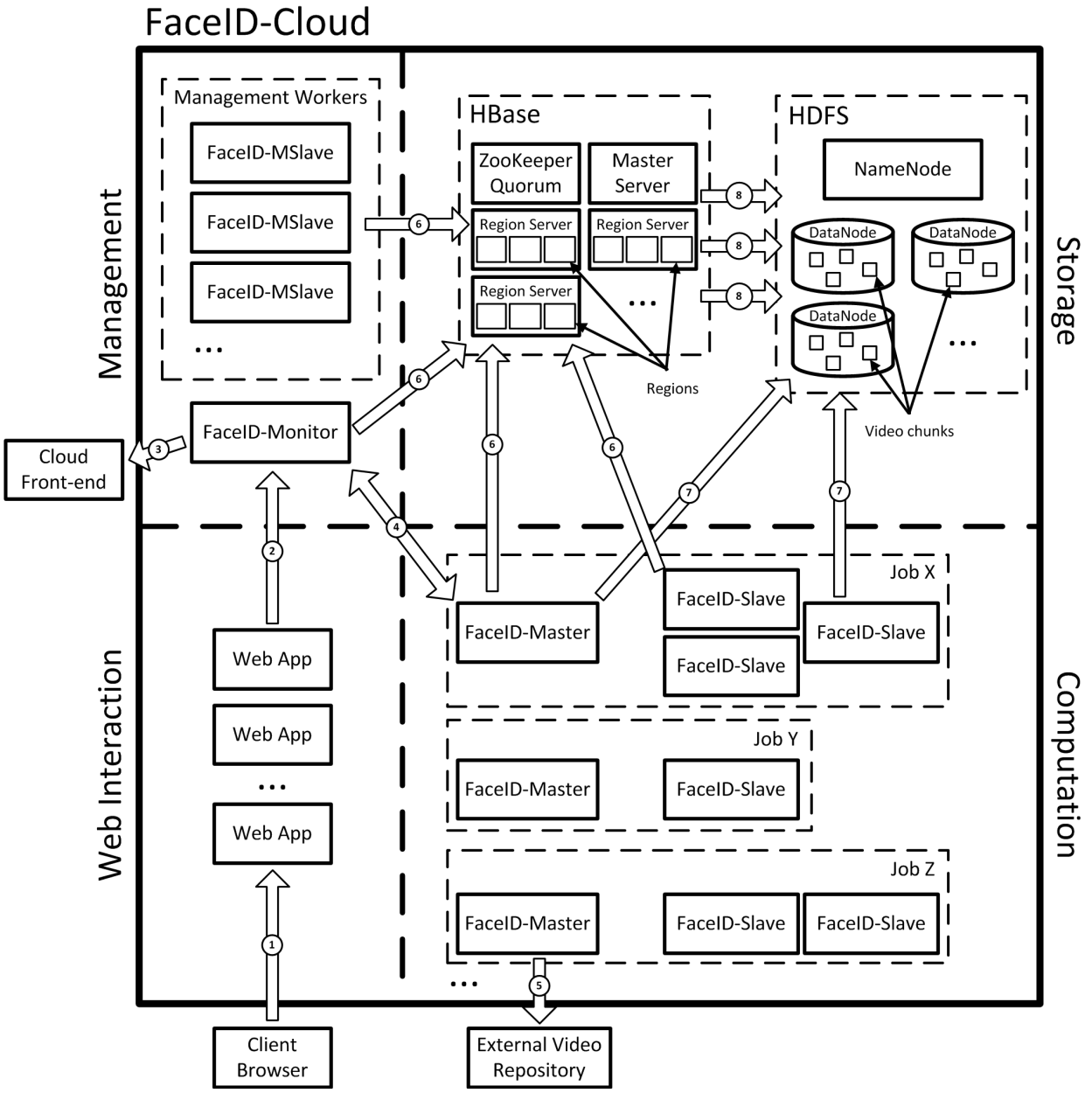


Figure 3.1: Runtime Components Diagram

Although the goal is in fact very simple, the system is still prone to some complexity, namely with the amount of components and interactions between them. To simplify the design and development, a logical division was made between the components in 4 major areas, each with assigned responsibilities: Management, Computation, Storage and Web Interaction. These areas can be seen

clearly in Fig. 3.1, containing the system components and their interactions which are explained in the following subsections.

3.2.1.1 Management

The Management area is primarily responsible for accepting and dispatching new requests and managing the system's resources by allocating and deallocating new nodes when needed. Besides this primary aspect, components in this area can also execute system-level tasks, such as cleaning database entries, or training the system according to the face recognition algorithm requirements.

There are two main components in this area, FaceID-Monitor and FaceID-MSlave. The former, dubbed simply as *monitor* for the rest of this document, is the brain behind the system and can be seen as the system scheduler. Its main functions include: accepting new requests and dispatching them as jobs to the components that will effectively process them; knowing the current state of all components; interacting directly with the cloud platform front-end to manage resource allocation/deallocation; and launching the training of the face recognition classifier.

The second component, dubbed as *mslave* for the rest of the document, is a special workforce to be used by the monitor as a worker node when it needs to perform tasks that require more computation. The rationale behind this separation is related to the general availability of the system, so that the monitor is mostly available to respond to new requests and is not occupied with heavy computation tasks. The main task performed by these components is essentially the training of the face recognition classifier, but can occasionally be used for other matters.

3.2.1.2 Storage

One of the most important areas in the system is data storage. As a data-oriented system, where it is the flow of data that most influences the computation and overall evolution, the way the system data is stored is crucial to achieve both a well-performing and scalable system. Due to the large average size of videos, even reaching the gigabytes mark, there will eventually exist network congestions, with all the bandwidth being occupied with large video transfers, so the system should try to minimize large transfers as much as possible.

There are two important types of data flowing through the system, which will be referred as structured and unstructured data, and each of them will have its own specific requirements.

Unstructured Data. With the main object of processing in the system being potentially large videos, it is important to define how the system will handle these entities and how to overcome the challenges they pose. Throughout the document, unstructured data will refer to videos and their direct products, such as smaller video chunks. In a distributed environment such as the cloud, access and management of fresh data is typically handled with strategies such as replication, concurrency control, caching and others, which usually offer a compromise between consistency, availability and

partition tolerance as stated by the CAP theorem (Brewer 2000), although performance may also be seen as an omnipresent part of the "game" being affected by all the others. The goal of this work is not to solve this specific type of problems, but all the properties are indeed necessary. Hence, an available solution that deals with the above mentioned problem will be used, in this case the Hadoop Distributed File System (HDFS)[1], which is part of the Hadoop framework and puts emphasis on high throughput of data rather than low latency. It supports very large data sets and has a simple coherence model of write-once-read-many which fits the needs of our system to store videos. Besides being able to store a large amount of data, the distribution of data among several nodes will increase the performance of the system as several clients can be connected concurrently to different HDFS nodes. To simplify the nomenclature, HDFS will be referred as datastore for the rest of this document.

Structured Data. Although the datastore can perfectly store all the needed files for the system to work, some kind of structured data store is needed (for management, coordination, and directory metadata), where typically a database system is used, either relational or not, and is capable of storing processed data and answer queries over that data. The choice for the database to use in this system is based on the type of access and the properties that matter the most, namely random real-time read/write access to the data and scalability to millions of rows/columns across a data centre. Relational databases have shown signs of not being able to handle horizontal scalability very well due to the inherent complexity needed for coherence and query capabilites, leading to the use of key/value databases for cloud based systems that support lookups by key and dispense the complex select operations from relational databases. For this system, we chose HBase, a column-oriented key/value database that runs on top of HDFS and sits upon its scalability and replication properties. One of the problems of using HBase is the lack of query and join functionalities, passing that responsibility to the client application.

3.2.1.3 Computation

This area comprises the real workforce of the system, the components that will do most of the heavy duty processing. It is also the most dynamic, since resources will be allocated and de-allocated as needed, leveraging the cloud platform's flexibility and dynamic nature. There are two main components in this area, FaceID-Master and FaceID-Slave (*master* and *slave* for the rest of the document), which can be instantiated by the monitor when more resources are needed to handle a new job. Their responsibility is to process incoming videos and extract information about the people which take part in them by applying the face recognition algorithm.

[1] http://hadoop.apache.org/docs/r1.0.4/hdfs_design.html

A master component is instantiated whenever there is no other free master that can handle a new job that arrived. All the responsibility of the job is transferred to the master by the monitor, so that it becomes free to answer other requests. The master is prepared to retrieve the video to be processed from an accessible online repository and stores it in the datastore already in the form of chunks. This process of fetching and splitting the videos might be one of the biggest time consumers in the system, but it is considered out of the scope of this work as it is affected by many different areas such as telecommunications or video codec technologies.

The problem with the fetching process is, of course, the network bandwidth available to download videos from remote locations. If it is anything less than a few megabits per second (Mbps), the download time of a regular video (considering 100MB) can delay the whole computation by a very large value. Also, if several masters are trying to download videos concurrently through the same connection, the network bandwidth to the exterior will be even more limited to each one (although the cloud platform may have several different physical lines that reduce this concurrency effect). These are conditions that must be considered in part of the design of the system, but which are not supposed to be dealt with directly. To simplify this problem, it is assumed that every remote video repository and the cloud platform offer a 1 gigabit per second connection (Gbps), so that the bandwidth is sufficient to transfer videos in reasonable times even if several masters are downloading videos concurrently. Also, to further decrease the overall transfer time, some kind of parallel download could be applied, where a master could possess several streams downloading a part of the video. Again, this idea is not explored in this work.

The splitting of videos is another problem that is very hard to solve and constitutes a big threat to the system's overall performance, if not the biggest. Today, most videos are stored and transferred under different powerful compression codecs and containers, which greatly reduce their sizes and save network bandwidth. Yet, the usage of these compression techniques complicates the process of splitting the videos, as they cannot be simply split in half like a text file. The process of splitting a video needs, at least, to decode the video, analyse every frame, divide into chunks, write new information for every chunk, and re-encode everything again. It is, of course, a hard computational task, not to mention the multitude of different codecs and containers existing. Some efforts have already been made to develop solutions to accelerate this process such as the system proposed in (Morais, Silva, Ferreira, & Veiga 2011), but to simplify the design an existing third-party solution will be used, since there are systems available that are optimized to perform this kind of tasks and can be of use in this system, such as the *ffmpeg* tool[2].

After the fetching and splitting processes, the master can ask the monitor for slaves to perform the job according to the size of the video, but the number of actual allocated slaves will be determined

[2]http://ffmpeg.org/

by the monitor according to the system load. In a normal state the master will get all the slaves it requests, but the number may be lower if the monitor cannot dispense more.

Finally, the master is also responsible for distributing the load evenly among the slaves it controls and build a final list of people from the individual results of each slave. The load distribution process is simple, as the master just needs to indicate to each slave the location of their assigned chunks in the datastore.

The other type of component in the computation area is the slave, which is the one responsible for processing the videos and applying the face recognition algorithm. Slaves are instantiated on-demand when a master requests them from the monitor for a new job to which they become assigned, although they can be directly transferred from a job that just ended to a new one and avoid a de-allocation, reducing turnaround time and avoid slowdown due to constant virtual machine booting.

3.2.1.4 Web Interaction

This is mainly a support area to the system, enabling users to access the face recognition service provided through a web interface. The main component in this area is, of course, the Web Server, which acts like a middle-man between users and the system, simply contacting the system monitor when submitting new jobs and displaying the results to the user.

3.2.2 Jobs and Workflow

Throughout the previous section, the term "job" has been used freely without a formal definition, in the sense that it was some kind of workload activity related to a new request to analyze a video, but a more strict definition must be made. In this system, it is considered as a job the process initiated by the monitor with the goal of finding the identity of the people featured in a video indicated by a user. To process a new job, the monitor attributes it to a single master, so it is not a dividable work unit at master-level and cannot be split among several masters. A master can, nonetheless, subdivide its assigned job into smaller work units at slave-level, which are called tasks, in order to be able to distribute the load among the slaves under its command. Tasks are the smallest work units and cannot be subdivided, and each one corresponds to the processing of a video chunk in the datastore. Although a master can only have one assigned job, a slave can have several tasks attributed to it, a factor that depends on the number of slaves the monitor could dispense for the job and the size of the video.

To illustrate the evolution of a job in the system, a description of a typical workflow is provided, where a user requests a video to be analyzed through the web interface. Each step is related to one or more interactions displayed in Fig. 3.1, with the indication of the interaction number at each step (interactions within dash-lined groups are not shown in the figure).

1. The user contacts the web server component providing a URL for the video he/she wants to be analyzed by the system in search for human faces (1);
2. The web server contacts the system monitor and submits the information the user has provided, waiting for an asynchronous response (2);
3. The monitor starts by checking if the video requested was already processed. If so, the response is sent immediately to the web server, a job is not officially created and the interaction with the client terminates (6, 2, 1).
4. If the video is not known by the system, the monitor starts by checking its state for masters waiting to be de-allocated which have finished their previous jobs. If such a master exists, the new job will be assigned to it. If not, a new master is allocated through the cloud front-end (3);
5. The job information is sent to the chosen master, that will download the video from the URL provided by the user (4);
6. The master pre-processes the downloaded video by performing a conversion to a common format and splits it in several smaller chunks, which are uploaded to the datastore (7);
7. The master estimates how many slaves it needs to process the chunks and requests them to the monitor (4);
8. The monitor analyzes the system load and the resources available and sends the slaves it can dispense to the master, allocating more slaves if needed (3, 4);
9. Upon receiving the slaves' information, the master distributes the tasks among them and waits for termination by polling the database (6);
10. Each slave will execute their assigned tasks by applying the face recognition algorithm in the chunks. When each task is finished, the results are written to the database (7, 6);
11. When all tasks complete, the master officially finishes the job and sends a response to the monitor. All information is stored in the database for future requests on that video or the people contained in it (6, 4);
12. The results are propagated to the web server that presents them to the user (2, 1);
13. Finally, the monitor orders an m-slave to learn about the new faces found and integrate that knowledge into the database (6).

