FaceID-Grid

A Grid Platform for Face Detection and Identification in Video Storage

Filipe Miguel Rodrigues

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

November 2013

Acknowledgements

I would like to gratefully thank Professor Luís Veiga for all his guidance, patience and motivation. For always being available to help and listen to my difficulties.

I would like to thank my mother for the unconditional support that helped me to overcome the difficult times.

Finally, I would also like to thank FCT for the financing of this work in the context of the project PROSOPON, PTDC/EIA-EIA/102250/2008

Lisboa, November 25, 2013

Filipe Rodrigues

Resumo

Os sistemas de reconhecimento facial têm despertado um notório interesse por parte de investigadores e do mercado empresarial; as suas aplicações são vastas, por exemplo jogos de computador, redes sociais e segurança. Têm sido feitos avanços significativos no campo do reconhecimento facial, no entanto o reconhecimento facial é ainda uma tarefa computacionalmente pesada no contexto de vídeos, que se for executado numa única maquina tem serias limitações às suas potencialidades. O objectivo desta tese foi desenvolver um sistema de reconhecimento facial que seja executado num ambiente Grid a fim de criar um sistema distribuído, escalável e eficiente. Utilizamos software de reconhecimento facial executável numa só máquina da biblioteca OpenCV para suportar as funções de reconhecimento facial e o sistema é executado num cluster Condor. Desenvolvemos a arquitectura distribuída do sistema, um esquema de armazenamento dos dados do reconhecimento facial, um sistema de coordenação para a execução distribuída, um sistema de descoberta de recursos especialmente desenhado para as necessidades da aplicação e uma parametrização para o algoritmo de escalonamento.

Palavras-chave: Grid, Vídeo, Reconhecimento Facial, Eigenvectors, Condor, HDFS.

Abstract

Face Recognition systems have received significant attention from the research community as well as from the market; its applications are now vast, such as video games, social networking and security. Significant advances have been made in the face recognition field; however face recognition is still a demanding task in the context of videos that, if executed in a single machine, has severe limitations to its potentialities. The goal of this thesis was to develop a face recognition system that is executed in a grid environment to create a distributed, scalable and efficient system. We used existing stand-alone face recognition software from the OpenCV library to support the face recognition functions and the system is executed on top of a Condor grid. We develop the distributed architecture of the system, a file storage schema for the face recognition data, a distributed coordination system for the distributed execution, a resource discovery system tailored for the specific needs of this application and a scheduling algorithm approach for the system.

Keywords: Grid, Video, Face Recognition, Eigenvectors, Condor, HDFS.

Index

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Distributed computing is a successful way of proving high computational power to applications; one of the more common types of distributed computing is grid computing, where essentially the computational power of a number of machines is combined to execute a given application.

High resource demanding applications can benefit from being executed in a grid environment. Traditionally, these applications would have to be executed in super computers or large scale dedicated computer infrastructures which would provide the resources those applications need. However, super computers or dedicated computer infrastructures are not at the financial reach of all organizations; Grid Computing on the other hand, is able to harness the computational power of commodity hardware, and due to the available grid middleware, it makes a relatively affordable way of obtaining high computational capacity; moreover as grid infrastructures are not dedicated to a given application, organizations can share resources. All of these facts, justify the success grid computing has had and is having in the scientific community.

Face Recognition, especially in videos, is an example of a high demanding application that can benefit from grid computing. While, in the literature, we can find a large amount of research done in grid computing, to this date we are not aware of any developed or planned face recognition system that leverages grid computing, despite being relatively evident that such a system can be created. Grid Computing works well with a particular type of parallel computation, the multi-data-single-instruction; which is precisely what a face recognition system is, multiple images to be process always by the same code.

1.2 Shortcomings of current solutions

Face Recognition is a very resource demanding application. The current solutions, which are standalone applications are limited in:

- Processing power in the case of commodity hardware.
- Scalability in the case of super computers.

Also, in the existing applications:

- All the known faces are stored in a single machine, and even when they are not, they all have to be loaded into memory in one machine.
- The face recognition process compares a face that is being identified with all the known faces, resulting in very long operations.

1.3 Objectives

Our goals for this work were to develop a video face recognition system executed in a grid computing environment with the objective of improving its performance. The system is to be able to running in one or more clusters, leveraging existent grid middleware.

Specifically we have the objective of developing the following:

- A distributed system with better performance than a standalone application.
- A distributed systems that can scale to a large size.
- A distributed architecture of the system, incorporating existing stand-alone face recognition software.
- A data storage schema the distributed application, for a unified data access.
- A distributed coordination system, that manages the interdependencies between distributed elements.
- A scheduling algorithms approach that balances the load across the machines in the cluster.

1.4 Contributions

Bellow we list the main contribution that our work offers.

- Design and implement a distributed face recognition system, demonstrating it is feasible
- Develop a system that can be executed in multiple grids.
- Reduce as much as possible the grid requirements (control and storage) from grid to run the system.
- Reduce the memory requirements to perform the recognition process by not loading all known face repository into memory.
- Improve the search in a known face repository for a matching face when identifying a new face.

1.5 Document Roadmap

The rest of the document is organized as follows:

- Chapter 2 (Related Work): This chapter contains a related work study of the relevant literature concerning Grid Scheduling, Resource Discovery and Face Recognition, in this section we justify the decisions we made concerning the software we used in the system development.
- Chapter 3 (Architecture): In this chapter we present our solution, we present the system‘s distributed and software architecture, the resource broker architecture is also presents together with scheduling algorithm approach developed for the system. The file storage schema and distributed coordination is also described.
- Chapter 4 (Implementation): This chapter contains the implementation detail of the system; essentially the chapter explains was the system described in chapter 3 was implemented.
- Chapter 5 (Evaluation): This chapter contains the result from the tests done to the system and its evaluation
- Chapter 6 (Conclusion): This chapter contains a wrap-up of this work as well as a summary of the work done, finally it contains the future work.

Related Work 2

For the development of this work we studied literature from three areas:

- The **Grid Scheduling** in section 2.1 was studied because the system we had the objective of developing was to run in a Grid environment.
- Studding Grid Scheduling naturally led to the necessity of studying **Resource Discovery** systems (section 2.2) as the schedulers require such systems to work correctly.
- Finally we studied **Face Recognition** algorithms (section 2.3) as this work was about face recognition, we needed to understand and to choose what techniques and software were more adequate for purposes of this work.

Traditionally, the Related Work is a number of descriptions and comparison of various systems or solutions that are similar or solve similar problems to the one in development. In the case of this work however, there are no publicly available examples of system architectures or implementations of similar systems. Therefore, instead of describing similar systems we did a contextualization on the relevant areas for this work, and justify the choices made regarding what software to incorporate in our solution.

2.1 Grid Scheduing

Grid computing (Foster & Kesselman 1999) is a relatively successful type of multiprocessing infrastructure that has the objective of providing dependable, consistent and pervasive access to high-end computer resources (Dong & Akl 2006), it achieves this by **combining the computational power** of a high number of computers, usually for scientific computing and e-science tasks.

Any distributed system has to be controlled, and grids are no exception, these are **controlled by schedulers**, which acts as a resource brokers. Schedulers' function is to interact directly with the grid user that submits tasks to be executed, **select and assign the resources** appropriate to the submitted tasks using a predefined algorithm parameterized by information from the **Grid information Service**, another grid component responsible for the task of acquiring information about the grid resources.

In section 2.1.1 a description of some of the particular scheduling problems in grids is given, followed by a characterization of scheduling algorithms in sections 2.1.2 and 2.1.3. In section 2.1.4 a description of scheduler architectures is presented, followed by a list of some relevant examples of schedulers in the literature in section 2.1.5. Lastly an analysis of the whole section is done together with the selection of the appropriate scheduler for this work in section 2.1.6.

2.1.1 Scheduling Problems in Grids

Other branches of parallel computing like symmetric multi-processor machines or massively parallel processor computers operate under the assumption that schedulers have a single image of the whole system and control all resources, the resource pool is mostly invariable and the effect of adding new tasks can be contained through some policy (Foster & Kesselman 1999).

In Grids however, individual machines may have **different characteristics, different access delays** (connection may be done throw non-uniform links) and may be located in **different domains**. That means that it is very difficult for the scheduler to have a complete image of the grid, since grids in different domains may have local schedulers, which also mean that direct control over all resources is not possible.

Resources in grids are **volatile**, frequently, nodes become unavailable and other became available, also the load of each node cannot be determined, for instance a job scheduled to a certain grid may be interrupted by an internal grid job or by another external job that the local scheduler rated with a higher priority. So the performance of a grid is highly variable when compared with a multi-processor machine for example.

2.1.2 Scheduling Algorithms

Scheduling algorithms can be classified as Static or Dynamic, related with the time at which the scheduling decisions are made (Dong & Akl 2006), the following sections present a description of both types of algorithms.

Static Scheduling:

Scheduling decisions are made once, each task is assigned to a resource before the application is scheduled and that decision is **never changed** during its execution, information about all resources in the grid, as well as all tasks generated by applications, need to be known at scheduling time. This type of algorithms have the advantages of working with a **global system view** (information on all tasks and all resources are known at scheduling time), having a the **simplified tasks cost estimation** and having a **higher throughput** in general (less computation, less network delay). However, they have no **fault tolerance** (if a node fails, the application fails) and the **response time may vary** drastically if the selected resource is heavily loaded. These disadvantages may be mitigated using rescheduling mechanisms, but these reduce the performance advantage of these algorithms when compared with dynamic ones. Pegasus (Deelman et al. 2008) is an example of a static scheduler.

Dynamic Scheduling:

Scheduling decisions are made on the fly as applications are executed, meaning that the attribution of tasks to resources is done as the tasks are presented to be scheduled. It is especially useful when applications have **non-deterministic execution flows**, which is when the number and the nature of tasks cannot be determined prior to application execution. These types of algorithms have the advantages of **not needing to be aware of application execution flow**, providing **better load balancing** as each decision is made with updated information, and not in advance like in static algorithms, being **fault tolerant** (task can be rescheduled) and producing **consistent response times** (If the selected resource is heavily loaded, task can be returned to the scheduler). However, they have generally a **lower throughput** (more computation, more network delay) and the Scheduling tents to deliver **not the best execution times for each individual task**, they tend instead to maximize resource utilization. Condor (Litzkow et al. 1988; Tannenbaum et al. 2001) and Legion (Chapin et al. 1999) are examples of dynamic schedulers.

2.1.3 Scheduling Process

The scheduling algorithm itself is composed by resource selection and task selection; the following sections present a description of the most common algorithms for both of these components (Hamscher et al. 2000).

Resource Selection:

When selecting resources for a given task, schedulers normally have two lists, one with machines capable of executing the task to be scheduled and another, a subset of the first one with the machines that have enough free resources to execute the task immediately, in the work in (Hamscher et al. 2000), four selection strategies are presented.

- **Biggest First:** Selects the machine with the most free resources, essentially tries to attribute the task to the machine that will be able to execute it with the highest performance, a disadvantage of this is that very demanding tasks may be delayed due to powerful machines being inundated with small tasks.
- **Random:** Selects a machine from one of the list at random, which on average provides a fair and balanced way to distribute resources to tasks.
- **Best Fit:** Selects the machine from one of the lists that if assigned with the task will leave the least amount of free resources, essentially this strategy will try to fully utilize machines, attributing small tasks to machines with few free resources. When compared with Biggest First, that will assign tasks to the machine with the most free resources, this strategy is less likely to impose delays on demanding tasks.

- **Equal Util:** Selects the machine with the lowest resource utilization, with the objective of promoting load balancing on all machines.

It is important to note that the list of techniques above are general purpose strategies that work in scenarios where the only aspect taken into account is whether or not a machine has the necessary resources to execute a task, other strategies may involve other factors, such as the cost of executing tasks in a given machine, Nimrod/G (Buyya et al. 2000) is an example of a scheduler that selects machines to assign tasks depending on the cost of using them.

Task Selection:

When selecting tasks to be scheduled the most common strategies are based on list-scheduling, the work in (Krallmann et al. 1999) describes four of these strategies.

- **First-Come-First-Served:** Tasks are ordered by their arrival time, greedy list scheduling is used, meaning that the oldest task is scheduled as soon as enough resources become available, the rest of the tasks are not scheduled as long as the older tasks are not. This strategy has the advantage of producing fair schedules, not leaving behind or privileging any task, but has the disadvantage of causing a large percentage of machines to be idling for a long time if, for instance, the oldest task is high resource demanding and the rest of the tasks are not; they could be assigned to machines capable of running them but not the oldest one.
- **Random:** Tasks to be scheduled are selected at random, has the same issues as First-Come-First-Served, as high resource demanding tasks can cause large periods of idle time in machines, also it may cause starvation when selecting tasks randomly under heavy load there is no guarantee that each task will ever be selected.
- **Backfilling:** Similar to First-Come-First-Served with the difference that when a high resource demanding job is not scheduled, the scheduler tries to prevent idle time on machines by scheduling other jobs for which there are machines with enough resources. To prevent starvation of high demanding tasks, the scheduler requires the specification of a time estimate for the execution of each task, it will only assign a task to resources that will finish before that resource could be used to execute the high demanding task. For instance, if the oldest task is waiting for resources and a machine with enough resources is running another task that will finish in five minutes, the scheduler will not assign a low resource demanding task to that machine if it is predicted to last for six minutes, otherwise it could cause starvation to high demanding tasks (Feitelson & Weil 1998).
- **SMART:** Also requires execution times estimates, but does not respect task submission order, it groups tasks according to their execution time, each group ceiling is defined by a geometric sequence based on a parameter, which can be used to tune the load balance, all tasks in a group are assign to "shelves", essentially tasks are grouped so that all can run in parallel, after this, the shelves are ordered using Smith's rule (Turek et al. 1994) (sum of all execution times in a shelf

divided by the higher execution time of any task) , then the shelves with the highest values are scheduled first.

2.1.4 Scheduler Architecture

Schedulers can control grids and be positioned in relation to the resources in three ways, Centralized, Hierarchical and Decentralized (Krauter et al. 2002), the following sections present descriptions of these Scheduler Architectures.

Centralized:

In this architecture, **only one scheduler** exists, it is responsible for assigning all tasks to all resources. Tasks submitted that cannot be scheduled immediately stay in a queue in the scheduler until enough resources are free. All information required by the scheduling algorithm must be kept by the scheduler (it is not required that the scheduler it-self stores the information but at scheduling time the information has to be transferred to it if stored elsewhere). This type of schedulers does not scale well as they can become a **bottleneck**, also it has **no fault tolerance**, if the scheduler or its connection to the network fails, system availability is compromised. But they are theoretically able to produce **better schedules** since they have all information and control all resources. Condor is a classical example of such schedulers, although the actual scheduler is distributed, the scheduling decisions are done by a centralized resource broker.

Hierarchical:

More than one scheduler is present in a hierarchical architecture, tasks are submitted to a central scheduler, but that scheduler does not assign task to resources directly; instead, it **submits tasks to schedulers down the hierarchy**, until the task reaches a **local scheduler that will actually schedule the task to a machine**. In these schedulers, detailed information is kept by the lower hierarchy schedulers and the above ones have a summary of the resources available. This organization **mitigates the scalability issue of centralized architectures** since each scheduler has to make decisions about a smaller set of resources. **Fault tolerance is partially provided**, as long as the central scheduler does not fail, operability of the system is assured and the central one can have backup replicas. Legion (Chapin et al. 1999) in an example of a hierarchical scheduler.

Decentralized:

In a decentralized architecture, **no central scheduler** is present, instead a number of schedulers assign tasks to resources relatively independently and share information about their resources between themselves, allowing schedulers to send tasks to other schedulers. This architecture does not have the bottleneck caused by a central scheduler, therefore it is scalable and fault tolerant since only the failure of all schedulers will imply a system failure. The disadvantage of this type of schedulers is that each scheduler **is not able to produce an optimal assignment of tasks to resources** due to lack of information; coordination in resource discovery and allocation between schedulers (information sharing) may improve the optimality of the schedules but it will impose overhead and reduce the performance, therefore these scheduler normally work with incomplete information. MetaComputing Online (Gehring & Streit 2000) is an example of this type of schedulers.

2.1.5 Relevant Examples of Schedulers in the Literature

In this Section, examples of grid resource management system (RMS) are presented together with a description of the respective schedulers and their classification according to the definitions made previously.

Globus:

Globus (Foster 2006) is a grid resource toolkit, it does not have a particular interesting scheduler, in fact it is designed to have external schedulers, for instance Nimrod and Condor; there are versions of both of these schedulers integrated with Globus, Nimrod/G and Condor-G respectively. It is however an interesting system when it comes to support distributed execution across multiple domains. Globus has a number of components that can be used to build a grid network, for instance security, resource location, resource reservation, data transfer and remote execution calls.

Condor:

Condor (Litzkow et al. 1988; Tannenbaum et al. 2001) is a RMS originally designed to leverage the computational power of idle workstations and make it available to those who have higher computational needs than a single workstation can provide, but it can be used in a conventional grid, the only difference is the idle monitor is not present. Condor's scheduler is classified in the literature as **centralized**, but in reality it **works more like a decentralized or hierarchical one**, there is a **central coordinator**running in one of the machines and a **local scheduler** on each machine, local schedulers are the ones that do the actual scheduling, to avoid a large overhead of messages, local schedulers only communicate resource availability and need to the central coordinator that makes the decisions to attribute resources available in a given machine to a another machine that requires them. Condor ensures proper distribution of resources to all users giving higher priority to machines that require resources less often, this is done be associating a “scheduling index” to each machine, every time a machine is granted the resources from another, its index is incremented, every time a machine requests are not granted resources, the index is decreased. In each scheduling iteration, the coordinator attributes the available resources to the machines with the lowest index.

Condor-G:

Condor-G (Thain et al. 2005) is a RMS designed to allow users to **leverage computational resources in heterogeneous domains** as if they all belong to the same domain; it is basically constituted by the original **Condor scheduler** and the **grid resource toolkit Globus**. Its objective is achieved via the separation of concerns between remote execution and computation resources management; remote execution is handled by the Globus toolkit and computation management is handled by the Condor

Scheduler. In detail, Condor-G uses GASS (Bester et al. 1999) (Global Access to Secure Storage) to implement the tasks I/O and GRAM (Czajkowski et al. 1998) (Grid Resource Allocation Management) to submit and monitor tasks in remote machines, both GASS and GRAM are components of Globus.

Condor-G works similarly to Condor as far as the Scheduler Architecture is concerned, so it is officially a centralized scheduler but where the actual scheduling is done by distributed schedulers. To each user that submits tasks to be executed in the grid, a grid manager will be created, these can be seen as the local schedulers present in the original Condor and these are the ones that will actually schedule tasks to be executed in the machines controlled by the Globus tool-kit and also similarly to the original Condor a central coordinator (here called resource broker) assigns resources to each grid manager.

Pegasus:

Pegasus (Deelman et al. 2008) is a RMS designed to execute applications over various heterogeneous computer grid sites. It takes abstract workflows describing applications and finds the appropriate data and grid resources to execute them.

Workflows are direct acyclic graphs whose elements are activities and the connectors are dependencies between activities; applications are described by workflows, where activities represent individual application components (software components or logical execution steps) and the dependencies represent control and data flow between those components. These workflows are described in an abstract form making no reference to the grid resources that will be used to execute them.

Pegasus has a **centralized architecture**, users define abstract workflows prior to execution and the system builds concrete workflows containing the association between the activities and grid resources, the latter workflows will ultimately be executed; it is therefore a **static scheduler**, since the scheduling decision are made for the whole application and prior to application deployment. Pegasus uses three criteria for scheduling: file placement, code placement and computational resources, it uses tree different Information System to acquire this information.

Pegasus does not control the actual execution of a workflow, instead it uses DAGMan (Team 2012) which is part of the Condor project, DAGMan parameterizes the Condor scheduler to execute the workflow.

Pegasus is also available with a **dynamic scheduler**, it works in a similar way, but instead of generating a complete workflow all at once, the workflow is partitioned and scheduled dynamically.

Legion:

Legion (Chapin et al. 1999) is an object based RMS whose scheduler has a hierarchical architecture; every component is an object, these objects are organized in a hierarchy and are created and managed by the corresponding class and metaclass.

The architecture of the scheduler is hierarchical since the scheduling decision is not made by a single scheduler or by a variable number of seemly independent ones, the scheduling is done at various levels throughout the hierarchy, the various essential components are presented next:

- **Host and Vault:** Lowest rank objects in the hierarchy, Hosts interact directly with machines, collect the machines' status information and are ultimately responsible for deciding the attribution of a task to a machine. Vaults represent tasks, they contain all the information about a task, and they are used for task migration and/or recovery.
- **Tasks:** Representation of task a client requested to be executed in the grid.
- **Scheduler:** Responsible for creating the mapping between task objects and host objects, effectively assigning tasks to resources, it creates not one but a number of possible mappings.
- **Enactor:** Negotiator that tries to execute one of the Scheduler's mapings, it does this by requesting reservations to Hosts, that as stated before have the ultimate responsibility of scheduling tasks, a given schedule is only executed if its entire mapping of Tasks to Hosts is possible to execute.

MetaComputing Online:

MetaComputing Online (Gehring & Streit 2000) is a RMS designed to provide access to grid resources in heterogeneous sites, it differentiates itself form other RMSs, by allowing resources not only to be seen as hardware able to execute tasks, but also as services that perform tasks.

The scheduler has a **decentralized architecture**, a collection of machines is abstracted by a Central Node, normally only one per site. These modules encompass all the information about the resources they abstract, and act as gate keepers in order to allow or deny access to the actual resources. Information must be exchanged among these modules. For this purpose, the system uses a transactional based protocol over **shared memory**, shared objects and backups are kept in every Central Node.

Being a fully decentralized scheduler, it is fault tolerant, the failure of one module does not affect the access to any others, this also means it is a very flexible system, since modules can leave and join at any time.

2.1.6 Analysis

Scheduler	**Scheduling Algorithm**	**Scheduler Architecture**
Condor	Dynamic	Hybrid Centralized / Decentralized
Condor-G	Dynamic	Hybrid Centralized / Decentralized
Pegasus	Static / Dynamic	Centralized
Legion	Dynamic	Hierarchical
MetaComputingOnline	Dynamic	Decentralized

Table 2.1: Summary of Schedulers Classification

Table 2.1 contains examples of schedulers with their respective classification, regarding their scheduling algorithm and architecture.

From the analysis of previous sections we are able to conclude that there is no perfect scheduler for all types of applications; all the alternatives presented in the previous sections have advantages and disadvantages, it is therefore important to understand the problem this work will try to solve in order to make the correct scheduler selection.

From the nature of the application, performing face recognition in videos arriving at any given time and answering queries to retrieve information about the faces identified on the videos submitted earlier, it is clear that static scheduling would not be appropriate, since no prediction about the amount of tasks can be made, not even about the nature of those tasks, for instance it is not possible to know the length of the videos that will be submitted, therefore **dynamic scheduling** was adopted since static scheduling is not adequate to unpredictable scenarios.

The selection of the scheduler architecture is not straightforward. A centralized scheduler would be able to provide optimized schedules, but would easily become the bottleneck of the whole system and provide poor fault tolerance; a hierarchical scheduler would mitigate these problems but would not eliminated them; finally a distributed scheduler would provide good fault tolerance and scalability but would more difficultly produce optimal schedules, also it would require sophisticated coordination between schedulers which increases complexity and decreases performance.

For the development of our system we choose Condor for the scheduler, having dynamic scheduling it is appropriate for our purposes. And the fact that it is a **centralized scheduler** means that it is able to produce optimal schedules; it also means that scalability is limited, however as the execution time of a task, for instance, detection of a number of faces in a video is certainly measured in minutes and a scheduling decision time is measured in seconds in a worst case scenario, we are able to classify the bottleneck effect as minimal (the bottleneck effect is only relevant is the scheduler is not able to dispatch tasks to resources faster than the resources can execute them).

Condor only controls one grid site but the system works in more than one, a first approach to a solution for this problem would be to use a hierarchical scheduler, for instance we could have used Condor-G as a central scheduler that would decide in which site a task would be executed and all sites would be used in the same way. However due to the characteristics of the system, specially the fact that its IO demands are relatively high, it is useful to prevent the overhead of transferring data to outside the site (normally the network throughput within a site is much higher than between sites). With this idea in mind we distinguish the main site where the system will run until the all resources are used from the rest of the sites that are used as a backup when the main site capacity is all used. In this scenario, the additional overhead of having a scheduler only to decide which site to send a task is not justified, instead we use a trigger that redirects the tasks to other sites if the main site is overloaded.

2.2 Resource Discovery

Resource discovery is a vital component of a distributed systems, in a grid a scheduler would not be able to work if not informed about the resources availability and their detailed characteristics, static information (e.g. CPU architecture and OS version) and dynamic information (e.g. CPU load and free memory). Resource discovery systems enable one of the core objectives of a grid, create associations between tasks that need resources to be executed, and machines which have those resources and are able to execute tasks.

In section 2.2.1, resource discovery systems (RDS) are classified; next, in section 2.2.2, a list of some relevant examples of resource discovery systems in the literature is presented, lastly an analysis of the whole section is done together with the selection of the appropriate resource discovery technology for this work in section 2.2.3.

2.2.1 Resource Discovery Process

Resource discovery can be seen as two processes, **resource discovery** and **resource dissemination**, which are, an application trying to find resources and a resource advertising its availability respectively. The work in (Krauter et al. 2002) provides the appropriate classification for both processes:

Resource Discovery:

Can be done using **queries** or **agents**; the major difference between these two is where the resource discovery process takes place. While in resource discovery based in queries, a given scheduler that needs to obtain information, will ask an external directory system which contains all the resource information; in agent based resource discovery machines exchange information by sending active code fragments across the network which are interpreted on every reached machine, agent approach is used in systems where all nodes have global information and/or a directory service is not present. Query based resource discovery systems can be further classified as centralized, hierarchical and decentralized according to how the information database is accessed, the advantages and disadvantages of these different approaches are similar to the ones presented in the description of scheduler architectures. MDS (Fitzgerald et al. 1997; Schopf et al. 2006) is an example of a resource discovery system based in queries and Bond (Boloni & Marinescu 2000) is an example of a RMS whose resource discovery system is based in agents.

Resource Dissemination:

Can be classified in two categories, **batch/periodic** and **on demand** according to when the resources disclose information about their status. In a periodic approach, information is updated in predefined time intervals, that way the resource information is maintained softly consistent (changes in availability of resources are not immediately propagated to the resources discovery system), but the system uses a relatively low amount of communication. In an on demand approach, updates are done immediately after resource status changes; the information is always consistent but it has a relatively high overhead in communication. MDS is an example of a RDS with periodic resource dissemination and R-GMA (Cooke et al. 2003) is an example of a RDS with on-demand resource dissemination.

2.2.2 Relevant Examples of Resource Discovery Systems in the Literature

In this section, examples of resource discovery systems are presented and classified according to the definitions made previously.

Bond:

Bond (Boloni & Marinescu 2000) is an agent based RMS. Its resource discovery system, also based in agents, works without the need for a directory service. Agents have a global view of the resources available, which is kept updated by periodically (**periodic resource dissemination**) sending active code fragments across the network; for this purpose Bond agents use the **knowledge querying and manipulation language** (KQML) (Finin et al. 1994). Bond standard is to work in a flat organization, no hierarchical levels or multiple grid sites are supported.

R-GMA:

R-GMA (Cooke et al. 2003) is an implementation of the Grid Monitoring Architecture proposed in (Tierney et al. 2002), supported by a relational database. This system defines three types of entities in a grid, **producers** (sensors that collect information about resources), **consumers** (users of the information collected by the producers, e.g. resource brokers) and a **directory service** (allows consumers to be aware of producer's existence and the information they collect).

The system has a **centralized architecture**, with the directory service as the central node; producers register their identity and information on the relational database of the directory service and consumers query this database. At this point, it is important to refer that R-GMA manages different types of information in different ways. Static information or information that does not change frequently is stored in the centralized directory service and is used in consumers' lookups, and dynamic information is obtained by the consumers directly from the producers, after their identification is provided by the directory service. This separation is not rigid as it could impair the capability of resource discovery, dynamic information vital to producer identification can be stored in the central database.

MDS:

MDS (Fitzgerald et al. 1997; Schopf et al. 2006) is a grid **resource discovery and monitoring system** based in queries, it is part of the Globus toolkit. Grid components and their detailed information are represented by **objects composed of type-value pairs**, these objects are organized and stored in a **hierarchical structure** called Directory Information Tree, this structure translates the relations between the various components, for instance a machine that has a network card that connects the machine to a local network is represented by three objects and the respective associations.

The system has a **hierarchical architecture**, composed of two types of modules, **producers** and **republishers**. Producers can be seen as the lowest hierarchical workers, they collect information directly from the computational resources via scripts or modules using well defined APIs, these modules compose the **Grid Resource Information System** (GRIS). Republishers are the higher hierarchical workers, they collect information from producers or from republishers lower in the hierarchy, these modules compose the **Grid Index Information Service** (GIIS), and their function is to preprocess resource information in a useful and customizable manner; for instance one republisher can index resources by their operative system or by their available memory while other can index the resources following a completely different schema. This means that specialized republishers can be used in resource brokering, easing the scheduling task, also republishers may hide details to the next one in the hierarchy making this a very versatile system.

Communication between modules is done using two protocols, **Grid Information Protocol** (GRIP) used by republishers to address queries to producers or other republishers and **Grid Registration Protocol** (GRRP) used by producers and republishers to register thenselves in republishers.

The system is based on **periodic resource dissemination**, this types of systems are characterized by an implicit delay between resource status change and the RDS users becoming aware of that change, this delay is particularly significant when the system is composed by many levels of republishers; to diminish this limitation, users are not limited to top level republishes, information can be obtained from any module on the hierarchy, even directly from producers.

ClassAd:

ClassAd (Raman et al. 1998; Coleman et al. 2003), is a **resource discovery system** based in **matchmaking**, it is part of the Condor system; it is used to make brokering decisions. One of the most distinctive features of ClassAd is that both providers and requesters of resources have a standard and straight forward way of **expressing restrictions** to the match between requesters and providers, unlike most systems where providers only advertise their characteristics and requesters ask the resource discovery system to find an appropriate provider.

The system has a **centralized architecture**, providers send ClassAd messages to a matchmaker describing their characteristics and restrictions; requesters also send ClassAd messages to the matchmaker describing a task waiting to be scheduled for execution, both have similar structure and are composed by a variable number of attribute names and their respective expressions, the matchmaker may not have any standard rules for the matching algorithm. Instead, the matching rules are defined in two special attributes contained in the classAds, the Constraint attribute and the Rank attribute, the first translates the compatibility between requesters and providers, and the second one is used to choose a matched pair when more than one match occurs, these attributes are in fact functions, that can be defined using the other attributes (e.g. other.ram>512 translates the condition that a task requires a machine with at least 512 MB of ram).

ClassAd does not define a resource dissemination policy, it can work with both periodic and on-demand approaches, however when used in Condor, **periodic resource dissemination** is used. The system itself works periodically, classAds are received, stored and periodically the matchmaking algorithm will run, produce the matches and inform both the requester and provider, the actions resulting from that information are not of the concern of the resource discovery system.

D-Dimensional DHT:

The work in (Ranjan et al. 2007) presents a **decentralized resource discovery system**, the system is based on distributed hash tables; it was developed to address the limitations of centralized and hierarchical systems, limited scalability, reduced fault-tolerance and poor network load distribution, it is especially useful for interconnection of a large number of grid sites where systems with non-decentralized architectures would not be able to operate.

In computational environments composed by a large number of sites, normally each site has a **gate-keeper** or controlling node, each one of these nodes collects information about the resources within its grid site and performs resource discovery in the other grid sites. That means that these nodes need to process two types of queries among each other, **Request Lookup Query (RLQ)** and **Request Update Query (RUQ)**, not having a centralized service means that such queries must be processed in a decentralized manner, information must be shared between all controlling nodes which traditionally would mean a high volume of update messages so that all could process the lookup queries correctly. Instead, a **distributed hash table** is used to maintain the resource information and to support the answering of queries, DHT naturally provide associations between keys and values, in this application keys are specific attributes (e.g. amount of ram) and the values are the machines that have those attributes.