3.2.3 System Modules Diagram

The next step in the design of the system's architecture, and one of the most important for the rest of the work, is the decomposition of the system in modules, blocks of software that provide a specific functionality to the rest of the system. Modules are hierarchical structures, so that each

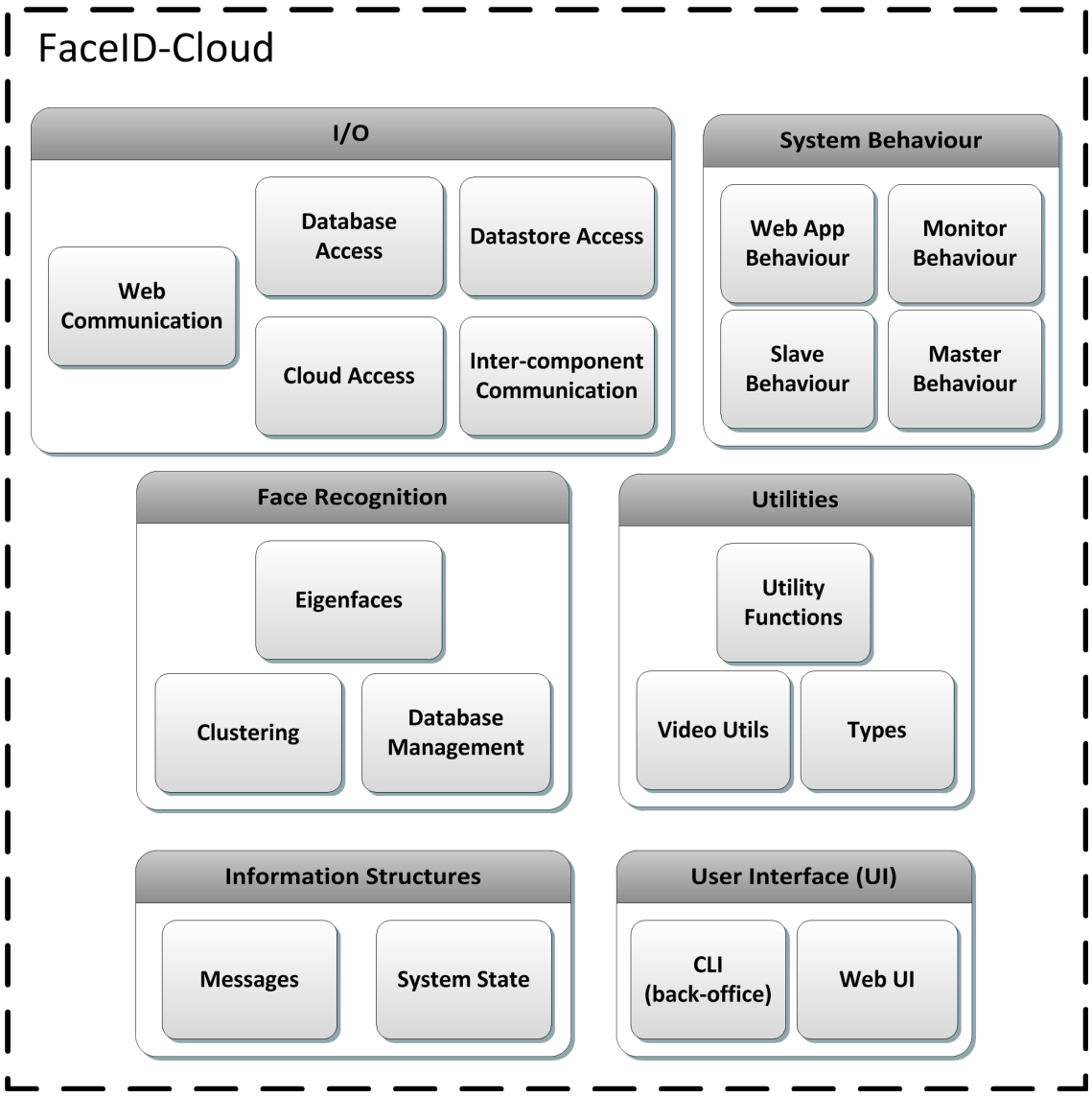


Figure 3.2: Modules Diagram

module can have sub-modules that constitute it, and so on. Also, modules can have relationships among them, where one module needs to use another one to correctly fulfil its duties. A 2-level decomposition diagram is presented in Fig.3.2, illustrating the global modular structure of the system. Each module will now be described in detail.

3.2.3.1 System Behaviour Module

This module is responsible for defining how the four main system components (monitor, master, slave and web app) behave and interact with their peers to perform their jobs:

1. Web App Behaviour: knows what information is presented to the user and the features the system provides, such as a new request to process a video or a search for a specific person;
2. Monitor Behavior: knows how the monitor holds and manipulates the system state and in-memory structures, and the management of the available resources;

3. Master Behavior: knows how a master handles videos and the coordination of the slaves it controls;
4. Slave Behavior: knows how a slave handles video chunks and applies a face recognition algorithm to them;

3.2.3.2 Face Recognition Module

This module provides an implementation of a face recognition algorithm by defining a common interface that can be called by other modules without knowing which algorithm they are indeed using. Under this common interface other sub-modules implementing different face recognition algorithms can be added:

1. Eigenfaces: knows how to apply the eigenfaces algorithm to a video chunk;
2. Clustering: knows how to perform clustering in a set of weight patterns;
3. Database Management: knows how to store and retrieve face recognition data from the database.

3.2.3.3 I/O Module

This module contains all the necessary mechanisms for components to communicate with each other and the outside world, encapsulating and abstracting the protocols and methods used with common interfaces:

1. Web Communication: knows how to handle communications with the end user through the web server;
2. Database Access: knows how to operate the database;
3. Datastore Access: knows how operate to the datastore;
4. Cloud Access: knows how to access the cloud platform to obtain info and allocate and deallocate resources;
5. Inter-component Communication: knows how to contact other components inside the FaceID-Cloud scope (monitor, master, slave and web app).

3.2.3.4 Utilities Module

Some functionalities are used by most modules and provide some operations that are not specifically related to any of the existing modules. This led to the creation of a separate support module for these utilities, acting like a store for generic operations like calculating some math result or byte-level manipulations:

1. Video Utils: collection of functions that perform video manipulation;
2. Support Types: collection of generic types that facilitate some of the other modules operations;
3. Utility Functions: collection of generic functions that facilitate some of the other modules operations.

3.2.3.5 Information Structures

This module defines a common format for keeping and transferring the system's information by the runtime components:

1. Messages: Definition of the format of messages to be exchanged between the system components;
2. System State: Definition of structures to hold system state, mainly by the system monitor.

3.2.3.6 User Interface Module

The interaction with humans, both administrators and end-users, is handled by this module:

1. CLI (Command-Line Interface): module that enables an administrator to control different aspects of the system through a textual interface;
2. Web UI: module that provides a graphical web interface for end-users to interact with the system.

3.2.4 Runtime Components Diagram

After having presented some views of the system's architecture that demonstrate its organization in terms of modules of software and their relationship among themselves, a logical description in terms of the runtime logical components will now be provided, illustrated by Fig. 3.1. It is now possible to identify the role of most of the concepts discussed in the previous sections in the whole system, namely the four areas of division of the system, the components responsible for handling jobs and the direction of information flow, in which it is defined that the direction of the arrows implicates a client-server request (server is always in the pointy end).

In Fig. 3.1 the four main components are presented as single identities, which is not necessarily true at a lower abstraction level. For example, the master component makes use of several threads to split a video and upload the chunks, which compete for some of the shared resources. The definition of the threads running on each component is important to analyze the impact of concurrency when accessing shared resources in the component scope (e.g. network). Across this description of the threads, the term "main components" is used to designate monitor, master and slave.

Server Thread. The three main components have a specialized thread to receive requests or instructions, the server/reception thread, where the communication channel is based on message passing. This thread does not contain any domain logic, acting as a proxy that receives and distributes the messages to the appropriate recipients of the component.

Message Processor Thread. To complement the server thread that receives the messages, a separate thread is responsible for routing those incoming messages. Each of the main components has a specialized version of this thread tuned to their necessities, defining the actions to take when specific messages enter the system. The separation between the message reception and routing is due to this individualized tuning on the latter, so that the server thread can be the same for all main components. Reception and routing could be merged together, but this design guarantees that when a component needs to change its routing strategy (the reception thread is not likely to change), the others will not be affected.

Resource Allocation Thread - Monitor specific. There is a specialized thread type to handle resource requests from masters that mediates the access to the resource manager in the monitor and answers their requests. Each request of this type will have its own resource allocation thread which will work at a lower abstraction layer level than the Server and Message Processor threads, which only care for directing messages.

Task Handler Thread - Slave specific. Slave components are responsible for processing video chunks, which are regarded in the system as tasks. To handle the tasks that flow from the job master, slaves will have a specialized thread to handle them, with each task being processed in a separate thread. This will enable slaves to process multiple chunks at the same time, something that is useful if the virtual machine instance configuration provides more than one virtual processor to each slave. The level of parallel work that can be done is, nonetheless, limited by the shared access to the database and network.

Classification Thread - Eigenfaces specific. During the recognition process performed on the slaves, there is a division between the detection of faces in all video frames and the classification of these faces. As videos can be very large and contain several thousand faces, it is not feasible to first detect all faces and only afterwards start the main recognition process, especially if the process is to be done only in the (limited) main memory - storing detected faces on disk is not an option as it would endorse severe performance losses when classifying. It is important to adopt a kind of pipeline strategy that enables detection and recognition to co-exist at the same time. To reach this effect, a separate thread must exist, in this case the Classification Thread. While detection is performed in the Task Handler thread that initiated the process, recognition will be in this separate thread, catching the faces as they are detected and processing them. One problem that must be taken into account is the different speeds at which both threads operate, with detection being much faster than classification due mainly to the extra database accesses. The strategy chosen is to limit the number of faces that can be generated by the detection thread so that the classification thread has sufficient time to process the faces in its queue. This will certainly limit the performance of the detection thread, which could continue doing its work and generating more faces, but as the process

is to be done in main memory such a limit must be imposed, or else the unclassified faces might get the component out of memory.

Database Request Handler Thread. Database accesses are costly, which means they must be well defined and strategies to reduce its costs must be applied. Among other strategies, one possibility is to perform several operations in the same database access where possible, as a batch. This effectively reduces the number of trips to the database and the overall time. Of course, this will implicate larger messages being transmitted, so a limit must be imposed on the amount of requests that may be aggregated on a single message so that there is less probability for errors on the transmission. Also, only non-urgent database updates will be performed in batch, typically to store new information such as faces or results, which are operations that do not need to be performed in any order and under time constraints. Other update operations that need immediate confirmation will still be performed as soon as possible, along with read requests, which are never performed in batch. To apply this strategy, the Database Request Handler thread will collect all database accesses and will try to optimize database accesses according to it.

Job Handler Thread - Master specific. A master's main function is to coordinate a job created by the monitor and return the results, hence the need for a thread specialized in handling jobs. This thread will do most of the job processing, like downloading the videos or controlling slaves, so it can be seen as the main thread of a master component.

Chunk Upload Thread - Master specific. To help the Job Handler thread storing video chunks into the datastore, there is the Chunk Upload thread, whose only function is to upload the chunk and register its addition in the database. Much like the Classification thread, this thread will consume the chunks coming from the main thread, Job Handler, and upload them, pipelining the process and accelerating it.

3.3 Face Recognition Integration

One of the crucial aspects of the system architecture is the integration of the face recognition algorithm, where there are two main points of focus. The first is to define the operations to be performed and their division over the workflow steps of a task. For example, on the eigenfaces approach to be used there is a clear distinction between detection and recognition of faces, so there should be at least one different operation for each one that could be called at different times during a task. The second point of focus is on trying to reduce dependencies between the rest of the system and the face recognition module, so that it can be used simply by invoking a known interface and enabling other different face recognition approaches to be plugged in with seamless integration.

In the next subsections, a description of the face identification module and its points of contact with the main system will be given, starting with the algorithm definition and its points of division,

and some strategies used to improve its performance in this particular system architecture, namely over database accesses.

3.3.1 Algorithm Definition and Integration

To perform face recognition in the system, the original eigenfaces algorithm by Turk and Pentland (Turk & Pentland 1991) was chosen, as it is a simple solution to implement with relatively good accuracy rate in controlled conditions (max at 96% correct classification averaged over lighting variation). Please refer to Section 2.3.3.2 on Chapter 2 for more details on the algorithm. According to the authors, the algorithm can be summarized in five steps:

1. Initialization: Acquire the training set of face images and calculate the eigenfaces, which define the face space;
2. When a new face image is encountered, calculate a set of weights based on the input image and the M eigenfaces, where M is a chosen value, by projecting the input image onto each of the eigenfaces;
3. Determine if the image is a face at all (whether known or unknown) by checking to see if the image is sufficiently close to the face space;
4. If it is a face, classify the weight pattern as either a known person or as unknown;
5. If the same unknown face is seen several times, calculate its characteristic weight pattern and incorporate into the known faces.

These steps describe and separate the original algorithm in its logical workflow. Yet, in this specific implementation, the capability to process videos and integrate with the rest of the system is needed, along with a more prominent definition and division between the training and the recognition phases, which could be performed by different components. This results in the following list of simplified steps:

Training - *generate face space*

1. Choose and obtain a subsample of all the faces to participate in the training;
2. Calculate the eigenfaces and eigenvalues which generate the face space to be used by the system;
3. Update the system to use the new face space.

Recognition

1. Acquire the video to process and obtain every frame;
2. For each frame from 1., detect all the faces that appear and store them;
3. Calculate a set of weights from the faces found in the frames, by projecting the images onto the face space, and store them in the database;
4. Classify the weight pattern as either a known person or as unknown by comparing with previously stored weights in the database.

Training - *learn about new people*

1. Group ‘‘close’’ weights from a new video to represent new individuals;
2. Cross the groups created in the previous step with the groups already in the database and search for similarities;
3. Merge similar/close groups into one single group, as they should belong to the same individual;

This division between training and recognition is essential, along with the division between the two training steps. The system should generate a new face space sporadically (in a normal scenario), while the recognition and learning of new faces will be performed much more often. Also, the recognition steps were simplified, namely by removing references to specific strategies to improve performance that will be discussed at the end of this section, such as the grid strategy (see Section 3.3.3), client caching (see Section 3.3.4) and clustering (see Section 3.3.5). A detailed description of the training and recognition processes is now given, leaving some implementation details to be explained in the Implementation (see Chapter 4).

3.3.2 Training I - *face space*

Training the system is the starting point for the eigenfaces algorithm to work, since it belongs to the supervised learning systems class, meaning that it needs a base set of faces for comparing and classifying new faces against. To start the training process, the system needs a representative set of faces to calculate the eigenfaces and eigenvalues that generate the face space to be used when classifying. The main purpose of building a face space out of the sampled faces is to reduce the dimensionality of face objects (and hence the computational effort) by focusing on the features that best describe the face set, which are not necessarily the regular features of a human face (i.e eyes or eyebrows) but, as an holistic algorithm, features that describe the greatest directions of variation of the whole face image.

To achieve a good representation of the features of the entire face set (including faces which are not used for training), the faces used must not belong to a small set of individuals, or, in the case of our system, should not belong to the same video or videos containing the same people. This would lead to a system specialized in recognizing a particular subset of human faces, as the training would focus on very similar and undifferentiated faces, while it would find it much more difficult to recognize a very different set of faces found in another video. For example, the mean type of face of an asian person is observably different from an african or a caucasian in terms of eyes, nose, eyebrows and even hair style, even if not at the same degree of difference between a human and another species (e.g. chimpanzees). Training with solely one of these types of faces would result in a face space that would not acknowledge the importance of the particular facial features of other

types of faces, and hence it would either not be able to differentiate between two different faces or would not assume two faces belong to the same person.

The main problem in achieving a well trained system is, then, how to gather sufficient and differentiated information from the face dataset so that the system is capable of differentiating and acknowledging the principal features of most face types existing in the world, or at least the types that are set to be recognized more often by the system. This problem can be summarized as finding the objects, in this particular case the faces, which best represent a dataset, almost like performing some kind of meta-Principal Component Analysis on the entire dataset before applying it to the face objects themselves. It is not a trivial problem and involves researching in areas which are out of the scope of this work, and so a simplified solution is provided where the faces which are used for training are gathered with a simple sub-sampling function. This function picks faces by their IDs in the system at regular intervals, allowing for different videos to be sampled since new face IDs are added sequentially as new videos enter the system. The results of this type of training are shown and analyzed on the Evaluation (see Chapter 5).