Although DHTs address the limitations of centralized and hierarchical systems, such as poor scalability, they are limited to **single pair associations**, for instance a DHT would only hold information about the amount of ram on each machine and it would only be possible to retrieve the information about which machines have a given amount of ram; that way, in a normal query where constraints may be imposed on various attributes, the system would have to retrieve information from various DHTs, reducing the performance drastically. The solution for this problem is the use of **d-dimensional indexing**, where the index used in the DHT is a concatenation of all indexes. Another limitation remains however, queries are normally done specifying intervals such as machines with a given resource equal or superior to a certain amount, this would require that all nodes containing information about machines with resources within that interval to be inquired, which would also decrease performance. To minimize this limitation the indexing of the DHT is actually done by range of index values, so that fewer machines have to be contacted when the system processes a range query.

2.2.3 Analysis

RDS	Resource Discovery	Resource Dissemination
Bond	Agents	Periodic
R-GMA	Centralized Queries	Hybrid Periodic / On Demand
MDS	Hierarchical Queries	Periodic
ClassAd	Centralized Queries	Any (in Condor is Periodic)
D-Dimensional DHT	Decentralized Queries	Any

Table 2.2: Summary of RDS Classification

Table 2.2 contains some examples of Resource Discovery Systems with their respective classification, regarding the resource discovery and dissemination.

The decision about the RDS to choose for the system must take into consideration the fact that the system runs in multiple grid sites, and that in each site the chosen scheduler (Condor) controls the machines. Condor is considered a centralized scheduler but is in fact composed by multiple schedulers, one for each machine. What is truly centralized is the **resource brokering** and **schedulers coordination**, in Condor this is done by the **ClassAd Matchmaker**. Therefore ClassAd was adopted, it is present in each site and works exclusively for the Condor instances in the respective site.

ClassAd requires an external system (script or application that generates ClassAd advertisements) to provide the dynamic information it needs, also the trigger system mentioned in the Grid Scheduling chapter equally needs resource information. These two modules need information with different levels of abstraction ClassAd needs detailed information on the machines' status and the trigger needs overall site information (a figure of the load in all machines).

MDS could serve the purpose of the external application that would provide the dynamic information; as it is a hierarchical information system which naturally allow for different views of the information at different levels of the hierarchy. However, the resource dissemination in MDS is periodic, which imposes delays in information updates. Although these delays can be tolerated in information such as CPU and memory usage, they certainly cannot exit in file placement information where delays would mean that the information would simply be unavailable at scheduling time. We chose to develop an application similar in architecture to MDS, but to which the file placement update can be sent on demand.

2.3 Face Recognition

Face Recognition has the objective of identifying or verifying the presence of faces in a given image. It has received significant attention from the research community, significant advances have been achieved already and it has a large number of applications such as entertainment (video games and virtual reality), security (parental control and building access) and law enforcement and surveillance (suspect identification) (Hjelmås & Low 2001).

Face Recognition is divided into 3 main steps (Hjelmås & Low 2001):

1. Face Detection
2. Feature Extraction and Normalization.
3. Identification or Verification.

Note that step 3 is commonly known as Face Recognition, however in the interest of clarity we will refer to it as Identification or Verification.

Steps one and two may be done simultaneously, both detection and identification techniques can be divided into two categories, featured based and image based, the following sections 2.3.1 and 2.3.2 explain these in detail, specific techniques and algorithms for video based face recognition are described in section 2.3.3. Lastly, an analysis of the whole section is done together with the selection of the appropriate techniques for this work in section 2.3.4.

2.3.1 Face Detection / Feature Extraction

It is the first step in the Face Recognition Processes, it takes the raw input image and outputs a smaller image containing a face present in the inputted image, concurrently, feature extraction and image normalization (size-wise is the more common normalization) can occur since the face detection process either relies on feature detection, or in the case of some image-based techniques, feature extraction is not done. This is a vital step for the performance and feasibility of the following steps as the identification algorithms will not performed well if faces are not accurately detected (Hjelmås & Low 2001). Face detection techniques are divided into two great areas (Zhao et al. 2003):

Feature Based:

This makes use of explicit face knowledge, such as skin color and face geometry, these are low level methods. These may be composed by three increasingly sophisticated methods that complement each other.

The first one is a **low level analysis**, based on edges, gray levels, color and symmetry. These were the first techniques used in face detection, essentially they try to find lines and angles that may belong to a face or/and try to match clustered colored pixels with skin color. The work in (Wu & Ai 2008) is an example of such technique, in this case based on color. From the previous description it is evident that low level analysis is prone to a large number of incorrect face detections, objects in the background can have similar color and lines characteristics to a face.

A second method is used to filter the results from the first one, here two techniques can be applied. **Feature searching** which the work in (Hjelmas & Wroldsen 1999) uses, tries to validate features by their size and relative position, for instance a pair of dark areas (eyes) at a certain distance may indicate the existence of a face, distance between the eyes and mouth may also be used to validate a face detected.

However this technique is too rigid and based on heuristic information, therefore it may fail if presented with face pose variations and complex backgrounds. **Constellation analysis** used in the work in (Burl et al. 1995) tries to do the same as feature searching but resorting to a probabilistic model for the spatial arrangement of facial features, this technique can tolerate missing features and relatively high face rotations. Haar-like Features (Viola & Jones 2004) is a particular successful example of feature based face detection system. In this particular system, an image is partitioned in boxes and each box has a value defined by the sum of the pixel intensities; a set of rules is than applied in order to determine whether or not the values of the boxes indicate a possible face region. These rules are applied in layers using what the authors call cascade classifiers, the algorithm starts by identifying edges for example, then in those regions it tries to identify lines and so forward, until it is able to conclude a certain images zone is a face.

The most sophisticated of these algorithms that try to validate features is based on **active shapes** that take the form of image features, **snakes** (Kass et al. 1988) is an example of these algorithms; it uses an energy-minimizing function as guidance to adjust itself toward lines and edges assuming their form.

Image Based:

Makes use of the pattern recognition theory, addresses face recognition as a regular pattern recognition problem, the image is analyzed as a whole and the face knowledge is not explicit but is contained in the training schemas for this kind of techniques. These Techniques have a clear advantage over the feature based ones, they do not depend on explicit face knowledge, eliminating the potential errors or incompletions of knowledge; moreover it has better performance in multiple faces detection and clustered background situations. These can be divided into three types.

One of them is **linear subspace methods** (Mohan & Sudha 2009), which makes use of eigenvectors in face representation. The work in (Turk & Pentland 1991) introduced a technique to detect faces using this representation, from a training set, a profile eigenface is built; this result is called the ”face space”. Face detection is then achieved using the “distance-from-face-space”, an error resulting from the calculation of the eigenvector that is used to determine a given image “faceness”. Another method is **neural networks**, which is largely used in pattern recognition problems, can also be used for face detection, the work in (Lin et al. 1997) is an example of such system. Finally there are statistical approaches based in information theory like Bayes decision rules (Meng et al. 2000).

2.3.2 Identification or Verification

This is the second step in the Face Recognition process, here face images outputted by the first step are analyzed and identified, similarly to face detection the identification also has feature and image based techniques, in addition it has Hybrid Methods.

Feature Based:

As previously stated, feature based algorithms make use of explicit face knowledge, most commonly eyes, nose and mouth position and form are used to perform the identification (Hjelmås & Low 2001). Examples of algorithms of this category are (Cox et al. 1996) and (Mian et al. 2008), in the last one features are represented as nodes in a graph and the face identity is represented by the Euclidian distance between the nodes, the recognition is then performed by graph matching.

Image Based:

Similarly to face detection, image based algorithms treat face identification as a general recognition problem, process the image as a whole, not giving any particular relevance to local features such as eyes and mouth, also they have a high resilience to partially occulted faces and crowded background situations. The work in (Agarwal et al. 2010) is an example of such a system, as described before an image is transformed in eigenvectors, values that represent the identity of a face, face recognition is then done by comparation of the values of a face to identify with the values of know faces.

Hybrid Methods:

Humans use both feature and image based techniques when recognizing faces (Hjelmås & Low 2001), so a natural evolution for face recognition systems would be to adopt both techniques, improving therefore their performance, in the work discribed in (Pentland et al. 1994) for instance, eigenvectors are used for the global image as in a normal image based algorithm, but they are complemented by eigenfeatures which are the application of eigenvectors to local features like eyes and mouth. In addition, they also make use of various images for the same person, with different angles and expressions, that allows for better recognition rates and improved resilience to variations in different images of the same person.

2.3.3 Video Face Recognition

The previous section describes face recognition methods for still images, but video face recognition has a few particularities worth mentioning.

Challenges of Video Face Recognition:

The face images extracted from a video sequence are often of much **lower quality** when compared with the still images normally used for face recognition, video acquisition may happen outdoors or even indoors with bad conditions for video capture; illumination, visibility and the angles or poses of captured subjects may not be the ideal. Also, face images are **small**, video face regions may be as small as 15 x 15 pixels in contrast with 128 x 128 pixels of a still image (Hjelmås & Low 2001), the reduced dimensions will difficult the face detection and therefore impair the Identification which depends on accurate face detection.

Face Detection in Video Sequences:

The same techniques used for still image face detection can be applied to video sequences; each frame can be treated as a still image and the techniques may be applied directly, however it is possible to make use of motion to indicate the presence of faces. The work in (Graf et al. 1996) makes use of **frame differentiation** to detect faces and other body parts, it does this by calculating the differences between a number of frames and determining the areas where accumulated differences reach a certain value (threshold). The work in (Low & Ibrahim 1997) uses the same technique to locate and extract face feature like mouth, nose and eyes.

Identification or Verification in Video Sequences:

Again, the same techniques used for still image face identification can be applied to video sequences, but the fact that many similar images of a face are present (large number of frames contain similar images of a subject's face), allows for improvements over these methods, the problem of partial **face occlusion** may be mitigated by synthesizing a virtual frontal view from the large amount of different posed and angles present in the video frames. Another technique that improves face recognition is representing an identity not only by one image but by many, here **voting based recognition** is possible (Hjelmås & Low 2001), where essentially an image to be identified is not only compared with one already identified image but with many images from the same person, each with a different pose, angle or illumination, that way improving the accuracy of the Identification.

2.3.4 Analyses

It was not the objective of this work to develop a novel face recognition algorithm, nor was its actual performance of grate importance, we simply had the objective of improving the performance of an existing face recognition software executing it in a grid environment.

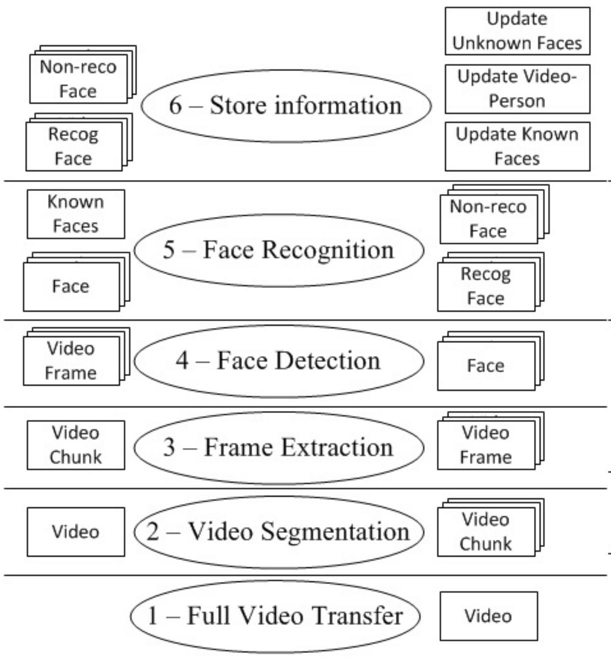
Nevertheless, we chose the software that seemed to perform better in video base face recognition, remembering that face recognition in videos normally means that the system has to work with low quality images, reduced size images and crowded background.

We chose **Haar-like Features** for the face detection process, although it is a feature based algorithm, which theoretically performs worse than a image based algorithm, the fact that it uses a list of rigorously studied and defined rules to perform the detection will assure the same or better results, with the advantage of having better performance and with the additional benefit of not being necessary to train the system before the execution. For the face recognition process we chose **Eigenfaces** which produce better results in low quality images, common in videos.

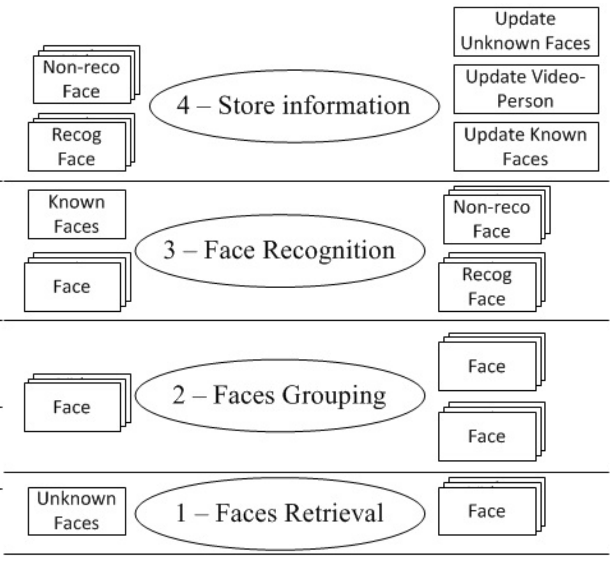
Architecture 3

In this chapter, the architecture of the system is detailed, first we specify the application division in individual tasks for each use case in section 3.1, next we detail the system's distributed architecture in section 3.2, followed by the software architecture including the more relevant middleware used to support the architecture in section 3.3 and we conclude the architectures with the resource broker in section 3.4. Lastly we describe the scheduling approach that is used in the system, its file storage schema and distributed coordination in sections 3.5 to 3.7.

3.1 Use Cases



(a) New Video Submission.



(b) Unknown Faces Reiteration.

Figure 3.1: Use Cases.

The diagram in Figure 3.1 refers to the more relevant elements of the system that are now explained: **Video** is a complete video file, **Video Chunk** is part of a video file,**Video Frame** is a still image from a Video, **Face** is a still image containing an isolated face, **Known Faces** is a repository of the mapping between Faces and persons' names used in the face recognition process, **Recog Face** and **Non-reco Face** are respectively a Face identified and still not identified in the recognition process, **Video-Person** is a repository of the mapping between videos and persons' names, and **Unknown Faces** is a repository of Faces not yet identified.

Figure 3.1a) describes the main use case; the reception of a new video file, the execution of this use case is divided into six stages. **Stage 1** is the file transfer between a client application and the system; when completed the system has a complete video file to be processed. **Stage 2** is the segmentation of the video, since the application runs in a distributed system, the bulk of processing a video files has to be shared by a number of machines; here two options could have been adopted, either machines would process entire files or smaller portions, the first option would difficultly achieve good load balance, therefore the second was adopted. Another alternative would be to distribute the actual algorithm, but in the case of a multi-data-single-instruction the division of data produces better results. **Stage 3** is the frame extraction, necessary since the face detection software works with frames. **Stage 4** is the first stage where the actual face recognition takes place, in this one, face images are extracted from the video frames. **Stage 5** is the second and final stage of face recognition; it produces a list of Recog Faces and Non-reco Faces. **Stage 6** is the final stage of this process; the various repositories are updated in this stage.

Figure 3.1b) describes the reiteration of the Unknown Faces from previous iterations; these Faces are kept in a repository. **Stage 1** is the retrieval of these faces from the repository. **Stage 2** is the grouping of these faces, for the same reason video files are segmented, the list of faces also have to be divided into groups. **Stages 3 and 4** are the same as stages 4 and 5 in the reception of a new video file.

The system has two more use cases, not represented by images due their simplicity: those are the system queries and the face identification or rejection of query results. The first one is the answering of a question such as, what persons are present in a given video, for that purpose the video-person repository is used. The second one is the users providing the identification for a given face or informing the system that a stored identification is wrong, this updates the various repositories.

3.2 Distributed Architecture

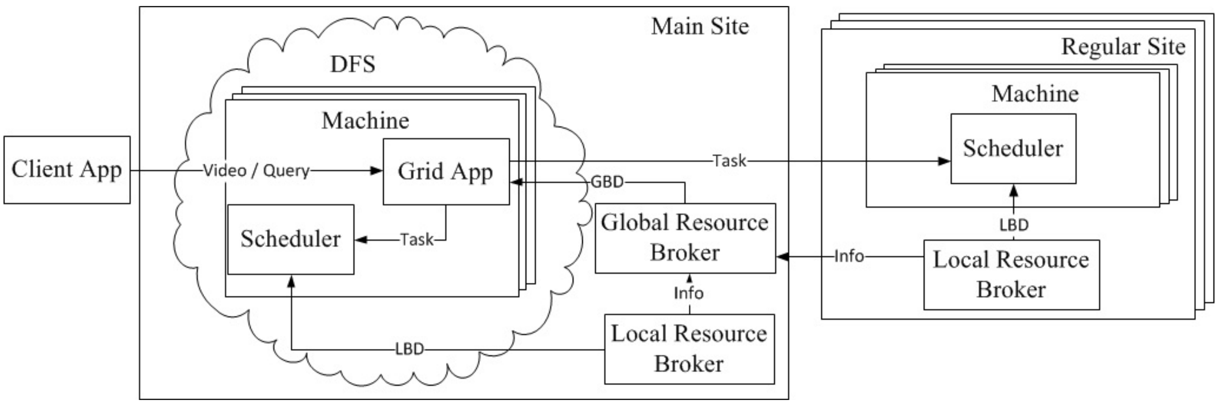


Figure 3.2: Distributed Architecture.

Figure 3.2, shows the system's distributed architecture, the arrows of the resource brokers mean: **GBD** - global brokering decision, **LBD** - local brokering decision and **Info** - information flux between resource brokers.

In the architecture, we distinguish two types of cluster, the **main site** and the **regular sites**. The main site is where normally the whole application execution takes place and where all persistent data is kept. The decision of maintaining all data in the main site is justified by the reduction of software and control demands on other clusters (clusters to which we may have limited access), it is easier to gain access to a cluster if we only require the execution self-contained applications that don't require direct control over the infrastructure. Also as the face recognition in videos is a IO intensive application, data transfer in low bandwidth networks (which the connection between two different clusters normally is) is to avoid as much as possible. Since all persistent data is kept in the main site, it makes sense to execute the application always in the main site; however computational power is not infinite, therefore to scale the system, there is the option of sending tasks to other clusters.

The machines on the **main site** make part of a **Distributed File System**, where the repositories, videos files and intermediate files are stored. Each machine has a **Scheduler**, a **Grid Application** and the **Recognition Software** (not represented in the figure). The Scheduler makes decisions about in which machine of the site to execute tasks, the Grid Application which serves as a gatekeeper interacts with the Client Application (user interface) and submits tasks to the scheduler, submitting to the local machine scheduler or a scheduler in a machine of another cluster effectively makes the decision of in which cluster to execute tasks. To perform these decisions and maintain consistency both the Schedulers and Grid Applications rely on resource brokers, systems that make the actual decisions.

The machines in the **regular sites** only have a **Scheduler** and the **Recognition Software** as those are just worker machines to which tasks can be sent from the main site. The Schedulers the regular sites also rely on a resource broker which works in a similar way to the one present in the main site. Note that the architecture described here also include components to control the regular sites, which seems to go against the previous description, they are in fact placeholders for any other grid control system. They were used in this work but as they are loosely connected to the components of the main site, they can be easily replaced by adapters that send tasks to other types of schedulers and translate the information from other resource discovery systems.

3.3 Software Architecture

Figure 3.3 details the system's software architecture, for simplicity the figure only shows components of the main site, the regular sites have only a subset of these components, but they have the same behavior. The main system is composed by the Client Application, the Grid Application and the Recognition Software.

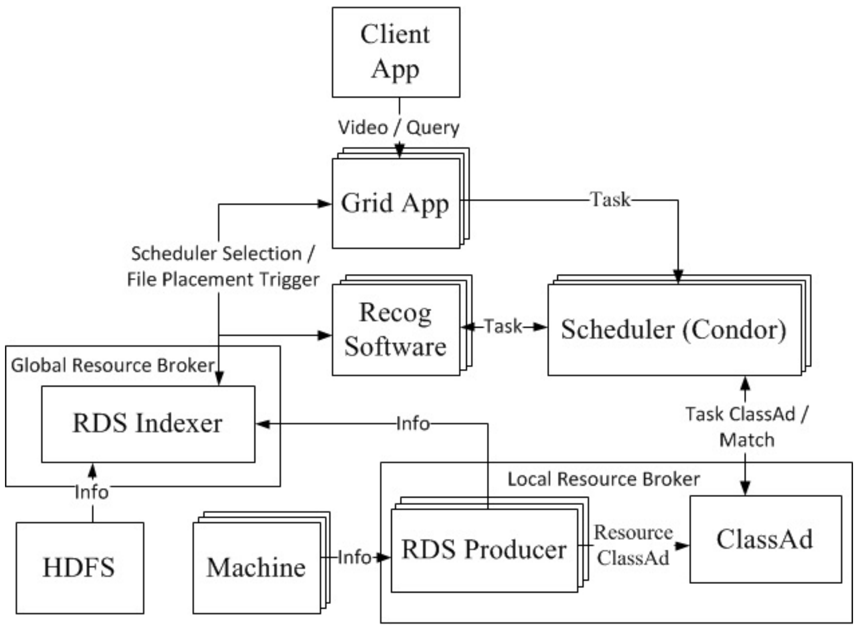


Figure 3.3: Software Architecture.

The **Client Application** interacts with the Grid Application in one of the site's machines; it submits videos to be processed or queries. Each Grid Application submits task to an instance of the Condor Scheduler present in the local machine, or in a machine in another site, if the task is to be executed in a regular site. The scheduler in turn selects a machine and sends the tasks to be executed by the Recognition Software.

The **Recognition Software**, component that actually performs the objective of the system also submits tasks to scheduler in the same way the Grid Application does, the reason for this is that the recognition process is divided in a number of steps, therefore these steps can be done in different machines.

As referenced before the repositories (video files and intermediate files) are kept in a **distributed file system**, all communications between the executables that compose the system are done throw the distributed file system except the communications between the Client Application and the Grid Application, these use standard **TCP**.

Traditional DFS keep complete files in a single machine and transfer those files on demand, as the system execution will require intensive file access, a traditional DFS would easily become the bottleneck of the system. To eliminate this difficulty **HDFS** (Borthakur 2012) was adopted, this DFS standard behavior is to divide big files in smaller chunks and store those chunks in various machines and the chunks themselves are replicated; these characteristics not only allow for good **fault tolerance** but also for good **load balance**. In the particular case of this work the segmentation of the videos in chunks was done according to the HDFS file segmentation (one video chunk always feats in in one HDFS chunk), and the chunk placement information is used for resource brokering, having the scheduler giving preference to machines that contain the files required to execute a given task.

The Condor Scheduler does not actually make the scheduling decisions, it relies on another component of the system to make scheduling decision and maintain consistency, that component is the **ClassAd Matchmaker**. ClassAd requires an external system to provide dynamic resource information, in this system the application **RDS (Resource Discovery System) Producer** present in each machine of the cluster gathers information from the local machine, translates it to a ClassAd advertisement and sends it to the matchmaker. These updates are done periodically, which impose delays in changes propagation, while these delays are acceptable in machine status, they are not in file placement, since the delays will mean that information is simply unavailable in contrast with the slightly outdated information of the machine status. The **RDS Indexer** solves the issue of file placement propagation, when either the Grid Application or the Recognition Software stores new files they trigger the RDS indexer which will fetch the information from HDFS on demand, this information is returned together with the scheduler selection, the local scheduler or another site's scheduler. The Scheduler selection is done according to the information provided by the RDS Producer.

3.3.1 Use Cases Mapping to Aplications

The main system is composed by the Grid Application and to Recognition Software which is divided into two distinct applications, the Face Detection and the Face Recognition, next we explain what stages from the use cases each application executes.

The stages in the new video submission use case are executed by the systems' components according to the following mapping:

- **Grid Application:** stage 1 - Full Video Transfer and stage 2 - Video Segmentation
- **Face Detection:** stage 3 - Frame Extraction and stage 4 - Face Detection
- **Face Recogniton:** stage 5 - Face Recogntion and stage 6 - Store Information

This aggregation of stages has the purpose of reducing overheads.

The two first stages are done by the Grid Application directly and are not sent throw the systems' scheduler. This may cause an imbalance in resource utilization if many videos are submitted at the same time, but the alternative, which would be to store the complete video file and execute the partition in another machine, would not pay off because the partition of videos is a too small task when compared with the transfer of the video file.

Next, stages 3 and 4 are executed by the Face Detection, again the frame extraction is a too small task to be executed alone, it would just impose overhead on the system from the file transfers, instead it is executed together with the face detection stage.

Finally the stages 5 and 6 are executed by the Face Recognition, once again there would be no advantages in storing the results from stage 5 and then having another machine fetching those results and updating the repositories.

The stages in the unknown faces reiteration use case are executed by the systems' components according to the following mapping:

- **Grid Application:** stage 1 - Faces Retrieval and stage 2 - Faces Grouping
- **Face Recognition:** stage 3 - Face Recognition and stage 4 - Store Information

The Grid Application is continuously running, therefore it periodically starts the execution of the unknown faces reiteration use case, after the unknown faces are retrieved and grouped, the Grid Application submits a number of new task to the scheduler, tasks which are later executed by the Face Recognition in the same manner as the stages 5 and 6 of the new video submission use case are executed.

Lastly, the query execution and face identification submission use cases are both processed by the Grid Application.

3.3.2 Execution Flow

In order to better explain how the use cases execution takes place and the components involved we present the following execution flow diagrams:

Figure 3.4 details the New Video Submission use case execution and the interaction of the system's components used in its execution.

The execution is started by the **Client Application**; it contacts the **Grid Application**, waits until the Grid Application is ready to receive the video file (signaled by the send video!) and then sends the video file completing stage 1. The Grid Application then executes stage 2, it segments the video file in a number of chunks and stores those chunks in **HDFS**. Once the stage 2 is completed the Grid Application submits a number of new tasks to the scheduler (one face detection task for each chunk), in order to do that, for each task it triggers the RDS Indexer, informing it of the existence of a new file. The **RDS Indexer** in turn answers with the scheduler selection (effectively choosing the site where the task will be executed) and file placement information which is sent to the scheduler together with the task submission for scheduling purposes (details on how this is achieved can be found in the Resource Brokers Architecture section in section 3.4).

The scheduler selects a machine in which to execute the **Face Detection** executable (details on how this selection is made can be found in the Scheduling section in section 3.5) and starts its execution. The Face Detection first retrieves the video chunk it is to process from the HDFS, note that even if the file is stored in the local machine, it still has to be transferred form the logical HDFS hard drive to a temporary folder accessible by the executable. Then the frame extraction (stage 3), and subsequent face detection and storage of detected faces (stage 4), are executed. Once stage 4 is completed the Face Detection submits a new face recognition task to the scheduler, prior to which it interacts with the RDS Indexer for the same reason the Grid Application does.

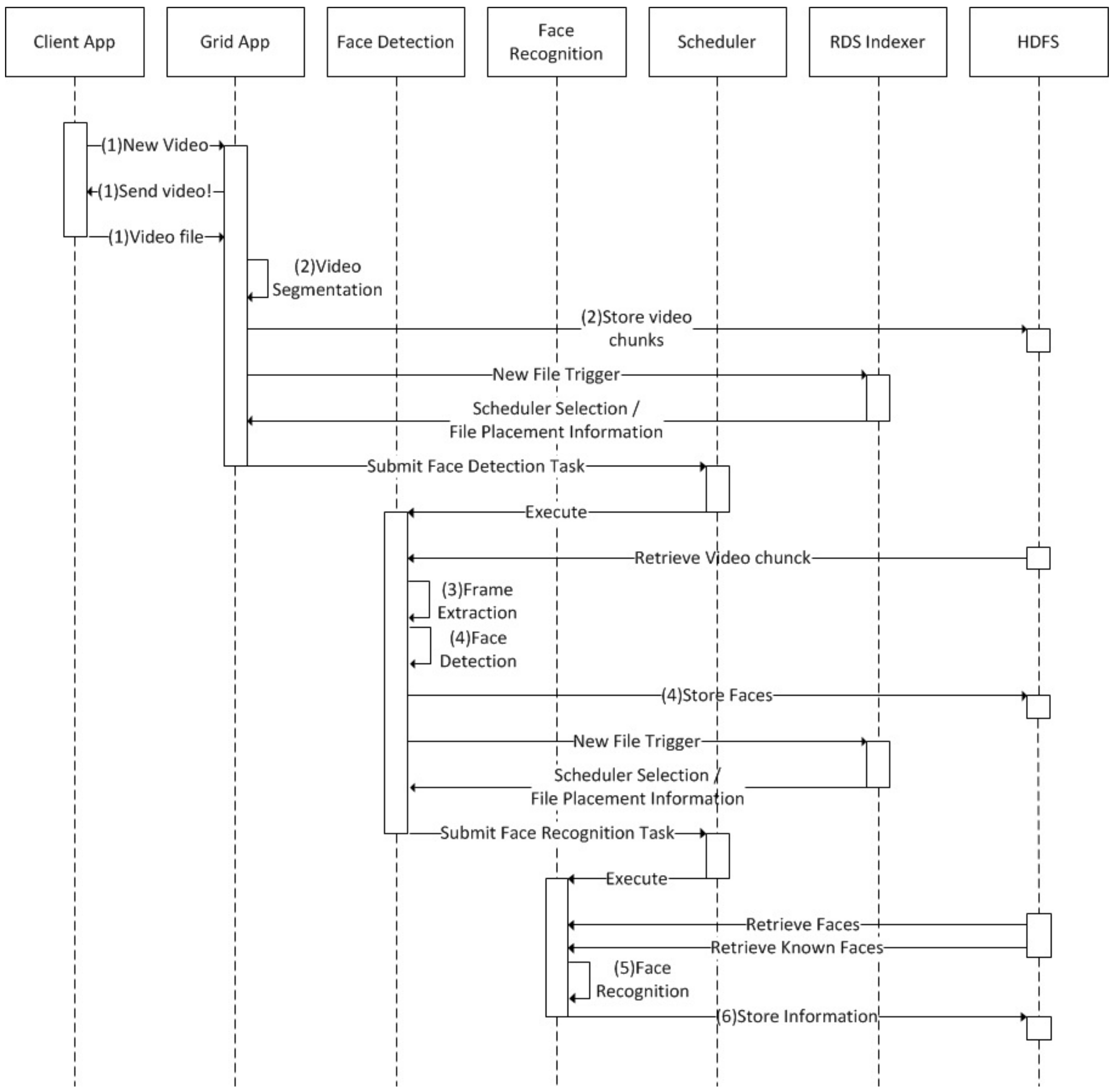


Figure 3.4: New Video Submission Execution Flow.

Finally, the scheduler selects a machine in which to execute the **Face Recognition** executable and starts its execution. The Face Recognition first retrieves the faces to process and the known faces repository which it need for the recognition process, then it executes the face recognition (stage 5) and after that updates the repositories with the new information (stage 6). That concludes the execution of the new video submission use case.

Figure 3.5 details the Unknown Faces Reiteration use case execution and the interaction of the systems components used in its execution.