3.3.3 Grid Strategy

The face space is a very sparse multi-dimensional space resulting from the dimensionality reduction strategy applied in the eigenfaces algorithm using PCA. Nonetheless, even though the number of dimensions is greatly reduced, a low number of eigenfaces can still generate a very large number of vector combinations in this space. Also, although the faces of one individual might be very similar, in this new coordinate system they can still be very far apart on some dimensions, hence the sparseness of the face space. This brings some new problems that must be discussed, the first being: in a database with millions of faces, which ones should the algorithm use to compare the new face to (assuming an Euclidean-distance based classification)? It is very hard to compare the new face (its weight pattern in reality) with every single face available on the database, either in terms of time, memory or network bandwidth. Hence, a strategy must be applied in order to reduce the search scope for the classifier.

In this specific system, the database choice plays a great role on the choice of the strategy to use to reduce the search scope, namely its query capabilities. Despite being directed towards building scalable data centers with simple data models, HBase does not offer many of the said capabilities, meaning that some query logic must go into the application design. The solution found to address the search scope problem in this particular database type is to use a *grid strategy*, which is integrated both on the way the system performs searches and the way the data is stored in the database.

The grid strategy applied involves partitioning sets of weights into cells according to their vector coordinates, by aligning each coordinate to their next hundred, either towards $+\infty$ for positive numbers or $-\infty$ for negative numbers (zero is considered positive). In a sense, a new ‘‘grid coordinate

system'' is being built, where each weight can be described by its corresponding grid coordinates. For example, take the weight vector

$W = [-142.65, 1053.93, 0.00, 300.00]$

To translate it to grid coordinates, each number is "rounded" to the next hundred

$W' = [-142.65 \rightarrow -200, 1053.93 \rightarrow 1100, 0.00 \rightarrow 100, 300.00 \rightarrow 300]$

Resulting on the point vector W' on the new multi-dimensional grid

$W' = [-200, 1100, 100, 300]$

With all the weight vectors described in this format, the application can ask for the specific coordinate intervals that most resemble the actual unknown face's weight vector to be classified. For example, if W was a new unclassified face, W' would be used to generate a search range to be requested from the database. If we consider a margin of 200, the search parameters would be

$W_{top} = [(-200 + 200), (1100 + 200), (100 + 200), (300 + 200)] = [100, 1300, 300, 500]$
$W_{base} = [(100 - 200), (1300 - 200, (300 - 200), (500 - 200)] = [-400, 900, -100, 100]$
meaning that all vector in the database whose coordinates are between those ranges will be retrieved. Please do notice that zero is considered positive, so any time a coordinate in a range calculation results in zero, it is rounded to the next hundred just like a positive number. In the example, from W' to Wtop the first coordinate goes from -200 to 100 for this reason. One thing that is worth considering is the case where the database would be relational and not key/value based. In this situation it would be much easier to design a grid strategy due to the query join capabilities of this systems, since it would only be necessary to store each vector coordinate in a different table field and state a query such as

SELECT vector
FROM vectorsTable
WHERE (C_1<X_1) AND (C_1>Y_1) AND (C_2<X_2)
AND (C_2>Y_2) ... (C_n<X_n) AND (C_n>Y_n)

and all the neighbour faces would be retrieved. Yet, a key/value database is essential to the scalability of the system, so the strategy should be adapted to it. This grid strategy enables a more efficient search, since the focus is on the area with supposedly more probability of finding a similar face. However, there is still some chance of losing a valid neighbor face that is outside this limited search scope. There is no trivial solution to this problem if this grid strategy is to be applied, so the best option should be to choose a margin for the search range that includes most, but probably not all, of the faces that are closer to the test face than the minimum distance chosen to classify as a match. This strategy will be further discussed in the Implementation (see Chapter 4 Section 4.5).

3.3.4 Caching

Even though the grid strategy applied greatly reduces the search scope and the number of data to be transferred from the database, there is still the problem of having to fetch the same rows several times. For example, the faces of the same individual in two consequent frames will most certainly be close to one another in the grid, meaning that they will share the same neighbors. Yet, when the classifier is analyzing both faces in sequence, the neighbors of each of them, which will be very similar, will be fetched from the database. This highly suggests the usage of client caching, i.e. storing the results from previous searches in the database client's memory for future re-use so that a new database queries can find what they are looking for faster.

3.3.5 Training II - *Clustering*

In the original eigenfaces algorithm, there was a property of the training data that was assumed for it to work, which is not always true with the data to be processed by this system. That property is the given identity of a set of faces, the fact that the algorithm is trained to recognize a set of faces as a single person. For example, the training input to an implementation of the eigenfaces algorithm could be a collection of folders which are named according to the person they represent, containing inside a set of images with faces from the person in different orientations and expressions. On the other hand, the input to this work's system is a constant flow of unknown faces with nothing in common from the start. It is a major drawback, as it is necessary to partition the faces into different persons so that they can be given a common identifier, even if it is not the person's real identity. Given the nature of the data and the expected result, one approach could be to turn the eigenfaces algorithm's supervised learning nature into a semi-unsupervised one, where faces from an unknown person would be joined together, even if they do not belong to a known person. The usage of the prefix ''semi'' comes from the fact that the core of the eigenfaces algorithm cannot be changed, as the database still has to be trained in order to recognize faces, but an unsupervised approach is necessary to pre-process the data and separate the faces from different persons.

One of the most successful unsupervised learning approaches is clustering, with several strategies proposed over the years such as the k-means or the expectation-maximization algorithms, to cite just a few. Yet, a common trait of most clustering algorithms is the necessity to provide an initial number of clusters to be returned, or at least the dimension of the clusters (Han, Kamber, & Pei 2006). This constitutes a limitation on the usage of clustering in the system, as it is very hard, if not impossible, to determine beforehand the number of persons in a video (assuming a cluster of faces represents a person). Nonetheless, there are some clustering algorithms that do not need an initial seed, and can discover by themselves an acceptable number of different clusters, the DBSCAN algorithm being one of them (Ester, Kriegel, Sander, & Xu 1996).

The rationale behind the DBSCAN algorithm, short for Density-Based Spatial Clustering of Applications with Noise, is that there are different areas in a dataset with different densities of points. Taking the face recognition problem as an example, the faces from the same individual would constitute a high-density area, as most of the faces will be very similar and close to one another in space, whereas the faces that are doubtful and can be either one person or the other are in low-density areas. DBSCAN considers that there are two types of points in a dataset, core points and noise points. Core points are the ones responsible for keeping a cluster together, the ones that have sufficient neighbors to constitute a cluster. Noise points are the ones that are not near any core points, and so are not considered as part of any cluster. A short description of the algorithm's steps is provided, but please refer to the original paper for more detailed information (Ester, Kriegel, Sander, & Xu 1996).

In short terms, the algorithm commences by analyzing the first starting point in the dataset and checking how many neighbors it has according to an epsilon distance measure. If the point has more than a preset minimum number of neighbor points, it is called a core point and a new cluster is born. From here, all of the first point's neighbors are checked for their own neighbors to determine if some of them are also core points. This process continues recursively until the "edge" of the cluster is reached, i.e. the points being checked do not possess a sufficient number of neighbors. All of the points analyzed are classified as belonging to the same cluster and then the algorithm restarts with another unvisited starting point, until all points in the database are visited. In the case one of the starting points does not have enough neighbors, it is considered as noise and is not assigned to any cluster.

To set a value for the minimum number of neighbors that a point must have to be considered a core point, a notion related to the specific case of tracking faces in videos is used. A face in a video is usually in-between two other faces, the ones in the previous and next frames, with the exception of the first and last faces in a sequence. With this idea in mind, it is safe to assume that a value of 2 should be sufficient to provide good clustering results. Of course a face may have more than two neighbors if, for example, a person does not move for a while and all its faces will be very similar.

A reasonable value for epsilon, the minimum distance between faces for them to be considered as in the same cluster, is not so easily reached. It depends a lot on the video characteristics (e.g framerate) or the conditions of the environment where the people in the video were filmed, suffering from the same problems as the eigenfaces algorithm which also has to consider a minimum distance for a face to be recognized. To simplify, this distance threshold will be considered the same for both the clustering and eigenfaces algorithms. The specific value to be used cannot be calculated or defined without experimenting, so it should be discussed in the Implementation and Evaluation Chapters.

3.4 High-level Data Model

A system's detailed data model is closely related with the choice of database type, its specific properties and must evolve around the system's implementation and its needs. In this section, a high-level view of the data model is presented, which takes into account the entities needed after knowing the system's components, their interactions and the integration with the face recognition algorithm, and the necessary relations between each of them. Each entity will now be described succinctly, reserving a detailed discussion for the Implementation (see Chapter 4).

1. **Jobs** - Holds information about current and past jobs, namely the unique ID of each job, the resources allocated, job state and the results.
2. **Videos** - Holds information about the videos the system has processed, such as their names, URLs from where they were downloaded, the unique video ID attributed to each and video chunks location in the datastore.
3. **People** - Holds information about individuals that were added to the system, specifically their name, a unique person ID and the clusters to which the person is related.
4. **Faces** - Holds every distinct face found by the system, being one of the largest entities.
5. **Eigenfaces Data** - Holds information about the face recognition algorithm, namely the training information and results for each training session.
6. **Grid** - Holds the grid structure data, with information about each cell and their containing faces.
7. **Clusters** - Holds information about all clusters found, namely the mapping a cluster and the grid cells that belong to it.

Summary

In this chapter, a thorough description of the solution's design process is given, which will condition the rest of the work and the results obtained. It starts with the enumeration of the architectural driver's of the system - performance, scalability and flexibility, the functional and non-functional requirements which it must respect and the properties it must display. Next, the main part of the solution is presented - the software architecture - under the form of several diagrams that focus on different areas of the system at both code modules and runtime components levels; and the description of the components which are part of it divided into four major areas: Management, Computation, Storage and Web Interaction. It continues with the integration of the face recognition algorithm with the rest of the system and some specific strategies developed to deal with some of the problems encountered, namely a grid strategy for efficient querying, client caching to accelerate some searches and face clustering to integrate unknown people in the database. Finally, a high-level view of the system's data model is provided with the description of all established tables, leaving some details to the Implementation Chapter.

4 Implementation Details

The second part of this work derives directly from the design of the system, which is to build a tangible version of the concepts defined that can be tested and evaluated to assert the proper functioning of the system. This chapter presents a concretization of the design concepts defined in Chapter 3, by describing the technologies used and a more detailed software view addressing the problem of generating IDs and a detailed description of the data model based on the general view previously designed.

4.1 Technologies

A system is very much dependent on the available technologies at the moment of creation, with some system designs being put away while technology cannot yet handle it. For example, distributed computing technologies brought a massive amount of computation power to investigators which could not be achieved with single machines. Technology takes a large part on the definition of the system capabilities, of what it can and cannot do. It is important to define the technologies the system will use and what is the added value to be expected from this utilization.

4.1.1 Programming Environment

The main system components and all needed functionality is developed in the Java programming language, as both the chosen cloud platform, the datastore and database have native an Application Programming Interface (API) in this language. Java code is usually not compiled for native machine code, so it needs a virtual machine to be executed which can arguably add some performance losses. In practice, the Java virtual machines have evolved to a point where the optimizations they perform do provide a good performance comparable to C and other native-compiled. languages.

4.1.2 Cloud Computing platform

One of the goals of this work is to integrate a face recognition algorithm with cloud computing, an emerging paradigm over the last decade and that is now experiencing an explosion of platform solutions, both commercial and open-source. With so many options available, an informed choice is essential. Through the analysis performed on Chapter 2, a global view of the most widely known solutions was considered, so that the choice of platform to use in the system was based essentially on following items: low cost, easily modifiable/tunable, offering an Infrastructure-as-a-Service model and private-cloud deployment with possibility to expand to public. With this set of characteristics

the offers were naturally narrowed to open-source solutions, with at least three platforms fulfilling the requirements: Eucalyptus, OpenNebula and OpenStack. Either one of these solutions would perfectly fit the system, but the choice was made to use OpenNebula as the cloud platform, for several reasons:

- Mature technology, with 7 years of active development;
- Highly customizable;
- Can use any distributed file system;
- Compatible with Amazon's EC2 Query API;
- Large support community, good documentation and strong research partners.

OpenNebula is oriented to private-cloud deployment, enabling organizations to make the best use of their available resources by virtualizing their infrastructure. It relies on a rather centralized management architecture, with a single node monitoring the others. Also, it is highly customizable in that a large part of the system components can be tuned to adapt the platform to an organization"s needs. It provides several types of interfaces (XML-RPC, OCCI, EC2), but in FaceID-Cloud the XML-RPC API is used by the monitor to interact with the cloud frontend. There are three main parts of OpenNebula that affect our system and must be recognized: VM scheduler (*onevm*), image repository (*oneimage*) and virtual network manager (*onevnet*):

- Virtual Machine (VM) scheduler: the main daemon that controls the deployment of virtual machine instances in the infrastructure. The monitor interacts solely with this component to request resources.
- Image repository: although no system component interacts directly with it, it has pre-configured images of every component, ready to be instantiated.
- Virtual network manager: a limitation on the number of virtual instances, specifically instances running slave components, is the size of the pool of available IP addresses, which are parameterized manually in this component.

The main goal of this work is focused on the performance of the system in terms of execution time and resources utilization, and the underlying cloud platform plays an important role in it along with the system architecture itself. Performance-wise, one of the most problematic areas is the deployment of new virtual machine instances. The OpenNebula VM scheduler is limited to deploy one instance per node at a time by the used hypervisor, so anything that can speed up the process around this area is essential. OpenNebula offers two different transfer drivers for deploying VMs, by using a shared file system or using SSH to transfer images. The former approach provides lower VM deployment times, hence it was chosen to be used in FaceID-Cloud. The shared file system utilized is NFS (Pawlowski, Noveck, Robinson, & Thurlow 2000), offering the basic functionality needed and maintaining the image repository shared across all nodes.

4.1.3 Datastore

The chosen distributed file system used as the datastore for the system was already mentioned in Chapter 3, the Hadoop Distributed File System which is part of the Hadoop platform. In this subsection a more detailed description is provided to analyse its impact in the system. According to the documentation of HDFS[1], it "is a distributed file system designed to run on commodity hardware", meaning that, in the context of this work, it will assumedly run relatively well on the modest (and cheaper) VM images available in the cloud. Also, it is built around high-throughput and large datasets, which fits very well with the type of data and manipulation needed. The internal structure of HDFS is based on a master/slave architecture, with a single NameNode controlling a large set of DataNodes that serve read and write requests to the file system's clients, illustrated on Fig. 3.1. To deploy HDFS in the cloud platform, special VM instances different from the others in the system will be used. These instances will feature two custom images at a time from the image repository, a smaller one with the operating system and main file system, and the other with a dedicated large disk image to accommodate HDFS files. Also, the former will be non-persistent, as it does not need to account for changes and so it can be shared by all HDFS nodes, while the later will be persistent and cannot be shared, with one image per node.