The execution is started by the **Grid Application**, periodically it retrieves the unknown faces from HDFS (stage 1), arranges them into groups, similarly to the groups of faces that result from the execution of stage 4 in the new video submission use case and store these groups in the HDFS (stage 2). Once the stage 2 is completed, the Grid Application submits a number of faces recognition tasks to the scheduler;

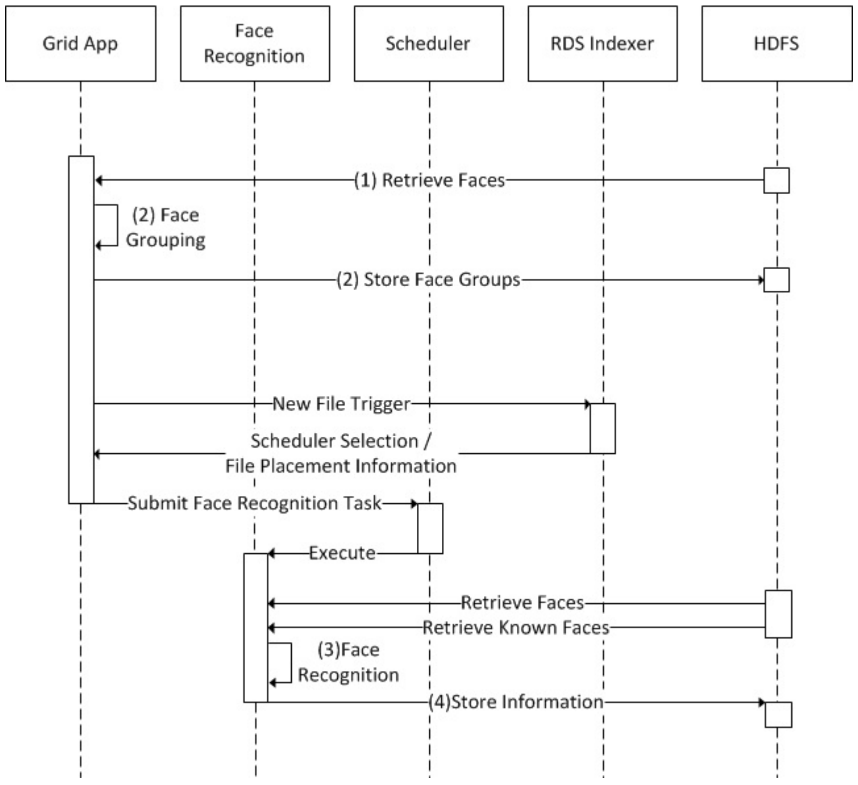


Figure 3.5: Unknown Faces Reiteration Execution Flow.

prior to the tasks submissions, the interaction with the RDS Indexer is similar to the one described from the previous use case.

The scheduler selects in which machine to execute the Face Recognition and starts its execution, with the execution of the Face Recognition being similar to the one described for the previous use case. That concludes the execution of the unknown faces reiteration use case.

3.4 Resource Brokers Architecture

In this section we describe the architecture of the resource brokers, enablers the scheduling decisions done in the system, together with a description of the resource discovery system that provides the information used by the resource brokers when making decisions. In figure 3.6 we detail the Resource Brokers and Resource Discover System architectures.

The **Local Resource Broker** that decides in which machine of a site to execute a task is composed by the **ClassAd Matchmaker** which coordinates the Condor schedulers and the **RDS Producer**. ClassAd coordinates the schedulers by matching the task's classAds, descriptions of tasks in terms of needs (requirements) and preferences (rank) , with the resource classAds , descriptions of machines' characteristics. The contents of the task classAds are defined by the FaceID Executable that submits the tasks, as for the machine classAds, the information contained in them comes from the RDS Producers which collects information from the machines.

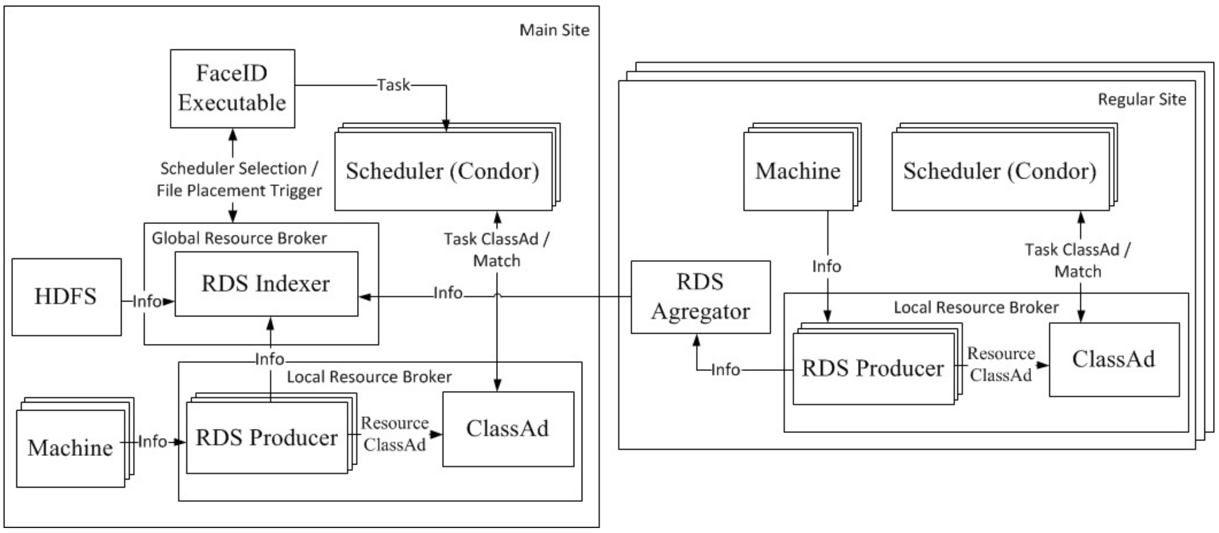


Figure 3.6: Resource Brokers Architecture.

The **Global Resource Broker** decides in which site to execute a task and is used to introduce file placement information into the schedulers on demand, avoiding delays and their associated problems as referenced before. The site to execute the task is selected using the summarized information from the main and regular sites; this information is obtained by merging the information sent by the RDS Producers in the main site or in any of the regular sites. To avoid a flooding of updates from the regular sites an additional component of the RDS is present in the regular sites, the **RDS Aggregator**, its job is to summarize the sites resource information and send that information to the RDS Indexer in the main site. As for the file placement, when a FaceID Executable is about to submit a task that requires a given file, it triggers the RDS Indexer that returns the identification of the machines where the file is stored.

All these functions, the site selection and the Resource Discovery, could have been achieved using pre-existent software, the site selection could have been done by a scheduler and the resource information could have been obtained using MDS, a hierarchical resource discovery system that naturally allows for different views of the information in different levels of the hierarchy. We choose to develop this simpler system because having the additional overhead of a fully-fledged scheduler only to make the decision of in which site to execute a task is not justifiable, especially as the system will most often execute tasks in the main site, and only send task out when resources are depleted in the main site. As for the resource discovery, MDS is not ideal since the resource dissemination is only periodic, so it would require an additional way of introducing the file placement information on demand, so to use MDS and an additional system to obtain the file placement would just further increase the complexity of the system.

3.5 Scheduling

In this section we describe the algorithm approach that schedulers use to choose in which machine each task will be executed. In general, the schedulers try to assign tasks to the machine in which the task will be executed in the least amount of time; in order to do that, we need to define a cost function that schedulers will try to minimize for each task.

3.5.1 Cost Function

The cost of executing a task for this particular system can be seen as the **predicted amount of time a task will take to be executed**, also, we must take into account the data transfer time. Therefore, the total task execution time is the sum of the time to transfer the input and output files plus the execution time. The calculation of this cost function is a very demanding task, requiring high processing power if a site is constituted by a large number of machines. It also requires very volatile resource information, the available inbound and outbound network speed as well as a prediction of the time a machine takes to process specific tasks, which would cause the resource information system to become the bottleneck of the system, as keeping this information constantly up-to-date is a demanding task. Also, it is not possible to make accurate previsions about the size of the output files (the results of processing a number of frames to extract faces can be one or one thousand faces). In addition **ClassAd is not designed to make brokering decisions based on minimization of a cost function**, instead it is designed to find matches of values between resources and tasks (requirements) helped along by a maximization of a numeric function (rank).

3.5.2 Resources Attributes

To solve the difficulties of the cost function calculations, we **describe the resources with groups of attributes**, in this case processor (**CPU**), ram memory (**MEM**), Hard Drive Capacity (**HDC**), which one of these attributes may have one of these values Low, Med, High. In ClassAd, the brokering decisions are made in two steps: in the first one **restrictions and constraints** are meet, tasks are matched with resources that can meet their requirement. If more than one match is possible a second step is executed. In this second step, one of the matched pair is chosen. In this work, we use the attributes CPU, MEM and HDC for the first step, where effectively we find machines that have the computational resources to execute the tasks. For the second step information about file placement is used to select one of the matched pairs (giving preference to machine that have the required files locally) along with the values of the attributes the machine (giving preference to the machine with the most available resources), following the rule of selecting the machine in which the task will be executed in the least amount of time.

3.5.3 Task Characterization

The system has two tasks that are scheduled, the tasks are the execution of the Face Detection and Face Recognition, incidentally these two tasks have very distinct characteristics in terms of the computational requirements. **The Face Detection** tasks are very time consuming, they have **high CPU requirements** (scan images for faces), but have **low memory and hard drive requirements** (video chuck and detected faces, relatedly low amount of memory). The Face Recognition tasks on the contrary are short tasks, they have **low CPU requirements** (calculation of distances between points in a n-dimensional space), but have **high memory and hard drive requirements** (not necessarily, but the known faces repository can have a large size). These polarized characteristics are actually beneficial for the system's load balance, as machines executing demanding tasks, face detection or others (machines are not dedicated to this system), are still able to run those least demanding task, the face recognition, instead of sending the tasks to other machine which could as a result not be able to execute a demanding task because the remaining resources are not enough.

3.5.4 Site Selection

As stated before the system will tend to execute the application always in the main site, however when the main site resources are depleted, tasks can be sent to other sites. When selecting a site to send a task, **the scheduler chooses the site with the most resources available**, again choosing the site which is able to execute the task in the least amount of time. Having two such different tasks we are able to tune the scheduling for sending tasks to be executed outside de main site in such a way we can benefit from the their characteristics.

Face Detection tasks have low time overhead from the file transfer when compared with the useful computation time, that is not the case for the **Face Recognition** tasks, the overhead from file transfers is very high when compared with the useful computation, in cases **more than fifth percent of the total execution time is file transfer overhead**. Also the total execution time for a Face Detection task is much higher than for a Face Recognition task. With this in mind we chose to only allow Face Detection tasks to be sent to regular sites as these tasks will in general have little performance lost from the additional time imposed by the file transfers, whereas the Face recognition tasks, not only have a significant overhead increase from the file transfers, but also as they are shorter tasks, the gains from sending them to be executed outside the main site will be much less than with the Face Detection tasks.

3.6 File Storage Schema

3.6.1 System Elements Representation

All system status is stored in the HDFS as files. That means that each system element is represented by a file in the file system. **Two main directories** exist in the file system hierarchy, the first one is the **temporary work files directory** which contains the video chunks and the faces extracted from video chunks and the second is the **recognition data directory** which contains all other system elements.

In the temporary work files directory, video chunks and faces must be somehow associated with each other, it must be possible for instance for a Face Recognition executable to fetch the faces extracted from a given video chunk. To achieve this mapping we use **unique identifiers for each**

element, these identifiers are derived from the original video file name concatenated with sequential numbers. For example the 6th face detected in the 2nd video chunk of a video file with the name video.wmv has the name video.wmv_2_6, this way the name of a file has all the information necessary to identify it, if the face from the example is identified by the recognition software, its name contains the name of the video it is to be associated with.

For the description of the rest of the system elements we make a brief technical explanation of the implementation of the system, further details can be found in the Implementation chapter, chapter 4. Faces are represented by a vector, named Eigen vector, that vector is obtained through the projection of the face image in the PCA matrix (Principal component analyses matrix); explained in a few words, the matrix represents the pattern characteristics of a group of train faces, and the Eigen vectors that represent faces are descriptions of those faces in terms of those pattern characteristics.

With this in mind we now describe the contents of the recognition data directory; a **subdirectory called PCA Matrix** contains a serialized representation of the PCA matrix, this matrix is required on every execution of a Face Recognition executable in order to calculate the projection of the face being identified. This matrix may need to be recalculated, if the faces become very different from the original training group, for instance if the matrix was build using European/Caucasian faces, the system may not be able to describe Asian faces well enough to differentiate them and as a result the recognition may start to have a high error rate. The need to recalculate the matrix is very unlikely tough, especially if the group of faces used in the matrix build is a diverse group, nevertheless we predicted the possibility of it having to be changed and we included a version number with the serialized matrix.

The **known Faces repository** is another subdirectory, it contained serialized representations of the Eigen vectors together with the name of the person, additionally for each vector and name the original face image is also stored, the reason for this is that if the PCA matrix needs to be recalculated, the projections become invalid and the original face image is used to recalculate the Eigen vectors.

The **Unknown Faces repository** represented by another subdirectory contains an unorganized collection of face images.

The **Video-Person repository** is the system's data-base, it is designed to answer two queries: what videos does a given person appear in and what persons appear in a given video. This **information is contained in the file system hierarchy it-self**; each video file has a directory with its name and each one of those directories has a number of subdirectories with the name of the persons in that video, and the other way round. In the subfolders, the original face images from which the information was derived are stored. Their function is to show the user an image that justifies the answer of a query. Also if the user says the information is wrong, having the original image we are able to calculate its Eigen vector and delete its association from the Known Faces repository, simply by searching for the equal Eigen vector, We do not store the Eigen vector as well, as that it would have to be recalculated if the PCA matrix changed, adding additional overhead to that task (the Video-Person repository can be very large).

3.6.2 Know Faces Repository Search

Traditional face recognition systems compare a new face (face they are trying to identify) against the entire data-base, such a solution would be a major problem on this system, since it would potentially require **very large file transfers**. In order to avoid this problem we needed a way of limiting the search on the repository.

An initial approach would be to organize the vectors according to ranges of values of their components, however this proved to be impossible as no easy relation between vector components values could be found, two faces from the same person would have some components with very different values.

We used instead mathematical properties of the vectors, their **norm and angle**, these two values do not vary wildly for images of the same person (otherwise the whole recognition system would not work), in fact we found experimentally that images of the same person on average would vary at most 1000 units in norm and 10 degrees in angle. With this information **we stored the vectors according to their norm and angle**, in ranges of 1000 units of norm an 10 degrees of angle. When trying to identify a face, we first determine its norm and angle, and transfer only nine groups of vectors, the ones within 1000 units of norm a and 10 degrees limit, figure 3.7 shows the groups (represented by a "T") that are transferred for a given projection.

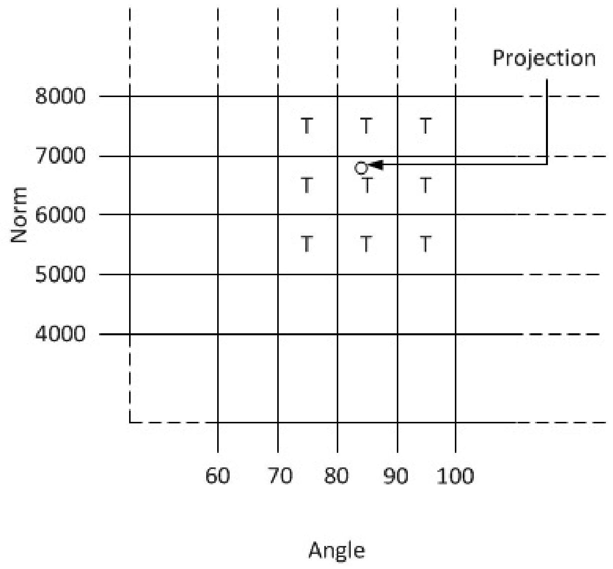


Figure 3.7: Know Faces Repository Search.

This limit also functions as a **threshold for the face recognition**, the recognition system returns the identification of the face that is more similar to the one being identified but in the case of a data-base with a low amount of information the most similar face may be one that is very different so a threshold is necessary to validate the identification.

3.7 Coordination

In this section we describe how the system applications coordinate, from all the descriptions made until this point, its safe to conclude that interdependency between elements of the system is fairly low. However coordination is still required in some points.

Up until the Face Recognition execution the files the system accesses and writes have **no need for coordination**, as there is never more than one executable trying to access them. In the first step, writing of the video chunks, retrieving of the video chunks, and then writing of the extracted faces, can be done with no coordination, as it is not possible for two executables to use the same files.

In the Face Recognition however, problems of consistency may appear as the result of race conditions. In these steps executables are reading, adding and changing potentially the same files. In the Known Faces repository the projections and original images are numbered, they are named 1.projection, 2.projection and so forth, this eases the process of adding new entries, and it also provides a simple way of programing the reading of the data base. As they are numbered, when an executable wants to add new projections it will simply name the files with sequential numbers starting from the highest number present in the repository, it is here where problems may arise. **All the executables need to be aware of the number they are supposed to start numbering their new files and is must not be possible for more than one executable to get the same number**. HDFS does not implement any kind of file system locks, therefore storing the number in a file that could be locked while writing is not feasible, instead we use a distributed share memory (DSM) system on top of which we implement the locks system, taking advantage of the atomic way variables can be changed, more details can be found in the implementation chapter, chapter 4. The number is also stored in the DSM system, as the access is several orders of magnitude faster than accessing the file system.