4.1.4 Database

Just like the datastore, the database system used was also already discussed. HBase is a column-based, or key/value based, database which scales horizontally by adding more nodes. It is optimized for random realtime read/write access on very large datasets spawning millions of rows, while lacking some of the relational databases features like table indexes or advanced query capabilities. HBase is composed by several complex components, but the most important for the system are the Master Server and the Region Servers. Much like HDFS, HBase also relies on a centralized management, although high availability is assured by a ZooKeeper quorum. Region Servers are responsible for serving read and write requests to the database clients and will be the main agents the system components will interact. Access to HBase is performed through its Java API.

4.1.5 OpenCV

In order to ease the implementation of the eigenfaces algorithm a third-party tool is used - OpenCV, an open-source computer vision framework that provides built-in functionality to process videos and images and perform some operations, namely the application of PCA to a set of images for training the database and projection of images into the new space. OpenCV is originally coded in C++, but the language chosen for the system is Java, hence a wrapper is also used called JavaCV.

[1] http://hadoop.apache.org/docs/r0.18.3/hdfs_design.html (accessed April 2013)

4.1.6 H2 Database

For the caching strategy chosen in Chapter 3, the H2 Java SQL database was added to the system, which provides in-memory databases compatible out of the box with the Java programming environment, and enabling a full relational database to be used in the small replica of the HBase database. Of course, some performance is lost by translating results from a non-relational database to a relational one, but in the long-run the caching effect will be much more evident.

4.2 ID Generation

For the system to be able to keep track of all its components and job workflow, and monitor what is happening at some particular point in time, several unique identifiers are needed. In FaceID-Cloud a mixture of local and distributed identifiers are used to address this need. While some identifiers are local to their parent components, such as the ID of a chunk in the scope of one job which is local to the master and can be modified without any special treatment, other identifiers are shared by several components inside a job and must be made concurrently accessible and, at the same time, in a time and resource efficient manner. A description of the solution developed to address the latter is now given.

There are six counters in the system that can be globally accessed by more than one component in the system - namely those which are used to generate unique IDs for new faces, jobs, people, videos, clusters and eigenfaces data. These are all stored in table ‘‘faceid-data’’ in HBase and hence the access is ruled by HBase's read/write properties and the limited set of ACID properties (Gray et al. 1981) guaranteed. As a key/value database made for massively scalable systems, HBase developers had to let go of some properties to achieve a highly performant system, namely by not guaranteeing all properties of the ACID standard. Simply put, HBase guarantees atomicity, consistency, isolation and durability on a row basis, and not on full transaction basis. This means that, respectively, a transaction will either modify all columns of a row or none of them; a read will always return a complete row that existed at some point of the table's history; transactions are always committed serially so that they do not see each other's inner events; and a row is always made durable in disk before it is read. Despite this limited, but sufficient, set of properties being guaranteed, when generating unique IDs from the counters stored in the database another crucial property is needed - sequential consistency (Lamport 1979). The counters must be incremented by several clients at the same time, but one cannot have two different clients generate the same ID, meaning that increment operations must be sequentially performed in some arbitrary order which can be different from the order at which the operations were issued - strict consistency - but must nonetheless be seen equally ordered by all participants.

Achieving sequential consistency in a distributed system in an efficient manner is not trivial, with the presence of conditions like de-synchronized clocks, latency, and a large number of concurrent

accesses. Among the different counters necessary for the system, the most likely to be heavily accessed and incremented is the face ID generator. Faces are the most numerous objects in the system, followed by grid cells (which do not need a counter), and must be uniquely identified within the system for the recognition process to work. On the other hand, leaps between identifiers are allowed, e.g. from ID 4 to 10, as long as they are unique. This softens the requirements and allows for an easier implementation of a sequential distributed counter.

When processing a video, the rate of new faces entering the system is sufficiently high to invalidate a one-at-a-time generation of IDs, as the continuous database accesses would both slow the process and flood the network due to all nodes having to access the same database table and row, which is implicitly locked by HBase when writing. To remove some pressure on the network and the database, we take advantage of not being forced to provide a fully-sequential generation of IDs (leaps can exist) and make each component ''grab'' a pack of IDs that it can use until it finishes a task or the pack ends, until a new pack will have to be requested again. The size of these packs of IDs is set to 100 so that the face counter does not grow too much with few IDs really representing a face, while also allowing for a considerable amount of faces to be tagged before the component needs another pack and a larger interval between counter accesses by all components.

To request a new set of IDs, a test-and-set strategy is applied, meaning that at least two database roundtrips are needed to definitely increment the counter - one to check the value currently in the database, and the second to write the new value. This second trip to the database will only be successful if the value of the counter is still the same as it was in the first trip. This means, of course, that this second trip must be an atomic operation by itself for it to be able to check the current value and put a new one.

To accomplish an atomic test-and-set operation a method on HBase's API is used which handily provides just that - checkAndPut. This operation atomically checks if the current value on a row on the database is equal to a given input parameter, and sets a new value if it did not change (meaning that no other component has written on that same row between trips). If the counter value in the database did change, meaning that some other component was able to write a new value, the operation must be fully restarted, with the current value being read and a new trip to test the value and try to set it being made. This type of strategy can generate some contention on the counter row, especially when the number of accessing components increases. To lower the contention impact on the system, an optimistic random back-off strategy is used, where each component will sleep for a random amount of time (up to 5 seconds) before trying to obtain a new ID. As an optimistic strategy, the evolution of the system state is guaranteed since there will always be one component that can definitely write its value.

4.3 Detailed Data Model

Following the preliminary definition of the data model on Chapter 3, a more detailed description of the system's data model is now given, enumerating the several tables and their constituents, with some table additions to the base ones defined in section 3.4 which are necessary for the various operational activities in the system not strictly related with the base domain logic. This data model is presented on the HBase table format, where each table is defined with a limited set of column families that cannot be altered at runtime and are represented by their name followed by ":" (ex: "info:"). Each table is composed by unlimited rows ordered alphanumerically by their row key, which are in turn composed by an unlimited number of columns. Each column is associated to a column family and has its own qualifier, being accessed with their corresponding row key, family name and column qualifier (ex: "row1", "family:qualifier"). As columns can be dynamically generated at runtime, sometimes their qualifier can contain some information and change according to the applications needs, so the special notation "<type of information>" is used to denote these special cases.

HBase tries to physically store column families and their columns in the same low-level storage files, so a semantic relation between all columns in a family is desired both for a better understanding of the data model division and to increase the probability of reading from the same physical file/region when performing database operations.

To detail the data model, it is first given a description of the table and its primary use, followed by all the column families and the type of columns they contain. At the end of this section, two subsections are included to describe in more detail some de-normalization that was applied to the data model (Section 4.4) and the strategy used for the representation of a grid cell in the database (Section 4.5).

faceid-data. This is a purely operational table that holds several counters needed for the system to work and accessed concurrently by all components. This is a special table, with solely a default row with several columns holding some system state information:

- **info:clustering** - this holds the current clustering ID for generating new clusters. It is concurrently accessed by all slaves when performing clustering
- **info:databaseid** - holds the current eigenfaces database version, synchronized with the training sessions performed by the monitor/mslaves
- **info:faceid** - counter for generating unique face Ids, accessed concurrently by all slaves
- **info:jobid** - holds the next job ID to be assigned by the monitor to new jobs
- **info:personid** - counter for generating unique people IDs for when registering a new person in the system

- **info:videoid** - counter for generating unique video Ids for when registering videos unknown to the system

faces. This table stores all the faces found by the system, making it the largest table as it will have to store several thousand images. The row key is given by the absolute ID of a face in the system, so that faces from the same video are more likely to be physically stored together, and should benefit from HBase storage strategies and the client cache of the system:

- **info:jobid** - identifies the job/video in which this face was found.
- **images:<relativefaceid>** - the column qualifier states the relative face ID in the scope of the job that created the face. It stores the face image itself in a binary format.

jobs. All job information is available on this table, where monitor, master and slaves can check and update the progress of specific jobs. The row key is a unique job ID assigned by the monitor and maintained centrally:

- **info:faceid** - the counter for the job-relative ID attributed to new faces found in the video
- **info:state** - the current job state, assuming one of the following:
  - BOOT: job initialization by the monitor, will stay in this state until master is allocated;
  - PREPROCESSING: when on this state, the master will be downloading the video from the remote repository and splitting it in chunks;
  - UPLOADING_CHUNKS: after finishing splitting the video, the master will finish uploading the remaining chunks and will wait for the monitor to allocate slaves requested for the job;
  - PROCESSING: in this state, all slaves are processing their assigned chunks and uploading the information generated to the database;
  - FINISHED: after the master acknowledges that all slaves have finished their work, it declares the job as finished, and the web server is free to retrieve the results to present to the client;
  - FAILED: this state represents a known reason of failure, such as an exception thrown by the code or one of the components being unable to contact another component.
- **info:master** - the ID of the master responsible for the job.
- **info:video_id** - the ID attributed to the video being processed, can be considered as a foreign key to the videos table in a relational database.
- **info:nchunks** - the number of chunks that were created from the video.
- **info:nslaves** - the number of slaves allocated to the job.

- **info:nFaces** - the total number of faces found on the video, after aggregating the results from all slaves.
- **faces:<faceid>** - the column qualifier is the relative ID of each face in this job, while the stored value is the absolute ID of the face which acts as a foreign key to the faces table.
- **state:<chunkid>** - tells if a chunk represented by its generation-order ID in the column qualifier was already processed by some slave.

eigenfacesdata. Contains the eigenfaces' training information and related data. Each row represents one training session, with the last row available being the more recent:

- **info:nEigens** - the number of eigenfaces that were considered to train the system.
- **info:nFaces** - the number of faces that were used to train the system.
- **info:eigenvalues** - the eigenvalues that resulted from the training of the system, in binary format
- **eigenfaces:<eigenid>** - holds an eigenface image, the column qualifier being the index of the eigenface.

grid. This table contains information about all grid cells, with each row representing a cell. On each row there is a reference for every face which weight pattern falls in the corresponding cell. The rowkey structure is explained in more detail in Section 4.5.

- **faces:<faceid>,<relativeid>** - holds the weight pattern of the face identified by the face ID in the column qualifier, in binary format. The used relative ID of the face in the job where it was found is also in the qualifier.
- **faces:<job grid key>** - this column indicates that the grid key in the qualifier (which was generated by a job), is part of a general grid key composed of the consolidation of all equal grid keys from all jobs.
- **info:cluster** - holds the cluster to which the grid cell was attributed, acting as a foreign key to the clusters table.

clusters. All clustered face data generated by the system is stored in this table, with the rowkey corresponding to the cluster ID attributed by the slave that created the cluster:

- **grid:<gridkey>** - the column qualifiers in a row hold the gridkeys that were marked as belonging to the cluster of that row, acting as a foreign key to the grid table

4.4 Denormalization

The normalization of a relational data model, assuming the third and last normal form (3NF) defined by Codd (Codd 1972), was proposed as a mean to achieve good relational designs without unnecessary redundancy, while at the same time allowing for easy information retrieval. The focus of normalizing to 3NF is, on a very simplistic view, assuring that all domain elements are atomic, i.e. are indivisible units, and that every non-key element solely depends on the relation (table) key and nothing else.

The effects of normalizing the data-model of an application is usually beneficial, as it reduces redundancy and unnecessary dependencies between data, and accounts for a reduced probability of ending with an incoherent database state upon insert or delete operations, for example, or facilitating the maintenance and upgrade of the data-mode. Yet, a fully normalized database can be affected by scalability and performance problems as the dataset grows and the data-model becomes more complex, so that a simple read can result on several different tables (logically and physically apart) being accessed due to the join query capabilities. There is a trade-off between performance/scalability and consistency/ease of maintenance of the database, being hard to attain both at the same time.

Relational databases, arguably the most common type today and those for which Codd proposed the normalization concept, are greatly affected by scalability problems when the sheer quantity of data to manipulate only fits on several hundred nodes and a join operation incurs on heavy costs due to complicated data-models spread across data-centers (possibly on different continents). Some strategies to reduce the effect of these limitations have been proposed and implemented by relational databases vendors (e.g. sharding (Perl & Seltzer 2006)), but for some cases the problem could be solved by relaxing the normalization standards and de-normalizing part of the data model, especially on simple models which do not need strong consistency properties or join queries.

In the FaceID-Cloud system, the data-model can be made simple enough so that it does not need strict consistency and transaction control, as the great majority of data is uncertain by nature (faces, images, etc) and the main focus is on storing and being capable of accessing large amounts of data, and not on the relations across the dataset. Also, HBase's column-based paradigm strongly implies a de-normalized model, as the concept of primary key is distorted into a ''row key/column family'' approach and foreign keys are discarded all-together. Given these properties of the data to be processed by the system, and the nature of HBase's storage model, the data model was partially de-normalized and some redundancy was purposely added to increase read performance.

The downside of this decision is primarily the added responsibility of the application to handle table relations and accesses to more than one table while maintaining consistency on the cases

where it is indeed needed. Nonetheless, the de-normalization will not carry a large performance effort on the system due to the simple model and the de-normalization-oriented HBase programming paradigm, which does not offer much query capabilities and assigns the responsibility of relating data to the application.

The de-normalization is more evident on the tables grid and clusters, which are closely related as 1-clusters to n-grid. All clusters and grid keys could be placed on a single table, but the issue that arises is related with the querying needs of the system. Imagining this scenario where only one table exists, if one needed to know which grid keys are classified as belonging to a particular cluster, the table should have, in theory, a cluster ID as the row key, and would retrieve all information associated with that ID with a simple read operation. However, if one needed to know which clusters are assigned to a particular grid key this operation would necessarily take a whole table scan searching all cluster IDs for that grid key, since the row key is the cluster ID. The problem would also present itself if the grid key was taken as the table row key. Hence, the de-normalization of these two entities, where one table will have all grid keys and their corresponding clusters - the grid table - and the other will have all clusters with their corresponding grid keys - the clusters table. In a relational and normalized database this problem would not happen at all, as a third intermediate table would be added between both tables which would represent this relation and link the two entities.

4.5 Grid Strategy Implementation

One of the problems of the grid strategy is how to store the necessary information in the database. In Chapter 3, the strategy was defined and the method to find each cell was described. Now on the implementation stage it is necessary to come up with a technical solution to represent cells in the database and uniquely identify them in a search-efficient manner.

HBase is a key/value database, with few query capabilities provided unlike relational databases. One of the major aspects that can influence the performance when querying this type of databases, and specifically HBase, is the row key design. With a clever engineered key, it is possible to extract the needed information efficiently. In HBase, there are three properties of keys that should influence the design. The first is that every row key has a textual representation in ASCII, unlike the columns which are byte arrays that might not have such representation; the second is that row keys are alphabetically sorted, both physically and logically; finally, it is much more efficient to obtain a range of keys than individually requesting them. From these three factors, a row key design for the grid table is proposed.

From Chapter 3, each cell is represented by a point in a grid coordinate system that is obtained by rounding each coordinate of a face's weight pattern vector to its next hundred. The example

given before with point W and its transformation W' into the grid system is now recovered to exemplify the construction of the key:

$W = [\ -142.65,\ 1053.93,\ 0.00,\ 300.00\ ]$
$W' = [\ -200,\ 1100,\ 100,\ 300\ ]$

First, all coordinates will always have two trailing zeros (they are rounded to the next hundred), so they will be divided by 100 from here on, resulting in a shorter key. Each coordinate will be attributed either an n or a p according to its signal, negative or positive respectively, so that the resulting key K in textual format will appear like:

$K = \text{"}n\text{"}\ 2\ \text{"}p\text{"}\ 11\ \text{"}p\text{"}\ 1\ \text{"}p\text{"}\ 3$

The next problem is to translate the numbers to ASCII characters, but guaranteeing that the resulting key can still maintain the natural order of integers - from negative infinity to positive infinity. To achieve this, numbers cannot be directly translated to their ASCII representations, as they will not be correctly ordered. For example, the result of ascendingly sorting the numbers $N = \{100, 2, 10\}$ is $N' = \{2, 10, 100\}$. However, if they are considered as ASCII characters $C = \{\text{"100"}, \text{"2"}, \text{"10"}\}$ they will be sorted in a different order $C' = \{\text{"10"}, \text{"100"}, \text{"2"}\}$. The problem is that number sorting considers the positions of digits in the numbers, so that, for example, a digit in the hundreds position will only be compared with a digit in the same category. String comparison does not make this distinction, and compares characters from left to right regardless of their category in the number.

To account for this problem and still leverage HBase's implicit sort, padding is added to smaller numbers so that the sort algorithm correctly compares digits from the same category. There is still one thing to consider: to add padding characters, the size of the ASCII representation of each number must be predefined. From observation of the eigenfaces algorithm, weight pattern coordinates are almost always under the 3 digits mark (considering all elements divided by 100), and it is very unlikely for a 4 digit coordinate to appear with the kind of entities that are being dealt with. This lead to the definition of a maximum of 2 padding characters per number to reach a 3-digit representation. For example, the smallest number in absolute value that can appear is 1, which needs 2 padding characters in its 3-digit ASCII representation - ''xx1''; a large number with 3 digits does not need padding and should appear as-is. The choice of characters to use when padding is also important, as they will be considered during sort. Characters 'a', 'b' are reserved for this task, where 'a' is always in the leftmost position. With padding, the representation K of vector W is now:

$K = \text{"}nab\text{"}\ 2\ \text{"}pa\text{"}\ 11\ \text{"}pab\text{"}\ 1\ \text{"}pab\text{"}\ 3$

The final step is converting the remaining digits to their ASCII representation, where there is still

an aspect to consider. There is no notion of negative and positive in ASCII sorting, so negative numbers must be encoded differently from positive numbers. On the negative side, numbers with the largest absolute value are actually the smallest and closer to negative infinity, whereas positive numbers are coherent in this aspect. To solve this problem, negative numbers are encoded differently by using upper-case letters and inverting their natural order in the alphabet, while positive numbers use lower-case letters in alphabetical order (see Table 4.1). Finally, the grid key for cell W' becomes:

$K =$ "$nabTpafdpabepabg$"

This encoding ensures a natural ordering between grid cells that enables efficient range-based searches on HBase in the alphanumerically ordered row list.

0	1	2	3	4	5	6	7	8	9
d	e	f	g	h	i	j	k	l	m

(a) Positive numbers

0	1	2	3	4	5	6	7	8	9
V	U	T	S	R	Q	P	O	N	M

(b) Negative numbers

Table 4.1: Grid key number to letter mapping

4.5.1 Clustering Implementation

In the Chapter 3 a description of the clustering strategy to be applied in the system was given, namely about the chosen method to implement called DBSCAN. This algorithm will try to identify individuals in an unknown set of faces by grouping similar faces and tagging them with a cluster ID (a real identification can later be added by an user). By defining the minimum distance ϵ between cluster members as the same minimum distance of similarity in the eigenfaces algorithm, DBSCAN can be seen as variant of this method that compares unknown faces and recognizes individuals in the same dataset by comparing faces within themselves and not with faces on the known face database. Nonetheless, there are some different requirements that should be taken into account when implementing clustering in the system, namely that it will be performed in the last step of the workflow *after* the results of recognition are sent to the end-user, while the eigenfaces method is under a much more restrictive time limit. For that reason, the version of DBSCAN implemented is very similar to the original, without any other strategies to speed up the process except for the utilization of the weight vector cache used by slaves to speed up recognition, since it was already in place when fetching weight vectors from the database and could be seamlessly leveraged.

It is possible to apply the same grid strategy used in the main recognition process to speed up DBSCAN, meaning that when trying to find a neighbor for a given point during the DBSCAN run, only those neighbors which possess a similar grid key are fetched from the database, avoiding a longer table scan and fetch that can lead to a higher load on the database and the network. Arguably, this solution would be a faster approach, but it can lead, nonetheless, to much more faces being ‘‘left out’’ and unclassified due to the sparse nature of the grid. In the main eigenfaces

recognition process, the goal is only to identify at least one of the many faces of a person detected in a video, allowing the algorithm to afford losing some accuracy by not scanning all table rows when searching for a match. Due to the human physiology and natural movements of the head when speaking or looking around, there is a high chance that a frontal view of the head (the optimal face angle to perform identification) will eventually appear in a video, leading to a higher chance of recognizing a person using a limited search scope. However, during this final process of clustering a job's faces the goal is to try to classify *every* face in the video, so that further recognition sessions can use all the information available to more easily identify new faces. Given that there are less restrictive time constraints on this final process, i.e. a user is not waiting for a result, the system can run clustering processes in the background after the job finishes, but not necessarily the moment after it does.

Summary

Following the architecture proposed in the previous Chapter, the process of developing a working prototype of the system is exposed, starting with the description of the technologies used for all parts of the system. Next, a technical problem that was not thought at first in Chapter 3, the generation of unique IDs for the whole system, where a back-off strategy is used. Finally, the data model presented in Chapter 3 is detailed, along with the discussion of some problems affecting the choice of database, namely de-normalization and the implementation of the grid strategy.

5 Evaluation

In this chapter the implemented prototype of the FaceID-Cloud system is evaluated through a series of scenarios that try to stretch it to its computational limits and stress the relation with the cloud platform. It is also necessary to assert that the implemented eigenfaces algorithm is indeed doing its part by setting some specific scenarios to evaluate the system accuracy under controlled conditions, and its behavior on a more "real-world" and uncontrolled scenarios.

The first part of the chapter will focus evaluating the raw performance of the system, making use of different scenarios and discussing the obtained results and how they could be improved. Next, a validation of the clustering algorithm and the usage of the grid strategy is performed, followed by a discussion over the results and their meaning. Finally, a wrap-up on the system's evaluation as a whole is presented and a summary on its strengths and weaknesses. All tests were run under the test environment specified in Appendix A.

5.1 Performance Evaluation

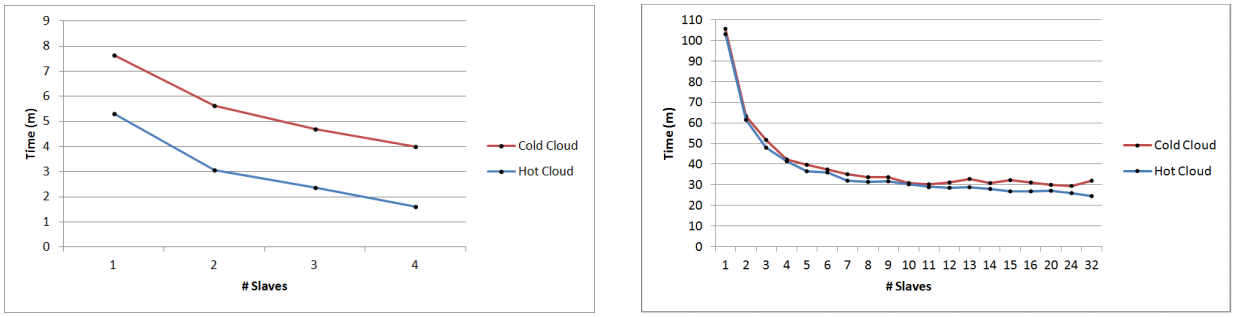
Achieving a much better performance than a sequential version of the eigenfaces method applied to video streams is, as stated in Chapter 1, the main goal of this work. To assert that the goal is reached with the proposed design and the implemented prototype, a series of scenarios were proposed and the system was tested against them, with execution times being measured during the jobs' lifetime through their interaction with the database.

By analyzing the systems's workflow and its most likely bottlenecks before executing the test scenarios, the most probable areas that affect a job execution time are: ''cloud state'', number of slaves available per job, number of simultaneous jobs and the size of the videos. Each of this should not only effectively increase or decrease the performance of the system when altered, but can also affect the others. Hence, three different scenarios are presented that test the cloud state, number of slaves per job, and number of simultaneous jobs, presenting results for small and large videos in each subsection.

5.1.1 Cloud State

In this context, the state of the cloud is the amount of VM instances which are already running in the system. It affects a job's execution time when there is a shortage of VMs available, leading the monitor to allocate more instances. This allocation of new nodes is costly and is negatively

limited by the hypervisor capacity of deploying new VMs simultaneously (only 1 for each node for KVM, a total of 6 in the testing infrastructure), so that recycling of instances is imperative. By making an analogy with common cache systems which are said to be "hot" if they are full and "cold" otherwise, the terms *hot cloud* and *cold cloud* are respectively coined for states where either all instances needed for a job are already running and available, or some or all instances need to be allocated. The two scenarios illustrated in figures 5.1a and 5.1b assert this difference between hot and cold clouds and its impact on the system.



(a) Small Video (b) Large Video

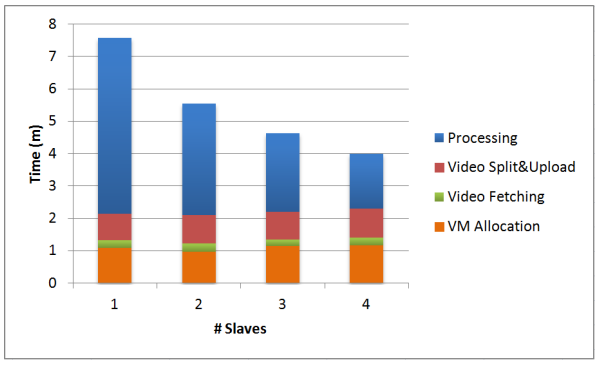
Figure 5.1: Hot and cold cloud scenarios

As expected, it is observed that, as the size of the video grows, the impact of the cloud state in the overall performance is attenuated. In a small video the time that is spent to allocate new VM instances can reach half of the execution time and does make a difference when several small jobs are to be executed leading to a waste of 50% of the cloud resources. On the other end, with large videos the time to allocate VM instances is mostly insignificant when compared with the total execution time. Nonetheless, the allocation time is related with the amount of instances being deployed, and that is the reason for the two curves to slowly start deviating from one another as the number of slaves per job increases. Assuming the database would not become a bottleneck, there would be a moment where the amount of slaves to be allocated would neutralize or even regress the effect of using more parallelism and making the overall execution time go up (that can be noticed in the 32 slave scenario).

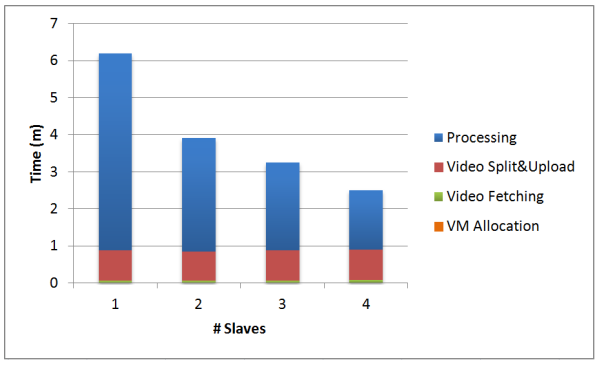
5.1.2 Number of slaves

The main performance driver of the system is the parallelism degree that can be "squeezed" out of a job without touching sensible bottlenecks such as the cloud state or the database. In theory, the addition of more slaves to a job should decrease the processing time or, following Amdahl's Law (Amdahl 1967), increase the theoretical speedup as a function of the problem size and the number of processing units available.

Amdahl states that, given a problem of size n constituted by an inherently sequential part $\sigma(n)$ and a completely parallelizable part $\varphi(n)$, the maximum speedup $\psi(n, p)$ that can be achieved by a system with p processing units can be calculated through the following equations:

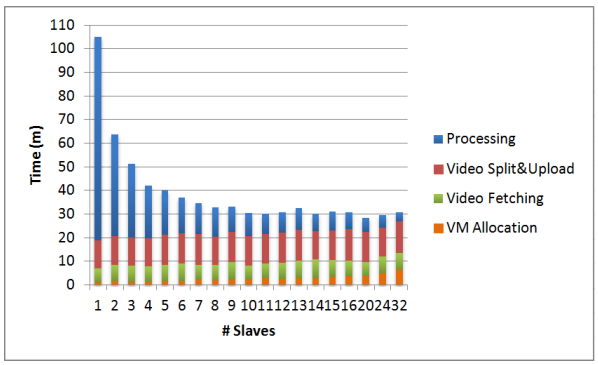


(a) Cold Cloud

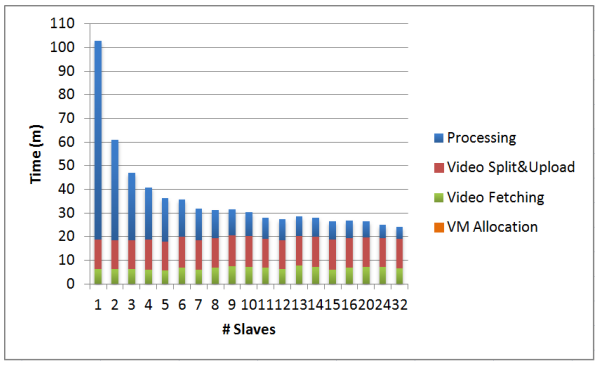


(b) Hot Cloud

Figure 5.2: Small video time distribution



(a) Cold Cloud



(b) Hot Cloud

Figure 5.3: Large video time distribution

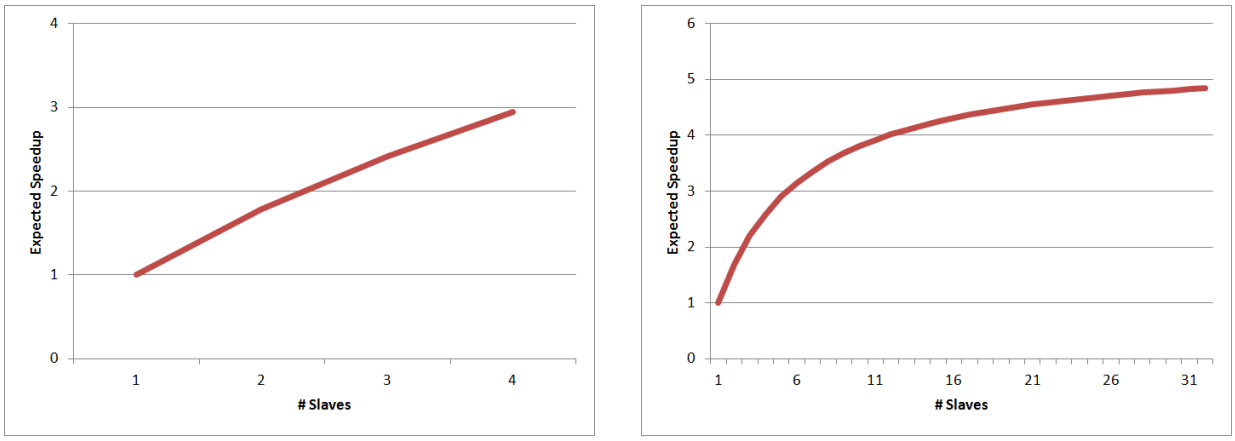
$$f(n) = \frac{\sigma(n)}{\sigma(n) + \varphi(n)} \tag{5.1}$$

$$\psi(n,p) \leq \frac{1}{f(n) + \frac{1-f(n)}{p}} \tag{5.2}$$

$$MAX(\psi(n,p)) = \lim_{p \to +\infty} \frac{1}{f(n) + \frac{1-f(n)}{p}} = \frac{1}{f(n)} \tag{5.3}$$