The locks previously mentioned are also used for access to the Unknown Faces repository so the system never has two applications gathering the same images to be reprocessed.

Finally when the **system needs to read and write the same files concurrently** for instance when the PCA Matrix has to be recalculated, the recalculation is done in the background. The new PCA Matrix and the new version of the known faces repository is recalculated while the system continues to execute on the older data. When all data is ready the system has to be stopped for the data to change, we cannot have executables working with two different version of the matrix; to achieve the coordination here we developed a semaphore inspired coordination system, also supported by the DSM system more details can be found in the implementation chapter, chapter 4.

Implementation 4

In this chapter we describe in detail the components of the system as well as some optimizations done to improve performance.

All executables developed for this system were written in C++, the reason for that was that the main dependency of the system, the OpenCV Library (OpenCV 2012) was written in C++. As it is a library that has to be linked with the code it was only evident that C++ was the appropriate language to develop the system, despite the existence of abstraction/conversion layer in other languages such as Java, the extra overhead and the increased potential for errors of such solutions do not justify the use of other languages.

The system dependencies extend further from just the OpenCV library, but all other dependencies are independent applications, the interaction with those application was done using the C function int system(const char *command), that executes shell commands, the result output of those command is then parsed by the system executables, although this may not be the ideal interface between applications, it is an effective, systematic and generic method of interacting with any application.

4.1 System Applications

In this section we describe in some detail the implementation of the systems' components together with their most relevant software dependencies.

4.1.1 Grid Application

The Grid Application is composed of two threads. **One thread is normally sleeping but periodically wakes up and starts the execution of the Unknown Faces Reiteration use case. The other tread is continuously waiting for a connection from a Client Application**. When the Client Application sends a query it answers it with information from the Video-Person repository, and if the Client Application submits a new video, the Grid Application segments the video and submits new face detection task. To segment the video the Grid Application uses MEncoder a command line application distributed with MPlayer (Mplayer 2012), we chose to use this application because it can segment a video according to file size with no quality loss, in this process we remove the sound from the video as it is not necessary.

4.1.2 Face Detection

The Face Detection application is not more than a series of steps that extract frames from video chunk and detect and extract faces from those chunks. The frame extraction is done using the OpenCV library functions CvCapture* cvCaptureFromAVI(const char* filename) and int cvGrabFrame(CvCapture* capture); that open a video file and extract a frame from a video file respectively. For the Face Detection we use **Haar-like features**, also part of the OpenCV library. Before the faces are stored they are normalized, all resultant face images have the same dimensions, they are all gray scale and their luminosity is normalized, all these transformation are achieved through tools from OpenCV. **Face images are normalized because the recognition process requires it**, if images are not of the same dimension the system does not work at all and if the images are not normalized in color and luminosity the accuracy of the recognition is greatly reduced.

4.1.3 Face Recognition

The Face Recognition Application similarly to the Face Detection Application is also a sequence of steps, but this time that identify a number of faces. The recognition process uses a implementation of Eigen Faces available in OpenCV, we based our implementation on the **libfacerec** (libfacerec 2012) which is dependent on the OpenCV ‘s implementation of Eigen Faces. The library is developed to perform recognition in a stand-alone computer, in order to use it in our system we did a few modifications; essentially **we wrote extra methods which allow for the projections used in the face recognition, as well as the PCA matrix to be introduced on demand instead of being encapsulated in the library**.

4.2 Resource Brokers

In this section we describe in some detail the components of the resource brokers and the interaction between the resource discovery system and the system schedulers.

4.2.1 RDS Producer

The RDS Producers are the base of the resource discovery system and, in conjunction with the ClassAd Matchmaker, they form the Local Resource Broker whose responsibility is to decide in which machine of their respective site a task runs.

4.2.1.1 Information Production

The RDS Producers themselves are a series of system calls to collect the machine information, specifically the **CPU load and speed, the memory and hard drive utilization**. The resource information collected is translated into the tree possible values for the resource attributes **(Low, Med and High)**; in order to do so we needed to establish what is high available memory, low CPU available a so forth. We used the machines from the main site as reference; we defined the following rules for the conversions:

CPU Available = n CPUS x CPU idle x CPU MHZ

CPU Base = 8 x 3400;

- **CPU High:** CPU Available $>$0.5 x CPU Base
- **CPU Med:** CPU Available $>$0.25 x CPU Base AND $<$0.5 x CPU Base
- **CPU Low:** CPU Available $<$0.25 x CPU Base

Here we use for reference the 8 cores (4 real) and the 3400 MHZ of the processor in the machines of the main site. We consider the CPU availability to be high when more that 50 % of the Base CPU capacity is available, that means that the real cores are still not completely used. We consider the CPU availability to be medium when it is between 25 % and 50 % of the Base CPU capacity and low when it is below 25 %. This classification is used for all machines where the system may send tasks to be executed, even if the characteristics are different, the calculation are still doable and valid, least powerful machines will be seen as more powerful machines that are always heavily loaded, for example a machine with the same CPU clock speed but with half the cores, will at most be seen as machines with medium CPU availability.

The memory and hard drive availability are converted in the same manner according to the following rules.

MEM Base = 2000;

- **MEM High:** CPU Available $>$MEM Base;
- **MEM Med:** CPU Available $>$0.1 x MEM Base AND $<$MEM Base;
- **MEM Low:** CPU Available $<$0.1 x MEM Base;

HDC Base = 5000;

- **HDC High:** CPU Available $>$HDC Base;
- **HDC Med:** CPU Available $>$0.1 x HDC Base AND $<$HDC Base;
- **HDC Low:** CPU Available $<$0.1 x HDC Base;

The values for the base memory and base hard drive capacity were chosen to allow the load of a large part of the Known Faces Repository with a substantial size, which is very unlikely due to the restrictions in the Known Faces Repository search, but still possible.

4.2.1.2 Information Dissemination

The information produced is sent to two different elements of the system, the **RDS Indexer and the ClassAd Matchmaker**; the updates are done periodically, **every 5 seconds**. We chose 5 seconds for the interval between updates as we determined experimentally that the ClassAd is not able to process updates much faster.

The ClassAd is configured to fetch the resource information from a file with a given frequency, so all we can do is create the file and wait for the ClassAd to fetch the information, we determine experimentally that for periods of less than 2 seconds the update frequency would became unstable possibly due to overloads.

We chose 5 seconds to include a safe margin that allows the system to work with a stable update frequency, moreover 5 seconds is a small fraction of the time the smallest task takes to be executed, therefore the information is kept enough up-to-date.

4.2.2 RDS Indexer

The RDS Indexer collects information from all the machines in the main site and resumes it in order to have an overview on the site's load. It also receives information from other sites. As it receives the resource information, it makes the decision of to which site to send tasks, the decision is made in the background and is ready when an executable request it.

4.2.3 RDS Aggregator

The RDS Aggregator that executes in the regular sites is similar to the RDS Indexer application, only it sends the summarized site information to the RDS Indexer in the main site.

4.2.4 Scheduling

The system makes two scheduling decision, in which machine of a site a task is executed and in which site a task is executed.

Helping the scheduling choosing in which machine of a site a task is executed, are the machine attributes listed in 4.2.1.1 and the file placement information. The memory and hard drive capacity are fixed requirements, meaning that only machines who meet those requirements are considered to run a given task, the CPU available and the file placement information are used to balance the resource usage and to prevent starvation.

The system is not required to respond in real time, instead it must be able to process large amount of videos using the available resource as efficiently as possible.

```
Universe     = vanilla
Executable   = Face-Detection
Arguments    = <args>
Log          = <logfile>
Output       = <outputfile>
Error        = <errorfile>
Requirements = FACEID_CPU == 2 || FACEID_CPU == 3
Rank         = 2 * (FACEID_CPU == 3) + Machine == <MachineName>
queue
```

Figure 4.1: Face Detection ClassAd advertisement

```
Universe     = vanilla
Executable   = Face-Recognition
Arguments    = <args>
Log          = <logfile>
Output       = <outputfile>
Error        = <errorfile>
Requirements = FACEID_MEM == 3 && FACEID_HDC == 3
Rank         = 2 * Machine == <MachineName> + (FACEID_CPU == 1 || FACEID_CPU = 2)
queue
```

Figure 4.2: Face Recognition ClassAd advertisement

For each task that can be scheduled we wrote a deferent ClassAd advertisement (figures 4.1 and 4.2). Most relevant for the current description are the Executable element which identifies the executable to run and the Requirements and Rank elements which are the ClassAd parameterization that translate the Scheduling approach.

Requirements is a boolean expression and Rank is a numeric expressions, equality expressions evaluate to 1 or 0 when the equality is true or false respectively. ClassAd will choose the machine in which the Requirements evaluates to true and the Rank evaluates to the highest value.

In the machine information introduced into ClassAd, the High, Med and Low values for the attributes are represented by numbers, 3 represents High, 2 represents Med and 1 represents Low. The preference for a machine that have the required files locally is expressed by the Machine == <MachineName>.

As described before the two tasks are very different in terms of machine requirements and execution time.

The Face Detection tasks require mostly CPU therefore the Requirements are "FACEID_CPU == 2 || FACEID_CPU == 3" that means that only machines with High and Med CPU available are considered to execute these tasks. Within those machines ClassAd gives preference to machines with High CPU and machines that have the required files locally, therefore the Rask is "2 * (FACEIDCPU == 3) + Machine == <MachineName>". The CPU attribute is more important for the scheduling, hence the multiplication by two. The reason for this is that the file transfer overhead is of reduced significance in these tasks.

The Face Recognition tasks on the other hand are relatively short tasks that require most of all Memory and Hard Drive Capacity. The overhead from transferring the required files is not neglectable. The Requirements for these task are therefore "FACEID_MEM == 3 && FACEID_HDC == 3" which means that only machine with High Memory and Hard Drive Capacity are considered. When choosing between the considered machines ClassAd gives preference firstly to those that have the required files locally and secondly to machine the CPU med or low CPU. The Rank for these task is "2 * Machine == <MachineName> + (FACEID_CPU == 1 || FACEID_CPU = 2)".

The preference for machines with High CPU available when scheduling Face Detection tasks and Low CPU available when scheduling Face Recognition task produce a balance resource usage as demanding tasks are preferably sent to machines with more available resources and low demanding task to machines with lower resources. Machines with medium resources can have both tasks assigned to them in order to use all the sites resources. Preventing machines from being saturated with Face Detection tasks (not allowing these to be assigned to machines with low CPU available) prevents starvation of Face Recognition tasks. If this restriction was not imposed Face Recognition task could easily only be executed after all Face Detection tasks were executed.

For the choice of in which site to execute a task a site rank is used. This rank is a number calculated from the attributes of the site's machines. As explained before, only the Face Detection tasks are sent to regular sites, therefore the site rank reflects the capacity of a site to execute those tasks. The rank is the sum of CPU attributes if the available CPU is med or high which are the requirements for the face detections tasks. The actual scheduling follows a simple rule, when there are no machines in the main site with med or high CPU available the regular site with the highest rank is selected.

4.3 *Optimizations:*

In this section we describe some optimizations we did in the system's design, the changes described are not a strict requirement for the system to work, they are only relevant to improve performance.

4.3.1 Know Faces Repository Transfer

Originally the Eigen vectors and names that represent faces already identified by the system were to be stored and transferred individually, however early tests showed this procedure to be extremely inefficient since the transfer of each file would require an individual request to the HDFS, not that our application had to execute a separate command for each file, the file system did that internally. The solution for this problem was to **compress all files from a given directory in a unique file**, this reduce the transfer time massively. However having a unique file containing all vectors raises the problem of consistency, if an executable tries to change the compressed file, another may download a corrupted file, since HDFS has no locking mechanism. To allow concurrent reads and safe writes we use the previously mentioned semaphore like coordiantion system, a system described the Lock System section, section 4.4.

4.3.2 New Eigen Vectors

Being a video based recognition system, a large amount of face images will exist, and consequently a large amount of identified faces will exits too. We could simply store all the Eigen vectors from all the face images identified, but by doing so, the Known Faces Repository would quickly become too large. Also as the images in a video are all very similar, storing a large number of images from the same video would end up being redundant. Furthermore, studies such as the work in (Li & Tang 2002) have shown that **the number faces from different persons contributes far more for the accuracy of the recognition system than the number of faces from the same person**. We chose to only save up to 30 eigen-vectors per person per range (norm and angle as explained in section 3.6.2).

4.3.3 Unknown Faces Reiteration

In the face recognition process, when a face is not identified it is stored in the Unknown Faces repository to be reprocessed later, possibly when new information exists and the system is able to identify it.

However the face detection process inevitably produces images that are wrongly identified as faces, and most of those faces are never identified by the recognition process (same may also be wrongly identified). **The accumulation of non-face images would eventually become a problem for the system, as the repository would grow in size and more important, the unknown face reiterations would start to consume too much system resources**.

To address this issue **we implemented a system of priorities**, three in our implementation, essentially the more an image is processed the least priority it has and less often it is processed. A face image from a video just submitted that is not identified is stored in the highest priority group; once it is reprocessed and not identified it can be transferred to a lower priority group and so forward. In our implementation the faces with the highest priority are processed at least once a day, the faces with medium priority are processed at least once a week and the ones with the lowest priority are processed once a month. The faces in each group of priority may be processed more often providing the system's load is low; the faces are only transferred to a lower priority group when the time limit for a reiteration is reached.

4.3.4 File Caches

The system uses caches of files in two different situations to reduce the need for file transfers.

The first one is the **PCA matrix** which is rarely changed and of considerable dimension. Every computer in the grid has a copy of the PCA matrix. Before loading it the executable verifies whether it has the most recent version of the matrix (version number is placed in the distributed shared memory system). Only one matrix exists per computer, the fact that it can be changed requires the use of locks; we used the semaphore like locks to allow for concurrent reads.

The second situation is the transfer of **portions of the Known Faces repository**, each portion is only transferred once per execution; the portions are not permanently stored by any period of time because they change very often and therefore it would be almost certain that the machines cache would be outdated most of the time.

4.4 Locks System

In the system we implemented two types of distributed locks, the first ones behave like mutexes while the second ones behave like semaphores. Our implementation is supported by the distributed share memory system Memcached (Fitzpatrick 2004).

The **simple locks**, similar in function to mutexes are implemented directly on top of the distributed share memory system, in memcached variables can be created and deleted atomically; the implementation of the locks is therefore done according to the algorithms in figure 4.3.

Algorithm 1 AquireLock("Lock Name")

```
$SUCCESS$ ←false
$SUCCESS$ ←createVariable("Lock Name")
while $SUCCESS$ =false do
    sleep(1s)
    $SUCCESS$ ←createVariable("Lock Name")
end while
```

Algorithm 2 ReleaseLock("Lock Name")

```
deleteVariable("Lock Name")
```

Figure 4.3: Simple Locks Algoritm.

The creation of the variables is successful or not successful depending on whether the variable already exists or not. If the variable exists the executable sleeps for one second before it tries to create the variable again to avoid overloading the distributed shared memory system; as the creations of variables is atomic, only one executable will successfully create a variable at each time, achieving this way the functionality of a lock. The release of the lock is done simply be deleting the variable.

The **semaphore like locks**, which are used to allow for concurrent reads and only block access during writes, rely on the previous type of locks. The idea is that executables are able to read files as long as no executable has mark them for writing, once an executable marks a file for writing it must wait for the reads in progress to finish and then write the file. This is implemented though the association of a number to a lock. The implementation of the locks is done according to the algorithms in figure 4.4.

As the algorithms shows the semaphore like locks uses the simpler locks described before in their implementation. The write lock holds the readers once a writer appears, if not present it would be possible for the counter to never reach 0 as readers could be always appearing. The read lock is necessary to avoid inconsistencies when two readers try to increment or decrement, both could read the same initial value and one of the updates would be lost. Similarly to the simpler locks, when acquiring a write lock, the executable checks the values of the counter variable once every second, again with the objective of not overloading the distributed shared memory system.

Algorithm 3 AquireReadLock("Lock Name")

```
AquireLock("Lock Name" + "Write")
AquireLock("Lock Name" + "Read")

    $COUNTER \leftarrow$ getVariable("LockName" + "Counter")
    $COUNTER =$ COUNTER + 1
    setVariable("LockName" + "Counter", "Counter")

ReleaseLock("Lock Name" + "Read")
ReleaseLock("Lock Name" + "Write")
```

Algorithm 4 ReleaseReadLock("Lock Name")

```
AquireLock("Lock Name" + "Read")

    $COUNTER \leftarrow$ getVariable("LockName" + "Counter")
    $COUNTER =$ COUNTER - 1
    setVariable("LockName" + "Counter", "Counter")

ReleaseLock("Lock Name" + "Read")
```

Algorithm 5 AquireWriteLock("Lock Name")

```
AquireLock("Lock Name" + "Write")

$COUNTER \leftarrow$ getVariable("LockName" + "Counter")

while $COUNTER >$0 do
    sleep(1s)
    $COUNTER \leftarrow$ getVariable("LockName" + "Counter")
end while
```

Algorithm 6 ReleaseWriteLock("Lock Name")

```
ReleaseLock("Lock Name" + "Write")
```

Figure 4.4: Semaphore Like Locks Algoritm.

5 Evaluation

In this chapter we present and evaluate the results from tests done to the system. The evaluation is divided in two main sections, the first deals with the speedup achievable in the system, which is the main objective of this work, followed by the analysis of its scalability. At the end of this chapter we made an overall evaluation of the system.

5.1 Speed Up

For the purpose of testing and evaluating the system we run the systems in a grid composed by five computers, these were all identical and were equipped with an Intel I7 processor (4 cores, 3.4Ghz) and 12Gb of Ram Memory. They were connected via a 1Gb local network.

A number of videos were used in the tests, they were of two categories, HD (1280x720 resolution) and SD (640x360 resolution). Using videos of different resolution was important to understand if and how the system is affected by this.

For the test the video chunk size was 15 MB and each video was divided into 15 chucks. The graph in figure 5.1 illustrates the average full execution time for both these video categories.

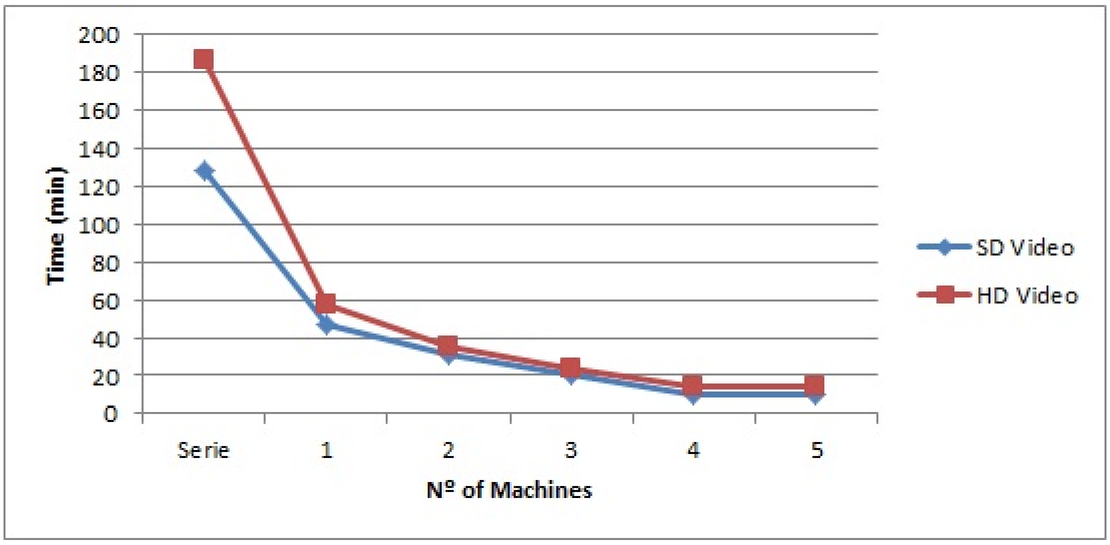


Figure 5.1: Complete Execution Time Vs Number of Machines.

A quick analysis of the graph show that the variation of the execution time is not linear, at this point it is important to remember that the system is optimized to process a large quantity of videos, more than the hardware can process at one time, while guarantying that there is no task starvation. In a test scenario where the videos to process are enough to fill all the machines the variation of execution time would be almost linear. However, we chose to do the evaluation with a scenario for which the system is not optimized because such scenario allows for a better demonstration of the scheduling characteristics of the system. For instance adding a fifth machine produced no decrease in execution time, this would not had been visible in an optimized scenario.

Each machine had eight slots available, that means that each machine was able to process eight tasks at any given time, the processors however, only had 4 real cores, so if more than half the slots were being used the execution time of the tasks sharing the real core would almost double. We refer to these two processing capabilities as Under Real Processor (URP) and Over Real Processor (ORP).

The graph in figure 5.2 illustrates the speed up obtained by adding machines to the system:

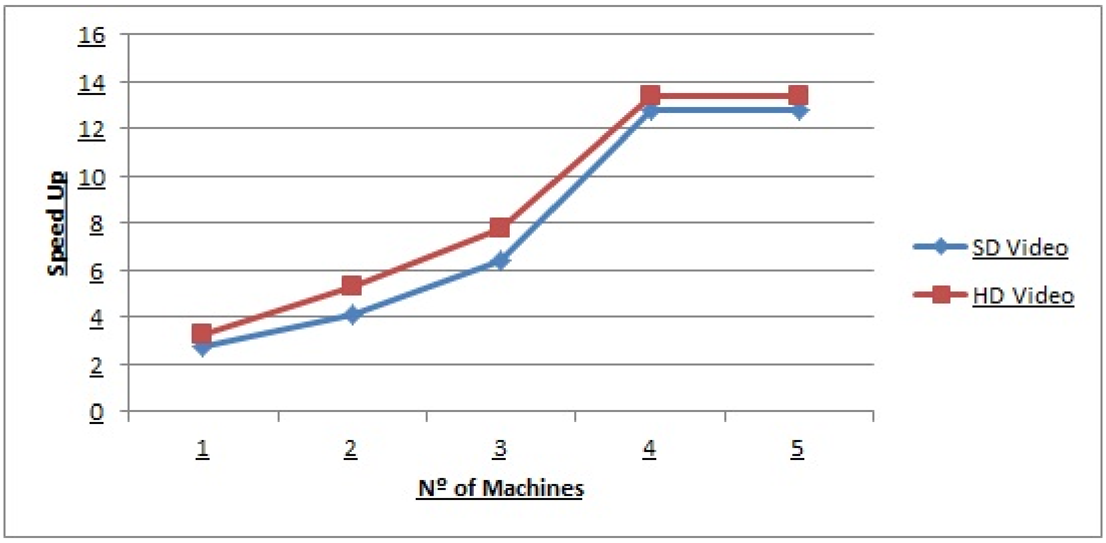


Figure 5.2: Speed Up Vs Number of Machines.

Both graph resume the information gathered during this part of the evaluation and they are now explained.

The first dot in the graph in figure 5.1 is the series execution that is a simulation of a stand-alone face recognition application; the simulation was achieved by executing the same Face Detection and Face Recognition executables used in the distributed system but instead of video chucks the executables processed complete video files, also the intermediate files were stored in the local hard drive.

Using only one machine (8 slots) we can see immediately a significant reduction in execution time, the videos files which were divided into 15 chucks, had 7 of those chunks being processed at the same time (8th slot is reserved for the Face Detection 4.2.4).

By adding machines 2 and 3 there is additional speed up despite the system being in ORP in that situation, in the case of two machines, both had to process 7 chucks followed by the 15th chunk, as for the case of three machines, all three had to process 5 chunks which is still above the 4 real processors.

Adding a 4th machine, the systems starts to work in URP, three of the node process 4 chunks and one process tree, now with each task having a dedicated processor the increase in speed up in very significant, as it is visible in figure /reffig:speedup

Once the system is working in URP, adding more machines produces no improvement, which was the case for the 5th machine, when it was added the execution time suffered no change and consequently the speed up did not change as well.

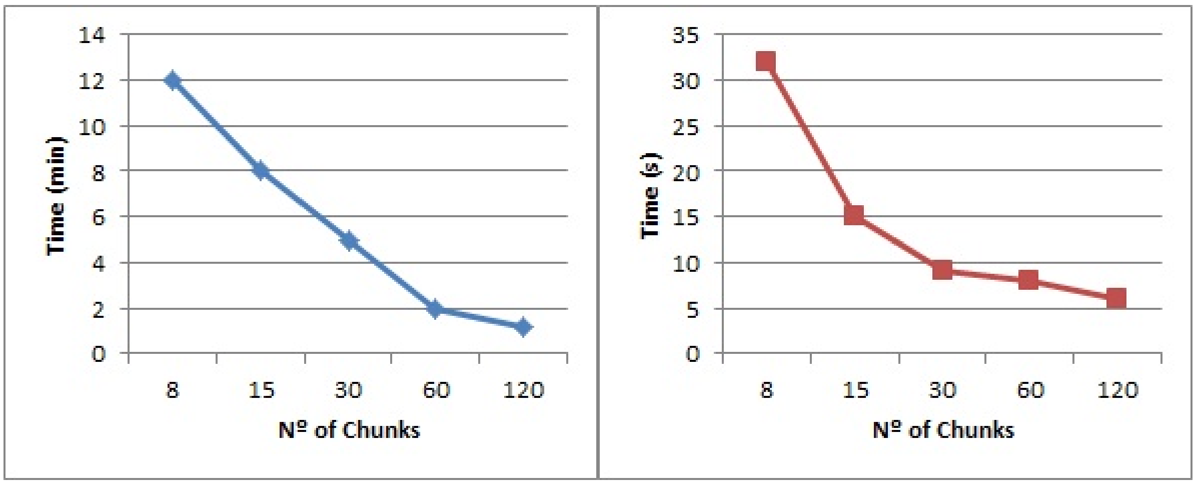
5.2 Scalability

With the test setup we used it was not possible to determine experimentally the scalability limits of the systems, the number of machines was not enough to reach any relevant bottleneck or congestion point that would prevent the system from processing more videos even adding more machines.

Still it was possible to get a good idea of the overheads present in the systems, and with that we were able to theorize about its scalability.

For this purpose, we increased the number of chunks a video is divided into, by means of reducing their size. The objective was to determine experimentally when the chunks were so small that the overhead would increase the overall execution time. We use that information to understand how the overhead imposed by the distribution affects the overall performance of the system.

The graphs in the following figures summarize the results from these tests.



(a) Face Detection Tasks.

(b) Face Recognition Tasks.

Figure 5.3: Individual Tasks Execution Time.

As expected the execution time of individual tasks is reduced when the chunk size is reduced as it is clear from the analysis of the graphs in figure 5.3. In this particular test, the fact that the machines are working in ORP for the cases of 30 chucks or more does not seem to have the profound effect on the execution times as we have seen for larger chunks. This is probably due to the much reduced chunk size which is 4 MB in the 30 chunk case and 1 MB in the 120 chunk case.

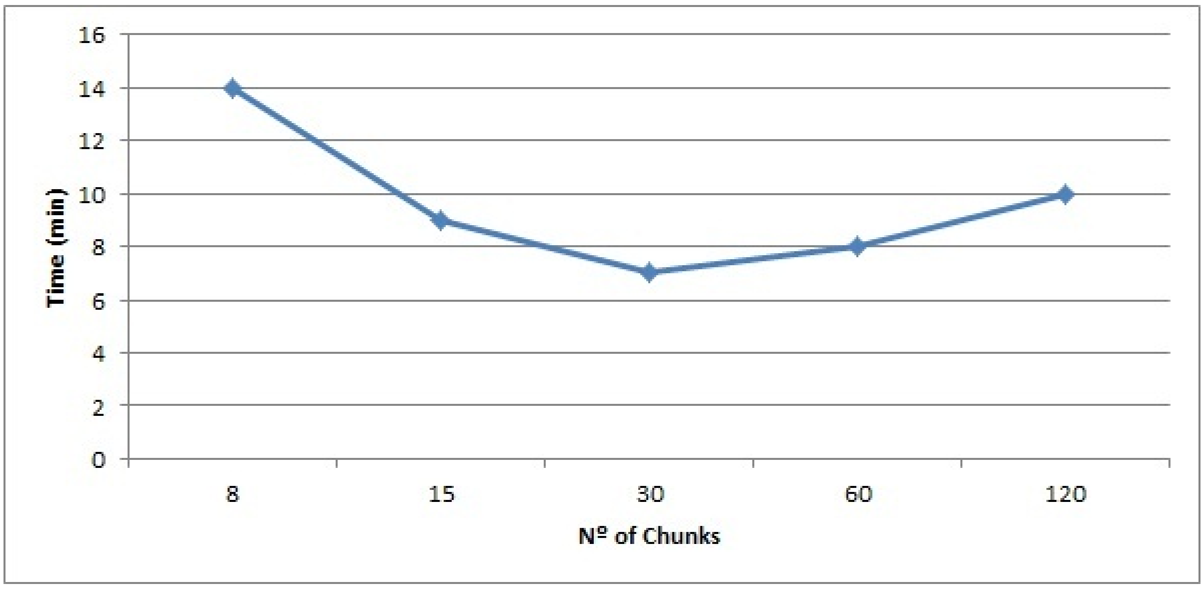


Figure 5.4: Complete Video Execution Time.

Although the individual task execution time always diminishes with the chunk size reduction as expected, the complete video execution time does not; thus, this allows us to identify the break-even point that reveals when the overhead of having more short running tasks becomes greater than the gains obtained by the extra parallelism. The overheads on this system are the files transfers and the delays imposed by the all tasks not being submitted at the same time and Condor not starting them immediately. While the file transfer overheads are reduced with the reduction in size of the chunks the delays in submission and task start only increase with the number of tasks being scheduled.

We can conclude from this information that the system does not scale indefinitely just by parallelizing further the execution of each video. The graph in figure 5.4 demonstrates this well, doubling the chunk number even when it reduced the execution time, produces a reduction of less than 50% while the execution of that video is using double the resources. Therefore it is safe to conclude that the system can in-fact scalable to very large dimensions proving that the chunks are large in order to keep the overhead time low when compared with the individual task execution time.

The graphs in figures 5.5 and 5.6 show the average CPU and memory usage while running the tests. In order for the system to scale, it is important that CPU and memory loads are balanced. The Scheduling approach used in this work aims to achieve a globally overall balanced load distribution. However, for smaller workloads such as the case of 8 chunks, this is not so, as machines 1 and 2 are idling, this is its normal behavior the scheduler will send a task to a machine with HIGH CPU available, regardless of its actual load and load of other, machines one and two happened not to be chosen. Overall, the most relevant aspect if that as the number of chunks increases in the workload, the system load converges towards balance which is a good indicator of the overall scalability of our solution.

The memory does not change much with the variation of chunk size, the Face Detection application only keeps one image in memory at each time, and therefore the memory usage is low and independent of chunk size. The Face Recognition application on the contrary can consume large amount of memory, in fact we did an optimization to deal with this specific problem. The Known faces database is divided in groups and only a small number of those groups are transferred to be processed as described in 3.6.2, therefore only that fraction of the database is loaded into memory and processed, this optimization allows for these tasks to be very short when compared with the Face Detection even for large databases.