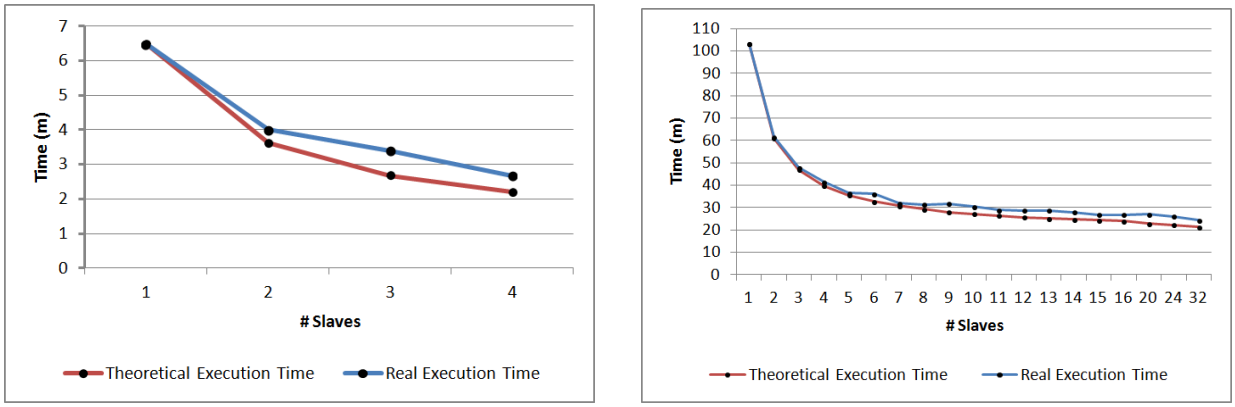
$$f_e(n,p) = \frac{{}^{1}/_{\psi_e(n,p)} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{5.4}$$

where $f(n)$ in 5.1 represents the fraction of the sequential computation in the original sequential program, which in turn is used to calculate the maximum speedup $\psi(n,p)$ in 5.2. Equation 5.3 calculates the theoretical maximum speedup that can be achieved when $p \to +\infty$. Figures 5.4a and 5.4b show the potential speedup of the system when processing the small and large videos with the same number of slaves as the hot cloud scenarios depicted in figures 5.2b and 5.3b. Both the sequential and parallel fractions were estimated based on the execution times with solely one slave.



(a) Small video (b) Large video

Figure 5.4: Theoretical speedup



(a) Small video (b) Large video

Figure 5.5: Execution time

From the speedup curve for the small video scenario, it is clear that the job would benefit with the addition of more slaves so that it could get closer to the potential maximum speedup of 8.37 calculated with equation 5.3, with only 4 slaves. However, it is important to notice that given that this is a small video scenario, the problem size, n, is by itself very small. Taking chunks as indivisible units with a standard size across jobs, even if the number of slaves allocated to the job were to be increased, there would not be enough chunks to process. On the other hand, if the chunk size decreases to allow for more chunks to be generated and more slaves used, large videos would suffer from an explosion of little chunks which would lead to more contention in the database. One could argue that the number of chunks could be dynamically related with video size, but due to the several video sizes and formats available, it is hard to know the nature of a video and how it will "behave" when decomposed into chunks before the process.

For the large video scenario, equation 5.2 sets a theoretical speedup ceiling of 5.53, which seems to be a low number for the amount of slaves provided. The reason for this small speedup is mainly due to the big role that the sequential part of a job plays in the overall execution time, given that after 5 slaves the system spends more time on fetching and splitting a video than on processing it to recognize faces. Nonetheless, by comparing the theoretical minimum execution time with the real time of the hot cloud scenario in Fig. 5.5b, at 32 slaves the system executes at 87.3% of the ideal time (a 3 minutes difference), which account for database and datastore concurrent accesses and oscillations on the fetch and splitting times.

There are two other metrics which allows a more thorough analysis of the system performance that can be obtained using the empirical results from the test scenarios - the efficiency $e(n,p)$, and the Experimentally Determined Serial Fraction (EDSF) $f_e(n,p)$ proposed by A. Karp and P.Flatt in (Karp & Flatt 1990). The efficiency of the system is simply the usage percentage of each processing unit available to the system to reach the resulting speedup, i.e. the measured speedup divided by the number of processing units, meaning that systems should strive to an efficiency closer to 1.0. The EDSF, which is calculated with equation 5.4, reveals aspects of the system performance which are not taken into account into Amdahl's equations, namely overheads inherent to the system (e.g. communication times, different load distribution between processing units). Tables 5.1a and 5.1b present the evolution of the efficiency and EDSF calculated from the measured speedup as p grows for the small and large video scenarios. The results from both tests show a steady speedup grow,

# Slaves	**1**	**2**	**3**	**4**
$\psi_e(n,p)$	-	1.62	1.91	2.43
$e(n,p)$	-	0.81	0.64	0.61
$f_e(n,p)$	-	0.24	0.29	0.22

(a) Small video

# Slaves	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**20**	**24**	**32**
$\psi_e(n,p)$	-	1.68	2.15	2.49	2.83	2.86	3.24	3.28	3.26	3.4	3.56	3.6	3.59	3.69	3.85	3.86	3.82	3.96	4.23
$e(n,p)$	-	0.84	0.72	0.62	0.57	0.48	0.46	0.41	0.36	0.34	0.32	0.3	0.28	0.26	0.26	0.24	0.19	0.16	0.13
$f_e(n,p)$	-	0.19	0.2	0.2	0.19	0.22	0.19	0.21	0.22	0.22	0.21	0.21	0.22	0.22	0.21	0.21	0.22	0.22	0.21

(b) Large video

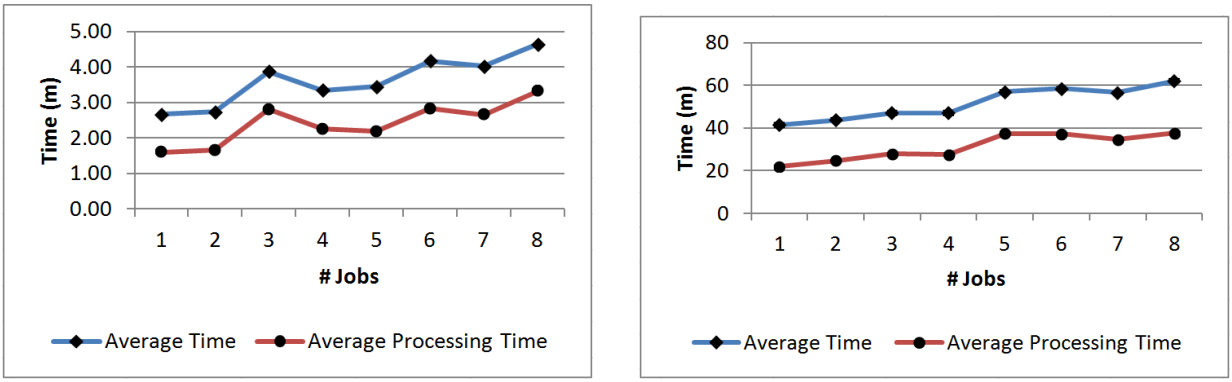
Table 5.1: Performance metrics

but the efficiency suffers deeply, especially on the large video scenario. Nonetheless, this behavior can be partially explained by the Karp-Flatt metric $f_e(n,p)$. According to the authors, there are two known patterns that this metric usually presents - a nearly constant value translates into a very large serial fraction (limited parallelism), while a steady growing value translates into overhead effects.

On the small video, even though there are very few values, f_e expresses a constant behavior with a sudden jump at the 3 slaves mark, indicating an overhead. In fact, this is most probably related with the uneven load distribution of the video chunks. This particular small video is split into 4 chunks by the system, so that when there are only 3 processing units available, one of them will have to do the ‘‘extra work’’, and hence this behavior.

The large video scenario's results are also somewhat expected. Since it is a large video, the load balancing is adequate as there are much more chunks than processing units, so the problem of the small video does not apply here. The key is the constant behavior of f_e which, together with a high decrease of the system's efficiency, translates into a very large sequential fraction that is not parallelizable. This part corresponds to the downloading and splitting of videos (green and red in Fig 5.3), which are not easily split into smaller tasks due to the complexity of video codecs.

5.1.3 Number of simultaneous jobs



(a) Small video

(b) Large video

Figure 5.6: Average execution and processing times with 4 slaves when several jobs execute simultaneously

The last subject of analysis is the behavior of the system when two or more jobs execute at the same time. In both scenarios each job was given 4 slaves to execute, varying from 1 to 8 jobs executing simultaneously for a maximum of 32 slaves processing at the same time. Before the test, a performance drop was expected on the fetching and splitting of videos, as the network should become a bottleneck due to the increased amount of masters trying to do their tasks. Yet, after the test ran the results were very different than expected. As can be seen in Figs. 5.6a and 5.6b, the total executing time (blue) is indeed rising as the number of simultaneous jobs increases, but the main culprit is, surprisingly, the parallel fraction when the slaves are working (red). The relation between the parallel fraction and the total is obvious from analyzing the chart, as they both follow the same type of oscillation, but the reason behind this behavior is much less clear, suggesting some kind of overhead is present. Although it was not expected to be the main reason of performance drop in this specific scenario, communication or database contention were always expected from the system. To perceive the overhead impact in the system, the Karp-Flatt metric is revisited (see equation 5.4 in Section 5.1.2), but this time taking into account solely the parallel fraction, i.e. $\sigma(n) = 0$, meaning that the resulting $f_e(n, p)$ will represent the time lost with overheads. The results from table 5.2 show a small, but slowly increasing overhead effect transmitted by the behavior of f_e. This overhead is not observed in the scenario of Fig. 5.3 due to the greater performance gain

# Slaves	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	20	24	32
$\psi_e(n,p)$	-	1.99	2.95	3.84	4.55	5.27	6.33	7.04	7.64	8.43	9.33	9.4	10	10.44	11.13	11.78	12.44	15.12	15.99
$e(n,p)$	-	0.99	0.98	0.96	0.91	0.88	0.9	0.88	0.85	0.84	0.85	0.78	0.77	0.75	0.74	0.74	0.62	0.63	0.5
$f_e(n,p)$	-	0.01	0.01	0.01	0.02	0.03	0.02	0.02	0.02	0.02	0.02	0.03	0.02	0.03	0.02	0.02	0.03	0.03	0.03

Table 5.2: Analysis of the parallel part of the large video processing

as more slaves are added to the job, but it is still there nonetheless. The efficiency is affected, but at 32 slaves it still performs at around 50% of the total apparent processing power. This number makes some sense given the type of processors featured in the test machines. Despite displaying 8 cores each, due to hyper-threading there are only 4 real cores that try to perform two tasks at the same time using shared resources. While capable of accelerating performance on some cases, it is not assured to do that in all applications as stated by Intel[1], and especially on a system with high memory bandwidth needs as FaceID-Cloud. Of course, this is not the sole reason of the apparent efficiency drop, but it does play a big part in it. Regarding other overheads, they can be reduced, but never fully eliminated, by avoiding database trips, namely when generating new face IDs and when storing job results and by-products.

5.2 Training and Grid Strategy Evaluation

The eigenfaces method is a known approach to face recognition and its performance was already evaluated by its authors (Turk & Pentland 1991), so this section will focus on the specific strategies used by FaceID-Cloud to recognize new people and incorporate them in the database.
There are three main values discussed in the following seciont that should be parameterized as they have an impact in the amount of faces recognized and the quality of the recognition:

- minimum neighbors - this value indicates to the DBSCAN algorithm the minimum neighbor faces that a face must possess to belong to a cluster, i.e. faces under distance ϵ;
- minimum distance ϵ - this value is used both by the eigenfaces method and DBSCAN, indicating the minimum distance under which two faces are said to belong to the same individual. A higher value will group more faces with possibly less accuracy, while a lower value groups less faces but more accurately;
- grid margin - this value indicates the accepted margins to retrieve grid cells from the database, centered on a given cell. A higher value will bring more faces from the database to compare to, raising the probability of recognizing a face, while bearing a higher overall executing time due to the increased amount of network communication and more faces to process. A lower value has, of course, the opposite effect, decreasing the probability of recognition by fetching less faces from the database, lowering network usage and processing time

[1] http://software.intel.com/en-us/articles/performanceinsights-to-intel-hyper-threading-technology (accessed May 2013)

5.2.1 Minimum neighbors

Starting with the minimum neighbors parameter, this value should be set considering the nature of the data to be clustered. In the case of face clusters found in videos, a pre-existing known property is the natural flow of a moving face, i.e. in a video a face is usually in-between two other similar faces, which enables the tracking of the head for example. Then, the system should perform clustering considering a minimum of two neighbor faces.

5.2.2 Minimum Distance ϵ

To ensure the quality of the system information, the value of ϵ should be set so that the system can build a minimum set of clusters that contain all the recognized faces with low noise. As ϵ is greatly dependent on the type of data to be clustered, some tests were performed to find a suitable value. Three test scenarios were devised, based on the amount of individuals featured in each video and the environment conditions (a known cause of underperformance of eigenfaces). The first video contains the perfect scenario, with a single individual frontally facing the camera on a well lit room with some horizontal head rotation. The second video contains two individuals side by side under poorly/difficult light conditions to evaluate the cluster division on noisy videos. Finally, the third video represents a more challenging real world scenario, where 26 individuals are interviewed on the street under different lighting conditions, with rapid movements and more noise-prone situations (e.g. cartoon faces). The results of the three scenarios are shown below on table 5.3, where each scenario was run with different values for ϵ, ranging from 500 to 3000. Information shown corresponds to, respectively, the scenario the data refers to; the ϵ used; the number of people in the video; the number of faces detected (i.e. only detected, not recognized); the number of clusters generated; the number of faces belonging to a cluster; the number of images inside a cluster which do not contain a human face (cluster noise); and the number of clusters which were wrongly merged, belonging in fact to two or more people.