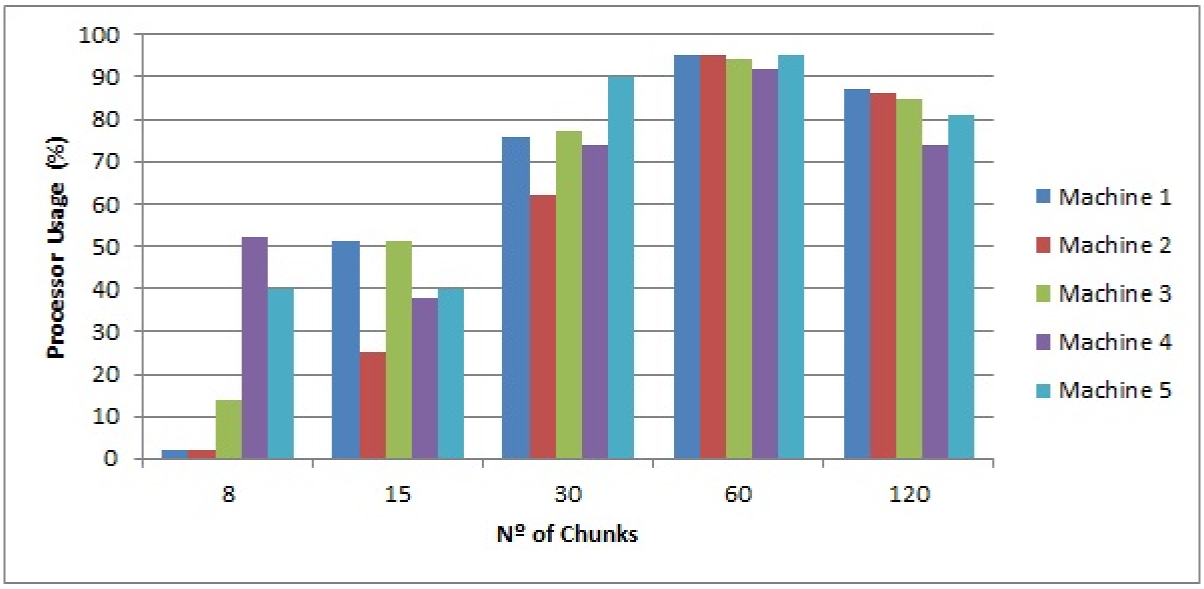


Figure 5.5: Processor Usage.

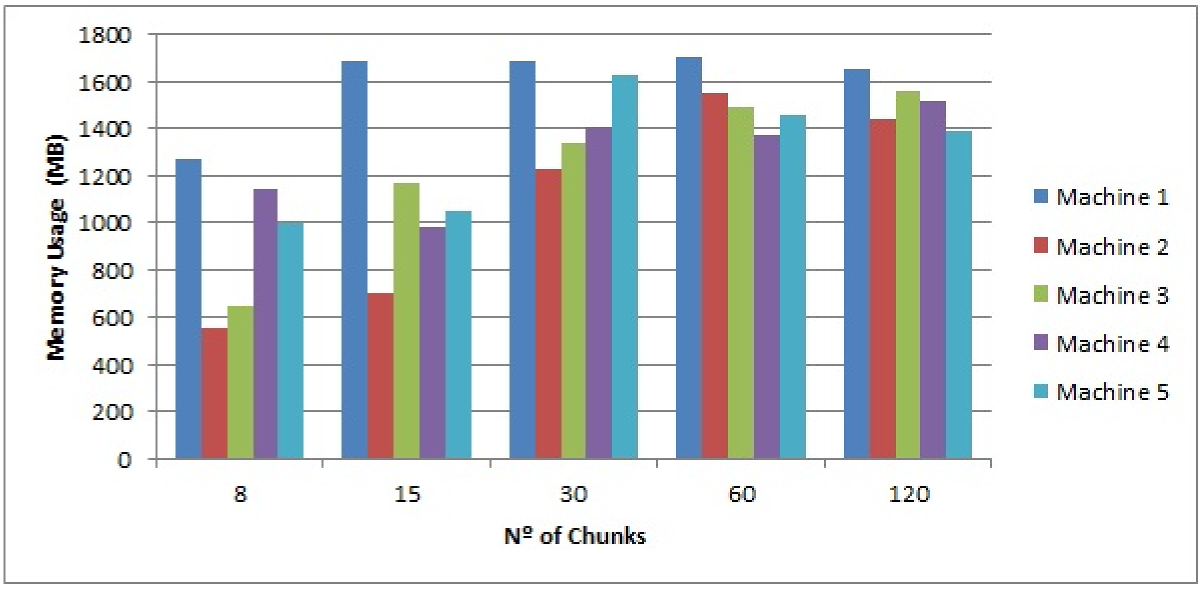


Figure 5.6: Memory Usage.

5.3 Overall Evaluation

We tested the system in order to determine if its main objectives were fulfilled, those where increasing the face recognition performance and build a scalable system. Both those objectives were achieved as it can be seen in the evaluation results.

We were able to get a Speed Up close to 13, in the demonstrated example. We analyze the scalability of the system, we were able to identify how the distribution overhead affects the systems and with that information, conclude that with large video chunks and the optimization for the known faces data transfer the system is able to scale to a large size.

6 Conclusion

In this chapter we wrap-up this work together with a summary of all the it's content.

6.1 Conclusion

In this work we designed and develop a Face Recognition system for videos that leverages the computational power of a grid environment. Face Recognition, especially in videos, is a high demanding application that can benefit from grid computing. While both Grid Computing and Face Recognition fields have been the focus for many research, the usage of grid computing to enhance the capabilities of Face Recognition had not yet been develop.

The main objective for this work was to develop a distributed face recognition system with better performance than a standalone application and that could scale to a large size.

We design the system around the two main use cases, the New Video Submission and the Unknown Faces Reiteration; we divided the video processing into two applications, the Face Detection and Face Recognition. For the scheduling we use Condor together with ClassAd Matchmaker. To feed information into ClassAd we use a purpose build resource discovery system. To improve scalability, the system is able to send tasks not only to one grid but to several grids, although it only does it when the main grid is filled. We did a few optimizations to improve performance; the main one was a new method to search for the closest face in the Face Recognition process which avoids the transfer the complete database to each machine.

Our implementation is based in OpenCV which is used for the image processing, we use HDFS for the file storage, we developed a purpose build parameterization for Condor and we maintain the distribution coordination by means of distributed locks that are implemented making use of the atomic way variables can be created and changed in MemCached, an distributed share memory system.

We tested the system and we were able to get a Speed Up close to 13, in the demonstrated example. Next we analyzed the scalability of the system, we were able to identify how the distribution overhead affects the systems and with that information, conclude that with large video chunks and the optimization for the known faces data transfer the system is able to scale to a large size.

We conclude now, iterating though the objectives of this work that we consider globally achieved:

- Develop a distributed system with better performance than a standalone application that could scale to a large size which was achieved.

- Develop a distributed architecture for the system, incorporating existing stand-alone face recognition software, which we did, the systems runs in a Condor Grid and is based in OpenCV.
- Develop a data storage schema for the distributed system, for a unified data access. We used HDFS and developed a folder hierarchy where information is stored and that represents the internal state of the system.
- Develop a distributed coordination system that manages the interdependencies between distributed elements. We develop a distributed locks system supported by MemCached.
- Develop a scheduling algorithms approach that balances the load across the machines in a cluster. We develop a parameterization for Condor that produces schedules that provide the better usage of the available machine resources. These schedules are designed in order to allow the processing of a large number of videos using the available resources in a balanced faction rather than executing single videos with the most speed up possible.

6.2 Future Work

During the development of this work there were a few aspects of it that we decided to simplify and leave their development for future work.

- Implement an execution assurance system so that jobs that for any reason fail, could be submit again. At this moment the system relies entirely in the Condor mechanisms to ensure a task is executed, which work well enough if machines do not fail permanently. If a machine fails permanently, the tasks queued in that machine will be lost.
- Carry out further studies on the ideal values for chunk size, taking into account cluster sizes, number of cores in each machine, available memory and video formats.
- Build Adapters to allow the connection of the system to grids that are not controlled by Condor. An adapter would have to be made to translate the resource information from another system into ClassAd and another adapter would submit the condor tasks to another scheduler.

Bibliography

Agarwal, M., N. Jain, M. Kumar, & H. Agrawal (2010). Face recognition using eigen faces and artificial neural network. *International Journal of Computer Theory and Engineering 2*(4), 1793–8201.

Bester, J., I. Foster, C. Kesselman, J. Tedesco, & S. Tuecke (1999). Gass: A data movement and access service for wide area computing systems. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pp. 78–88. ACM.

Boloni, L. & D. Marinescu (2000). An object-oriented framework for building collaborative network agents. *INTERNATIONAL SERIES IN INTELLIGENT TECHNOLOGIES*, 31–64.

Borthakur, D. (2012, January 03,). Hdfs architecture. URL: http://hadoop.apache.org/common/docs/current/hdfs_design.html.

Burl, M., T. Leung, & P. Perona (1995). Face localization via shape statistics. In *Int Workshop on Automatic Face and Gesture Recognition*. Citeseer.

Buyya, R., D. Abramson, & J. Giddy (2000). Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid. In *Proceedings. The Fourth International Conference/Exhibition onHigh Performance Computing in the Asia-Pacific Region, 2000.*, Volume 1, pp. 283–289. IEEE.

Chapin, S., D. Katramatos, J. Karpovich, & A. Grimshaw (1999). The legion resource management system. In *Job Scheduling Strategies for Parallel Processing*, pp. 162–178. Springer.

Coleman, N., R. Raman, M. Livny, & M. Solomon (2003). Distributed policy management and comprehension with classified advertisements. *University of Wisconsin*.

Cooke, A., A. Gray, L. Ma, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Byrom, L. Field, S. Hicks, et al. (2003). R-gma: An information integration system for grid monitoring. *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, 462–481.

Cox, I., J. Ghosn, & P. Yianilos (1996). Feature-based face recognition using mixture-distance. In *Proceedings CVPR'96, 1996 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 1996.*, pp. 209–216. IEEE.

Czajkowski, K., I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, & S. Tuecke (1998). A resource management architecture for metacomputing systems. In *Job Scheduling Strategies for Parallel Processing*, pp. 62–82. Springer.

Deelman, E., M. Livny, G. Mehta, A. Pavlo, G. Singh, M. Su, K. Vahi, & R. Wenger (2008). Pegasus and dagman from concept to execution: Mapping scientific workflows onto today's cyber-infrastructure. *IOS, Amsterdam*, 56–74.

Dong, F. & S. Akl (2006, January). Scheduling algorithms for grid computing: State of the art and open problems. *Technical Report 2006-504, School of Computing, Queen's University, Kingston, Ontario*.

Feitelson, D. & A. Weil (1998). Utilization and predictability in scheduling the ibm sp2 with backfilling. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998.*, pp. 542–546. IEEE.

Finin, T., R. Fritzson, D. McKay, & R. McEntire (1994). Kqml as an agent communication language. In *Proceedings of the third international conference on Information and knowledge management.*, pp. 456–463. ACM.

Fitzgerald, S., I. Foster, C. Kesselman, G. Von Laszewski, W. Smith, & S. Tuecke (1997). A directory service for configuring high-performance distributed computations. In *Proceedings. The Sixth IEEE International Symposium on High Performance Distributed Computing, 1997.*, pp. 365–375. IEEE.

Fitzpatrick, B. (2004). Distributed caching with memcached. *Linux journal* (124), 72–74.

Foster, I. (2006). Globus toolkit version 4: Software for service-oriented systems. *Journal of computer science and technology 21*(4), 513–520.

Foster, I. & C. Kesselman (1999). *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann, 1st Editon, ISBN-13: 978-1558604759.

Gehring, J. & A. Streit (2000). Robust resource management for metacomputers. In *Proceedings. The Ninth International Symposium on High-Performance Distributed Computing, 2000.*, pp. 105–111. IEEE.

Graf, H., E. Cosatto, D. Gibbon, M. Kocheisen, & E. Petajan (1996). Multi-modal system for locating heads and faces. In *Proceedings of the Second International Conference on Automatic Face and Gesture Recognition, 1996.*, pp. 88–93. IEEE.

Hamscher, V., U. Schwiegelshohn, A. Streit, & R. Yahyapour (2000). Evaluation of job-scheduling strategies for grid computing. In R. Buyya & M. Baker (Eds.), *GRID*, Volume 1971 of *Lecture Notes in Computer Science*, pp. 191–202. Springer.

Hjelmås, E. & B. Low (2001). Face detection: A survey. *Computer vision and image understanding 83*(3), 236–274.

Hjelmas, E. & J. Wroldsen (1999). Recognizing faces from the eyes only. In *Proceedings of the 11th Scandinavian Conference on Image Analysis*. Citeseer.

Kass, M., A. Witkin, & D. Terzopoulos (1988). Snakes: Active contour models. *International journal of computer vision 1*(4), 321–331.

Krallmann, J., U. Schwiegelshohn, & R. Yahyapour (1999). On the design and evaluation of job scheduling algorithms. In *Job Scheduling Strategies for Parallel Processing*, pp. 17–42. Springer.

Krauter, K., R. Buyya, & M. Maheswaran (2002). A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience 32*(2), 135–164.

Li, Z. & X. Tang (2002). Eigenface recognition using different training data sizes. In *Information Conference on Information Security, Shanghai, China*.

libfacerec (2012, October 03,). libfacerec. URL: http://www.bytefish.de/blog/pca textunderscorein textunderscoreopencv.

Lin, S., S. Kung, & L. Lin (1997). Face recognition/detection by probabilistic decision-based neural network. *Neural Networks, IEEE Transactions on 8*(1), 114–132.

Litzkow, M., M. Livny, & M. Mutka (1988). Condor-a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems, 1988.*, pp. 104–111. IEEE.

Low, B. & M. Ibrahim (1997). A fast and accurate algorithm for facial feature segmentation. In *Proceedings., International Conference on Image Processing, 1997.*, Volume 2, pp. 518–521. IEEE.

Meng, L., T. Nguyen, & D. Castanon (2000). An image-based bayesian framework for face detection. In *Proceedings. IEEE Conference on Computer Vision and Pattern Recognition, 2000.*, Volume 1, pp. 302–307. IEEE.

Mian, A., M. Bennamoun, & R. Owens (2008). Keypoint detection and local feature matching for textured 3d face recognition. *International Journal of Computer Vision 79*(1), 1–12.

Mohan, A. & N. Sudha (2009). Fast face detection using boosted eigenfaces. In *Industrial Electronics & Applications, 2009. ISIEA 2009. IEEE Symposium on*, Volume 2, pp. 1002–1006. IEEE.

Mplayer (2012, October 03,). Mplayer. URL: http://www.mplayerhq.hu.

OpenCV (2012, January 03,). Opencv. URL: http://opencv.willowgarage.com/wiki/.

Pentland, A., B. Moghaddam, & T. Starner (1994). View-based and modular eigenspaces for face recognition. In *Proceedings CVPR'94., 1994 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 1994.*, pp. 84–91. IEEE.

Raman, R., M. Livny, & M. Solomon (1998). Matchmaking: Distributed resource management for high throughput computing. In *Proceedings. The Seventh International Symposium on High Performance Distributed Computing, 1998.*, pp. 140–146. IEEE.

Ranjan, R., L. Chan, A. Harwood, S. Karunasekera, & R. Buyya (2007). Decentralised resource discovery service for large scale federated grids. In *Proceedings IEEE International Conference on e-Science and Grid Computing.*, pp. 379–387. IEEE.

Schopf, J., L. Pearlman, N. Miller, C. Kesselman, I. Foster, M. D'Arcy, & A. Chervenak (2006). Monitoring the grid with the globus toolkit mds4. In *Journal of Physics: Conference Series*, Volume 46, pp. 521. IOP Publishing.

Tannenbaum, T., D. Wright, K. Miller, & M. Livny (2001). Condor: a distributed job scheduler. In *Beowulf cluster computing with Linux*, pp. 307–350. MIT press.

Team, C. (2012, January 03,). Directed acyclic graph manager. URL: http://research.cs.wisc.edu/condor/dagman/.

Thain, D., T. Tannenbaum, & M. Livny (2005). Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience 17*(2-4), 323–356.

Tierney, B., R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, & R. Wolski (2002). A grid monitoring architecture. *Global Grid Forum Performance Working Group*.

Turek, J., U. Schwiegelshohn, J. Wolf, & P. Yu (1994). Scheduling parallel tasks to minimize average response time. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pp. 112–121. Society for Industrial and Applied Mathematics.

Turk, M. & A. Pentland (1991). Eigenfaces for recognition. *Journal of cognitive neuroscience 3*(1), 71–86.

Viola, P. & M. Jones (2004). Robust real-time face detection. *International journal of computer vision 57*(2), 137–154.

Wu, Y. & X. Ai (2008). Face detection in color images using adaboost algorithm based on skin color information. In *Knowledge Discovery and Data Mining, 2008. WKDD 2008. First International Workshop on*, pp. 339–342. IEEE.

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