For the first scenario, i.e. the one with the best conditions, the algorithm reaches $\epsilon = 2000$ with every face assigned to a cluster, and only two clusters generated. This division even in the best conditions can be explained as a missing link between the clusters caused by a blurry image when an horizontal rotation is being executed. Due to this mild blurriness combined with a movement, i.e. new faces are probably only ‘‘close’’ to the previous face, there is a gap where there are not a minimum number of neighbors for either of the ”edge”, hence the division of clusters. An example of a cluster resulting from the algorithm with $\epsilon = 1000$ is shown in Fig.5.7, where it can be observed a great similarity between the faces, but always a slight difference due to the head and eye movements.

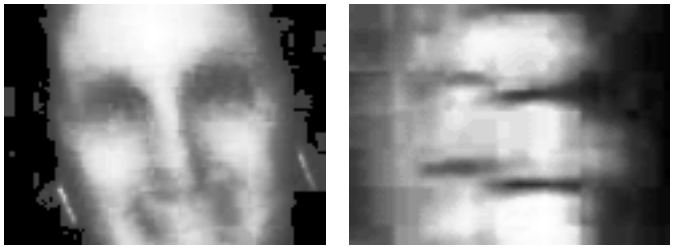
The purpose of the second scenario is to validate that clusters from different people are not merged under controlled conditions. As can be seen in Table 5.3, at $\epsilon = 3000$ still only 178 out of

Scenario	ϵ	People	Faces	Clusters	Clustered faces	Noise	Mixed clusters
1	500	1	97	6	24	0	0
1	1000	1	97	7	75	0	0
1	1500	1	97	6	95	0	0
1	2000	1	97	2	97	0	0
1	2500	1	97	1	97	0	0
1	3000	1	97	1	97	0	0
2	500	2	183	5	43	0	0
2	1000	2	183	9	123	0	0
2	1500	2	183	5	145	0	0
2	2000	2	183	6	169	0	0
2	2500	2	183	5	174	0	0
2	3000	2	183	4	178	0	0
3	500	27	1033	15	105	0	0
3	1000	27	1033	43	505	0	0
3	1500	27	1033	47	704	0	0
3	2000	27	1033	42	821	0	1
3	2500	27	1033	31	906	2	1
3	3000	27	1033	1	683	*	*

Table 5.3: Impact of ϵ on the clustering results

183 faces are recognized. This value of ϵ is very large when comparing the overall results with other ϵ values, so it is unlikely that the faces which are missing do belong to a person or at least to a reasonable quality face image. Indeed, some of the faces left out can be seen in Fig. 5.8, displaying a high level of distortion and blurriness which hardly resembles a human face.

Figure 5.7: Example of a cluster



(a) Noisy face due to poor lighting and blurriness

(b) Noise image wrongly detected by the face detector

Figure 5.8: Example of noise images

Finally, the third scenario is meant to validate the quality of the clusters from an uncontrolled environment video. The large amount of clusters was expected, as well as an increase of the noise both in the detection phase and later on the clustering phase with higher ϵ values. At $\epsilon = 3000$, the algorithm found enough "links" to merge all the clusters, indicating that a value above 2000 can start to display bad results on uncontrolled environments. When manually analyzing the clusters, it is also noted that noise images, i.e. with no faces at all, also constitute clusters on their own. These noise clusters are difficult to eliminate, as their images already passed the face detector and hence present some degree of ''faceness''. Nonetheless, no person will ever be recognized in the future based on these clusters as they are clearly not human, and will not match a human face

weight vector in the trained face space. From the results of the three scenarios, the value for ϵ that seems to display the most coherent and noise free results, while still capable of clustering a high percentage of the total faces is $\epsilon = 2000$, and so it is set as the default value for the system.

5.2.3 Maximum Grid Margin

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2} \tag{5.5}$$

$$d(m) = \sqrt{nm^2} \leftrightarrow d(m) = m\sqrt{n} \leftrightarrow m = \frac{d(m)}{\sqrt{n}} \tag{5.6}$$

After settling with an ϵ, the goal is to find a margin value m that, given the number of dimensions n of the face space, results in a maximum distance slightly above ϵ when the two points are at distance m from each other in every dimension. Given that the Euclidean distance between two points p and q is calculated with equation 5.5, it is possible to set every $(p_x - q_x) = m$ and use equation 5.6 to find the theoretical exact value for m which has the highest probability of fetching all the faces within range of the test face. Using the actual values of the system with $n = 20$ dimensions and a distance $d(m) = \epsilon = 2000$, the best m would be 447.21. However, to increase the likelihood of a successful classification m is set to 500, leading to a maximum distance of 2236.07.

Finally, please do note that it is possible for two faces from the same person to present a large distance in one dimension which is balanced by a small value in another dimension, leading to a distance smaller than ϵ. Arguably, this is not a desirable condition for the system, as it means that it is dealing with a distant edge face which has a higher possibility of being noise or from a different person, on top of the inconsistent variance between two or more dimensions that can cause this scenario.

Summary

The purpose of this Chapter is to validate the proposed architecture, testing it under different scenarios and observe its behavior. It starts by a description of the test environment and the pre-existing conditions that affect the different scenarios analyzed. It continues with an evaluation of the raw performance of the system by testing with small and large videos under different perspectives: cloud state, number of slaves per job and number of concurrent job, followed by a simple evaluation of the face recognition module and the usage of clustering and grid strategy.

6 Conclusions

This chapter aims to wrap-up this work by discussing the obtained results and the strengths and weaknesses of the presented solution identified during the development and their importance on the usage of this system in the real-world. Also, this document presented a working solution for the problem stated in Chapter 1, but it can, nonetheless, be further improved and extended to include more features.

6.1 Discussion

On Chapter 5 the system was evaluated in a variety of scenarios which have shown its overall good response to heavy load and the advantages of using such a solution. Given the system design and the obtained results, the following two lists summarize the main strengths and weaknesses of using FaceID-Cloud:

Strengths

- Automatic sizing of the system based on load, which is important in a pay-as-you-go model aimed for the cloud
- Horizontally scalable, leveraging the Cloud Computing inherent scalability together with an independence between tasks
- Database search scope limitation based on a grid strategy together with the usage of client caching that accelerate the identification of faces
- Capability of adding new unknown people to the database without human intervention with reduced noise levels due to the usage of DBSCAN (always depending on the quality of the videos)

Weaknesses

- Network bandwidth is an important bottleneck in the system due to the high throughput needed to transfer video material
- On large and/or complex videos, the steps to transfer, split and upload the videos start to take more time than the processing of the videos as the number of slaves increase, leading to an efficiency drop due to the large non-parallelizable portion
- As an automatic tool, it can wrongly acknowledge noise or very bad quality face images as real faces and add them to a real person cluster in the database

The list of weaknesses presented is mostly composed of inherent problems to video processing, namely the heavy load on the network and the time needed to transfer videos and prepare them to be processed. The last item is more questionable, and is a result of the specific strategy applied in the system. Noise is already a problem in clustering algorithms, even though DBSCAN deals well with it and actively removes noise points from the dataset. However, the noise that might affect FaceID-Cloud is on a different domain from that which DBSCAN operates, in the sense that noise in FaceID-Cloud is an image that is not a human face, and not a point which is far apart from the rest of the dataset. An example of this distinction is that DBSCAN might group a set of faceless images and consider it a cluster (even though it is noise in the FaceID-Cloud domain), while a real face image can be considered noise if it is located very far from other faces (a person appearing in a single frame, though unlikely).

While a full ''noise-cluster'' is not a problem given that it will probably not match a person face in the future, a noise image in the middle a real person cluster might get a noise image recognized as humans. Of course, one could reduce the minimum distance of similarity ϵ and accept only sets of faces which are deoitely very similar, but this would lead to less faces being recognized. In this trade-off, the decision was made, as stated in Chapter 5, to pick the largest value for ϵ which did not present intra-cluster noise. Another strategy to reduce the impact of this problem could be to clean the clusters after they are formed. DBSCAN has no notion of a cluster centroid (the average point), and it could prove interesting to mix this concept with this density-based cluster algorithm, enabling the removal of faces which are the farthest from the centroid and possibly eliminating intra-cluster noise.

Overall, it is the author's opinion that the strengths greatly overcome the weaknesses and that the system can perform solidly given a good network connection and a large installation of HBase+HDFS dedicated virtual machines - the two main bottlenecks.

Future Work Despite being a sound design which can handle large workloads, the system monitor can still become a bottleneck when acting as a middleman between web servers and the rest of the system. This design simplifies load balancing and avoids keeping all system state in the database. However, given the high scalability of the cloud and the total independence between jobs, it should be possible to take this system to the next level and deploy more than one monitor instance at the same time, each controlling part of the deployed virtual machines. This design could increase the availability of the system and reduce the response times when deploying geographically near the end-users.

Another area of improvement is on the client side and the access to the system. Currently, only a basic web interface is provided with the implemented prototype, but given the on-going

transition of computation to the cloud, the goal should be to provide access to the system through smartphones and tablets which can upload videos for processing and search the videos a person is part of.

Finally, the best improvement that could (probably) be made is to add a newer, more accurate face recognition algorithm. Eigenfaces was chosen as an easy solution to provide a basic recognition functionality, but it is surely not the best method today, so an effort should be made to integrate a new method, which is facilitated by the modular decomposition of the system.

Final Thoughts on the System Application This system could be leveraged in some ways, exploring essentially niche markets as a tool for companies with large video databases to build a catalog of people appearing in both old and new videos, enriching their knowledge database and allowing new features to be made available for end-users which were not possible 20 years ago. From another perspective, FaceID-Cloud could be used by individuals to search for videos where a person appears by providing a photo from their mobile device, leveraging the Cloud Computing paradigm.

References

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485. ACM.

Armbrust, M., A. D. Joseph, R. H. Katz, & D. A. Patterson (2009). Above the Clouds : A Berkeley View of Cloud Computing. Technical report, University of California at Berkeley.

Bartlett, J. C. & J. Searcy (1993, July). Inversion and configuration of faces. *Cognitive psychology 25*(3), 281–316.

Belhumeur, P., J. Hespanha, & D. Kriegman (1997, July). Eigenfaces vs. Fisherfaces: recognition using class specific linear projection. *IEEE Transactions on Pattern Analysis and Machine Intelligence 19*(7), 711–720.

Bent, J., T. E. Denehy, M. Livny, A. C. Arpaci-Dusseau, & R. H. Arpaci-Dusseau (2009). Data-driven batch scheduling. In *Proceedings of the second international workshop on Data-aware distributed computing - DADC '09*, New York, New York, USA, pp. 1–10. ACM Press.

Bojanova, I. & A. Samba (2011, March). Analysis of Cloud Computing Delivery Architecture Models. In *Workshops of International Conference on Advanced Information Networking and Applications*, pp. 453–458. IEEE.

Brewer, E. A. (2000). Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, New York, NY, USA, pp. 7–. ACM.

Brunelli, R. & T. Poggio (1993). Face recognition: features versus templates. *IEEE Transactions on Pattern Analysis and Machine Intelligence 15*(10), 1042–1052.

Buyya, R., C. S. Yeo, & S. Venugopal (2008, September). Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities. In *10th IEEE International Conference on High Performance Computing and Communications*, pp. 5–13. Ieee.

Caldeira, R. & L. Veiga (2012). Faceid-cloud - face identification leveraging utility and cloud computing. In *Proceedings of INForum - 4th Portuguese National Symposium on Informatics*. DI-FCT-UNLisboa.

Chellappa, R., C. Wilson, & S. Sirohey (1995, May). Human and machine recognition of faces: a survey. *Proceedings of the IEEE 83*(5), 705–741.

Choudhury, T., B. Clarkson, T. Jebara, & A. Pentland (1999). Multimodal Person Recognition using Unconstrained Audio and Video. Technical report.

Codd, E. F. (1972). Further normalization of the data base relational model. *Data base systems*, 33–64.

Cox, I. J. & N. Yianilos (1996). Feature-Based Face Recognition Using Mixt ure-Distance Joumaiia Ghosn. Technical report.

Dean, J. & S. Ghemawat (2008). MapReduce: Simplified data processing on large clusters. *6th Symposium on Operating Systems Design & Implementation 51*(1), 107–113.

Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR.

Ester, M., H.-p. Kriegel, J. Sander, & X. Xu (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. pp. 226–231. AAAI Press.

Etemad, K. & R. Chellappa (1997, August). Discriminant analysis for recognition of human face images. *Journal of the Optical Society of America A 14*(8), 1724.

Geelan, J. (2009). Twenty-One Experts Define Cloud Computing.

Ghemawat, S., H. Gobioff, & S.-t. Leung (2003). The Google File System. Technical report, Google.

Gong, C., J. Liu, Q. Zhang, H. Chen, & Z. Gong (2010, September). The Characteristics of Cloud Computing. In *39th International Conference on Parallel Processing Workshops*, pp. 275–279. IEEE.

Gray, J. et al. (1981). The transaction concept: Virtues and limitations. In *Proceedings of the Very Large Database Conference*, pp. 144–154.

Han, J., M. Kamber, & J. Pei (2006). *Data mining: concepts and techniques*. Morgan kaufmann.

Hongxun, Y., G. Wen, L. Mingbao, & Z. Lizhuang (2000). Eigen features technique and its application. In *5th International Conference in Signal Processing Proceedings*, Volume 2, pp. 1153–1158.

IBM & Google (2007). Google and IBM Announce University Initiative to Address Internet-Scale Computing Challenges.

Kanade, T. (1973). *Computer Recognition of Human Faces*.

Karp, A. H. & H. P. Flatt (1990). Measuring parallel processor performance. *Communications of the ACM 33*(5), 539–543.

Kosar, T. & M. Balman (2009, April). A new paradigm: Data-aware scheduling in grid computing. *Future Generation Computer Systems 25*(4), 406–413.

Kosar, T., M. Livny, W. D. Street, & M. Wi (2004). Stork : Making Data Placement a First Class Citizen in the Grid. In *Proceedings of the 24th International Conference on Distributed Computing Systems*. IEEE.

Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on 100*(9), 690–691.

Lawrence, S., C. L. Giles, a. C. Tsoi, & a. D. Back (1997, January). Face recognition: a convolutional neural-network approach. *IEEE transactions on neural networks 8*(1), 98–113.

Lin, S. H., S. Y. Kung, & L. J. Lin (1997, January). Face recognition/detection by probabilistic decision-based neural network. *IEEE transactions on neural networks 8*(1), 114–32.

Liu, C. & H. Wechsler (2000). Evolutionary pursuit and its application to face recognition. *IEEE Transacations on Pattern Analysis and Machine Intelligence 22*(6), 570–582.

Liu, X. & T. Chen (2003). Video-Based Face Recognition Using Adaptive Hidden Markov Models. In *Proceedings of the 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*.

Morais, J., J. N. Silva, P. Ferreira, & L. Veiga (2011). Transparent adaptation of e-science applications for parallel and cycle-sharing infrastructures. In *Distributed Applications and Interoperable Systems*, pp. 292–300. Springer.

Navarrete, P. & J. Ruiz-del solar (2002). Analysis and Comparison of Eigenspace-based Face Recognition Approaches. Technical report.

Nefian, A. & M. Hayes III (1998). Hidden Markov models for face recognition. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, Volume 5, pp. 2721–2724. IEEE.

Oracle (2011). Database as a Service : Reference Architecture - An Overview. Technical Report September.

O'Toole, A., P. J. Phillips, F. Jiang, J. Ayyad, N. Penard, & H. Abdi (2005). Face recognition algorithms surpass humans. *IEEE Trans. PAMI*.

Pawlowski, B., D. Noveck, D. Robinson, & R. Thurlow (2000). The nfs version 4 protocol. In *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000*.

Pentland, A., B. Moghaddam, & T. Starner (1994). View-based and modular eigenspaces for face recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, Volume 02139, pp. 84–91. IEEE Comput. Soc. Press.

Perl, S. E. & M. Seltzer (2006). Data management for internet-scale single-sign-on. In *Proceedings of the 3rd conference on USENIX Workshop on Real, Large Distributed Systems*, Volume 3.

Prodan, R. & S. Ostermann (2009). A survey and taxonomy of infrastructure as a service and web hosting cloud providers. In *10th IEEE/ACM International Conference on Grid Computing*, pp. 17–25.

Rajan, S. & A. Jairath (2011, June). Cloud Computing: The Fifth Generation of Computing. In *International Conference on Communication Systems and Network Technologies*, pp. 665–667. IEEE.

Rappa, M. a. (2004). The utility business model and the future of computing services. *IBM Systems Journal 43*(1), 32–42.

Sakr, S., A. Liu, D. M. Batista, & M. Alomari (2011). A Survey of Large Scale Data Management Approaches in Cloud Environments. *IEEE Communications Surveys & Tutorials 13*, 311–336.

Sinha, P., B. Balas, Y. Ostrovsky, & R. Russell (2006, November). Face Recognition by Humans: Nineteen Results All Computer Vision Researchers Should Know About. *Proceedings of the IEEE 94*, 1948–1962.

Sirovich, L. & M. Kirby (1987). Low-dimensional Procedure for the Characterization of Human Faces. *Journal of the Optical Society of America A 4*, 519.

Stewart, M., H. M. Lades, & T. J. Sejnowski (1998). Independent component representations for face recognition. In *Proceedings of the SPIE Symposium on Electronic Imaging: Science and Technology, Conference on Human Vision and Electronic Imaging III*, California.

Treleaven, P. C., D. R. Brownbridge, & R. P. Hopkins (1982). Data-Driven and Demand-Driven Computer Architecture. *Computing Surveys 14*(March).

Tsai, W.-T., X. Sun, & J. Balasooriya (2010). Service-Oriented Cloud Computing Architecture. In *Seventh International Conference on Information Technology*, pp. 684–689. IEEE.

Turk, M. & A. Pentland (1991). Face recognition using eigenfaces. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 586–591. IEEE Comput. Sco. Press.

Vaquero, L. M., L. Rodero-Merino, J. Caceres, & M. Lindner (2008, December). A Break in the Clouds: Towards a Cloud Definition. *ACM SIGCOMM Computer Communication Review 39*(1), 50–55.

Voas, J. & J. Zhang (2009). Cloud Computing : New Wine or Just a New Bottle ? *IT Professional 11*(2), 15–17.

Vouk, M. (2008). Cloud computing-Issues, research and implementations. *Journal of Computing and Information Technology 16*(4), 235–246.

Wang, L., G. Laszewski, A. Younge, X. He, M. Kunze, J. Tao, & C. Fu (2010, June). Cloud Computing: a Perspective Study. *New Generation Computing 28*(2), 137–146.

Wang, L., J. Tao, M. Kunze, A. C. Castellanos, D. Kramer, & W. Karl (2008, September). Scientific Cloud Computing: Early Definition and Experience. In *10th IEEE International Conference on High Performance Computing and Communications*, pp. 825–830. Ieee.

Weinhardt, C., A. Anandasivam, B. Blau, & J. Stöß er (2009). Business Models in the Service World. *IT Professional 11*(April, no.2), 28–33.

Wiskott, L., J.-M. Fellous, N. Kuiger, & C. von der Malsburg (1997, July). Face recognition by elastic bunch graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence 19*(7), 775–779.

Wood, T., P. Shenoy, A. Gerber, & K. Ramakrishnan (2009). The Case for Enterprise-Ready Virtual Private Clouds. Technical report.

Zhang, Q., L. Cheng, & R. Boutaba (2010, April). Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications 1*(1), 7–18.

Zhao, W., R. Chellappa, P. Phillips, & A. Rosenfeld (2003). Face recognition: A literature survey. *Acm Computing Surveys 35*(4), 399–458.

A Test Environment

The behavior of the system is, naturally, greatly dependent on the environment onto which it is being deployed. At a high abstraction level, the system will always see the same predefined virtual environment, with the same type of resources available to each type of virtual machine instance which imposes limits on the application usage of memory or the number of virtual processors available. Yet, it is the hardware and software beneath the cloud middleware that do make the difference and set limits such as processor speed or network bandwidth. For this reason, three layers of abstraction are specified: physical, middleware and application.

A.1 Physical Layer

All the test scenarios executed in this chapter were performed under the same physical hardware conditions, with no other computation/network heavy resource other than the cloud platform running at the same time. The cloud middleware was deployed into a 6 node cluster featuring Intel Core i7-2600K @ 3.40GHz processors (quad-core with hyper-threading and two memory channels), 12GB of DDR3 RAM @ 1333MHz, 7200 RPM hard-disk drive, Gigabit ethernet network connecting all nodes and Ubuntu Server 12.04.1 LTS 64bit installed.

Along with the hardware and operating system, other software was installed as well. The first one is the hypervisor, needed for the cloud platform to run VM instances on the machine. Among the ones supported by default by OpenNebula, KVM proved to be of easy installation on the system, and was chosen for that reason. While functional, it does impose a limit of 1 VM deployment per node at a time and contributes to the overall system performance. The second component which is needed by the cloud platform is a distributed file system for virtual machine images to be shared across nodes and accelerate the deployment process. As stated in Chapter 4, the Network File System (NFS) was chosen to provide this functionality. It was setup in all nodes with root on the image repository of OpenNebula for a seamless integration. It should not, however, make a noticeable difference in the system performance, as most images used are predefined and do not change overtime, so that NFS is not forced to constantly update an image in a node which is to receive a new VM instance.

A.2 Middleware Layer

OpenNebula, the cloud platform chosen to use in the system, is deployed directly into the hardware specified in the previous subsection with no modifications and/or tuning. As described in section 4.1.2 of chapter 4, there are three services that must be configured to have a functional cloud: VM scheduler, image repository and virtual network manager. Each of this components is configured through the use of templates which contain information to instantiate the type of object handled by the service, i.e. VM instances, images in the repository and virtual networks. Each template specifies the virtual hardware that will be perceived by the FaceID-Cloud components, such as the number of processors or the IP addresses available in the network. A list of the key points of each template is now given:

- **CPU:** All VM instances use at most 1 virtual processing unit with the exception of masters, who are allowed to use up to 2 virtual processing units to try to perform video splitting and upload simultaneously
- **Main memory:** Database specific VM instances can use up to 2GB of RAM, while the rest can only use 1GB. HBase is very dependent on RAM, and it is essential that nodes with region servers do not start swaping, hence the extra memory allocated. On a real production system, this number should be higher to account for the extra load, but on the test scenarios 2GB is enough to cover all the system needs
- **Disk image:** There are three types of VM instances which use different pre-configured images running a minimal version of Ubuntu Server 10.04.3 LTS 32bit:
  - Web Application instances use an image (1.3 GB) with a tomcat server pre-installed and loaded with the FaceID-Cloud front-end application
  - Database instances use two types of images. One is shared by all database nodes and is pre-configured with HDFS and HBase executables (1.1 GB), while the other is much larger (maximum 10GB in the test enviroment) and specific to each node. This latter type can be altered during the lifetime of the owner VM instance, serving as the main storage site for HDFS data. There has to be a different image of this type for each database node added
  - All other components use the same base image (1.3 GB) loaded with all FaceID-Cloud executables

  Image sizes are specified, as they affect deployment times, but they do not represent the real disk space available within, as the format QCOW2 is used which incrementally adds space as needed and keeps the images small

A.3 Application Layer

The majority of the test scenarios were run under the same system state, with some exceptions existing when stated. The base scenario starts with all system areas except the Computation area with their components instantiated. Both Web Interaction and Management areas have each a single VM instance running the web app and the monitor respectively, with memory and cache completely free from system data of previous jobs. The Storage area has 3 VM instances running with HDFS and HBase deployed on them, but unlike the other areas the state is kept between tests. This should affect region caching on main memory which could influence the results, so before a round of tests the database is "warmed" with a dummy job so that every test is run under the same conditions.

B Additional Tables

Approach	Sub-class	Example systems
Holistic-based	Principal Component Analysis	Turk and Pentland(Turk & Pentland 1991)
	Fisher's Linear Discriminant	Belhumeur et al.(Belhumeur, Hespanha, & Kriegman 1997)
	Linear Discriminant Analysis	Etemad and Chellappa(Etemad & Chellappa 1997)
	Independent Component Analysis	Stewart et al.(Stewart, Lades, & Sejnowski 1998)
	Evolutionary Pursuit	Liu and Wechsler(Liu & Wechsler 2000)
Feature-based	Purely Geometric Methods	Kanade(Kanade 1973); Cox et al.(Cox & Yianilos 1996)
	Dynamic Link Architecture	Wiskott et al.(Wiskott, Fellous, Kuiger, & von der Malsburg 1997)
	Hidden Markov Model	Nefian and Hayes(Nefian & Hayes III 1998)
	Convolution Neural Network	Lawrence et al.(Lawrence, Giles, Tsoi, & Back 1997)
Hybrid-based	Modular Eigenfaces	Pentland et al.(Pentland, Moghaddam, & Starner 1994)
	Probabilistic Decision Based Neural Networks	Lin et al.(Lin, Kung, & Lin 1997)

Table B.1: Classification of Face Identification systems

Approach	Example systems
Still-image methods	Turk and Pentland(Turk & Pentland 1991); Pentland et al.(Pentland, Moghaddam, & Starner 1994); Lin et al.(Lin, Kung, & Lin 1997)
Multimodal methods	Choudhury et al.(Choudhury, Clarkson, Jebara, & Pentland 1999)
Spatio-temporal methods	Liu and Chen(Liu & Chen 2003)

Table B.2: Classification of Video Face Identification systems

Feature	Amazon Web Services	Microsoft Windows Azure	Google App Engine	Eucalyptus	OpenNebula
Computing Architecture	Elastic resources (EC2), multiple instance types operating systems and software packages; elastic IPs; availability zones; automatic load balancing	Automatic management of VM instances and load balancing	Automatic scaling and load balancing; task queues	Cloud requests serviced asynchronously; elastic IPs; automatic scaling and load balancing	Dynamic resizing and partitioning; centralized management; utilizes existing heterogeneous resources
Service Model	IaaS; PaaS	PaaS	PaaS	IaaS	IaaS
Deployment Model	Public; Virtual Private Cloud	Public; Hybrid	Public; Virtual Private Cloud	Public; Private; Hybrid; Community	Public; Private; Hybrid
Virtualization Technology	Xen	Windows Azure Hypervisor (Hyper-V)	No hypervisor	Hypervisor-agnostic architecture (Xen, KVM compatibe at the moment)	Hypervisor-agnostic architecture (Xen, KVM and VMare compatible at the moment)
Storage System	"Bulk" storage (S3); query-capable non-relational database (SimpleDB)	Blobs, Tables, SQL Azure	BigTable	Walrus (S3 compatible)	SQLite; supports most file systems through textual configuration files
API	SOAP; REST	Windows Azure Service Management API	APIs provided for main services (e.g. datastore)	AWS-compatible	XML-RPC; EC2 Query subset; OGF OCCI
Programming Model	Amazon Machine Images; Amazon Elastic MapReduce	Windows Azure Hosted Services Application Model	Google App Engine SDK	Unknown	Unknown
Service Level Agreement	99.95% uptime, client elegible to Service Credit otherwise	99.95% uptime for two or more instances in different domains, 99.9% for all other services	Up to 99.95% uptime with different client Financial Credit compensations	Best-effort basis only on community deployment	Not applicable (open-source)
Pricing	Per hour (computation); per GB (network, storage)	Per instance (computation); per GB (network, storage); 6-month plans	Per hour (computation); Per GB (network, storage); per operation count (datastore)	Not applicable	Not applicable
Security	Firewall between instance groups; VPC; identity and access management; security groups	Confidentiality, integrity, availability, accountability	Multi-layered security strategy; multiple levels of data storage, access, and transfer	Similar to EC2, except VPC	Auth subsystem for users and groups
Component Coupling	High coupling (e.g. EC2 / S3)	High coupling	Access to some services is code-based (APIs), so allows other substitute components in principle (medium coupling)	Substitute components need to implement Web Services interface (medium coupling)	Low coupling; in general, substitute components can be attached with minor effort

Table B.3*a*: Cloud Computing systems classification

Feature	Nimbus	Flexiscale	IBM SmartCloud Services	Force.com	OpenStack
Computing Architecture	Fast-propagation; non-invasive site-scheduler; auto-configurable clusters	Highly automated and rapid provisioning of resources; allows multi-tier architecture; designed to deliver a guaranteed QoS level	Fast deployment and dynamic provisioning of resources accross heterogeneous environments; Integrated into the IBM software and hardware ecosystem	Metadata-driven software architecture enabling multi-tenant sofware applications	Shared-nothing design; Multiple network models; Floating IPs; Distributed scheduler; Asynchronous architecture
Service Model	IaaS; some PaaS features (Nimbus Platform)	IaaS	IaaS; PaaS; SaaS	PaaS; SaaS	IaaS
Deployment Model	Private; Hybrid	Public	Public; Private; Hybrid; Virtual Private Cloud	Public	Public; Private
Virtualization Technology	Xen; KVM	Xen	Heterogenous hypervisor support; POWER Hypervisor	Unknown	Hypervisor-agnostic (Hyper-V, Citrix XenServer, Xen, KVM, VMWare ESX, LXC, QEMU, UML
Storage System	Posix filesystem backend storage system	Virtual disks on highly redundant SAN disk array	IBM Storwize; IBM XIV; IBM SONAS	Virtual relational database structures	AoE over LVM; S3 compatible; Several supplementary block storage options provided
API	WSRF; EC2 SOAP and Query subset; Cumulus	Extility (SOAP)	RESTful API; Java API	SOAP	Openstack API; EC2 and S3 support
Programming Model	Nimbus Platform	Unknown	IBM SmartCloud Application Services	Force.com Platform	Unknown
Service Level Agreement	Not applicable (open-source)	100% uptime, client credit if not achieved	Up to 99.9% uptime	99.9% uptime	Not applicable (open-source)
Pricing	Not applicable	Unit pricing system for all services	Per hour (computation); per GB (network, storage)	Applications per user per month	Not Applicable
Security	X509 Credentials; group policies	Port based firewall; VLANs and private virtual disks	Firewall; IP-filtering; VPN; patched and scanned images; hypervisor isolation; access control	Access control (digital and physical); firewall and edge routers; intrusion detection sensors; third party scan of the network	Rate limiting and autentication; Security Groups; Role Based Access Control; Federated Auth with Zones
Component Coupling	Low coupling; new components easily integrated	High coupling	High coupling	High coupling	Low coupling; Modular design can integrate with legacy or third-party technologies

Table B.3*b*: Cloud Computing systems classification