Distributed Password Cracking Platform

Authors

Gerrie Veerman gerrie.veerman@os3.nl
Dimitar Pavlov dimitar.pavlov@os3.nl

Supervisors

Marc Smeets Smeets.Marc@kpmg.nl
Michiel van Veen vanVeen.Michiel@kpmg.nl

Universiteit van Amsterdam
System & Network Engineering

February 20, 2012

Abstract

This project originates from the need for distribution when performing security testing-related password hash cracking. KPMG IT Advisory uses an MPI-supported John the Ripper cluster plus a separate system with several graphics cards for the cracking of password hashes. As they want to expand their operations, they wish to integrate GPU-capable machines with the current cluster. Initial research determined that, currently, no practical solution exists for such a problem. This project focuses on the creation of a new middleware system that is specifically tailored for cracking password hashes with existing cracking tools, while distributing the task on multiple hardware architectures. After researching the possible architectural and communication models for such a system, we created a functional and technical specification, based on a centralised architecture and a message-oriented communication protocol. We also created a proof of concept implementation, using Apache, PHP, MySQL and SQLite. This project lays the foundation for further extending the concepts and code behind this distributed password cracking platform.

Contents

1 Introduction **1**

1.1 Related Work, Motivation & Goal . . . 1
1.2 Project Requirements . . . 2
1.3 Research Question . . . 2
1.4 Scope, Time & Approach . . . 3
1.5 Report Structure . . . 4

2 Theoretical Definitions and Reasoning for Research **5**

2.1 General Definitions and Terms . . . 5
2.2 Why Existing Distributed Systems Are Not Suitable . . . 10
2.3 Which Research is Needed for This Project . . . 13

3 Theoretical Research **14**

3.1 Distributed System Architectures . . . 14
3.2 Coordinator-Worker Communication Models . . . 20
3.3 Existing Cracking Tools Overview . . . 26
3.4 Summary . . . 29

4 Functional Requirements Specification **30**

4.1 System Usage Patterns . . . 30
4.2 Functional Requirements . . . 32
4.3 Detailed Requirements . . . 34
4.4 Summary . . . 37

5 Technical Design Specification **38**

5.1 System Architecture Overview . . . 38
5.2 Controller Components – Models and Workflows . . . 42
5.3 Worker Node – Workflow & States . . . 51
5.4 Platform Communication Protocol . . . 57
5.5 Overall Platform Operation . . . 59
5.6 Summary . . . 61

6 Proof of Concept **62**

6.1 Overview & Scope 62

6.2 Directory Overview and File Explanations 65

6.3 Getting Started with the Proof of Concept 67

6.4 The PoC Workings through the Different Use-Cases 69

6.5 Summary & Advice 71

7 Conclusion **72**

7.1 Theoretical Research – Architectural and Communication Models 72

7.2 Functional Requirements 72

7.3 Technical Design 72

7.4 Proof of Concept 73

7.5 Future Work 73

Appendices **75**

A Acronyms **75**

B Three-tier model **76**

C Peer 2 Peer (P2P) **77**

D Data Structures **78**

E Method Definitions **80**

F Proof Of Concept File Explanations **94**

G Bibliography **98**

Preface

We would like to thank our supervisors – Marc Smeets and Michiel van Veen, for guiding us through this project. Their advice and support were always useful.

We would also like to thank Marek Kuczynski for his interest in our work and his extremely helpful support.

Lastly, we would like to thank the whole IT Advisory department of KPMG for the friendly and pleasant working atmosphere they provided us.

1 Introduction

Advances in the graphics cards industry have led to the emergence of General-Purpose Graphics Processing Unit (GPGPU) computing. GPGPU refers to the use of a regular graphics card's Graphics Processing Unit (GPU) for the execution of non-graphics-related computing tasks [1]. Compared to Central Processing Unit (CPU) computing, GPGPU computing allows for a significant increase in the computation capabilities for certain tasks – tasks that can be parallelised and do not require much input-output activity. In the field of information security, GPGPU has been shown to be applicable and relevant [2][3].

This project was conceived by the IT Advisory department of Klynveld Peat Marwick Goerdeler (KPMG). One of the primary tasks of the IT Advisory department is security testing. As part of such activities, multiple members of the department need password cracking resources simultaneously and multiple cracking machines are needed. For this reason, the IT Advisory department has created a password cracking infrastructure that supports security testing activities. The infrastructure relies on CPU computation power for the cracking of password hashes. This infrastructure comprises of a cluster of 30 computers (nodes), which is controlled via the Message Passing Interface (MPI) by a controller node, running *John the Ripper*. In addition to that, KPMG also has a separate infrastructure with GPGPU capabilities. However, the GPGPU-capable infrastructure is limited in its size, scalability and control capabilities. It is also not connected to the main password hash cracking infrastructure – the CPU-based cluster, and they cannot work together.

It is KPMG's principal intention to improve the capabilities of their infrastructure. Unification of the Central Processing Unit-based and Graphics Processing Unit-based infrastructure is desired. However, previous research has shown that a single unified and stable solution that combines distribution, CPU and GPGPU capabilities does not currently exist [3]. All of the existing solutions are either proprietary and closed source, of low development quality or lack distribution capabilities.

KPMG has decided that to combine the CPU and GPGPU infrastructures, a middleware layer must be designed. The middleware should be capable of using existing password cracking tools for cracking hashes on both CPUs and GPUs. It should be possible to integrate KPMG's current infrastructure within this middleware layer.

1.1 Related Work, Motivation & Goal

KPMG has supervised the research of GPU-based password cracking in the past – in 2010 [2] and in 2011 [3]. That research shows that GPGPU password cracking has advantages for certain types of password cracking attacks. However, the papers also show that a single unified cracking tool which is stable, open source, distributed and supports both CPU and GPGPU computation does not currently exist. For this reason, KPMG wants to research the possibilities for a solution to this problem.

The main goal of this project is to lay the foundation needed for the creation of a distributed password cracking system. This system will act as a distribution middleware layer, that is responsible for the resource mapping, cracking strategy, communication between the nodes and handling of user input

in a correct manner.

Since KPMG works with sensitive data, a proprietary closed source solution is not desirable. Also, non-distributed solutions are not useful in the context of password hash cracking. In addition to that, the platform needs to be stable enough for KPMG to use in a production environment supporting multiple concurrent users. Lastly, the cracking platform must allow for both CPU and GPU computing power.

1.2 Project Requirements

Ideally, the solution should allow for the following:

- Possibility to use existing CPU and GPGPU cracking tools
  – cracking tools of high quality exist, but are not distributed; the requirement is to be able to use these tools in a distributed manner
- Size scalability – the possibility of adding more nodes
  – the platform should support the addition of new nodes while existing nodes are operational
- Access transparency – self-management and automatic resource mapping
  – the system should be capable of deciding how much resources it needs and how these resources should be utilised
- Adjustable cracking strategy
  – the system should be modular in such a way, as to allow for easy changes in the cracking strategy
- OS and hardware independence for the worker nodes
  – the platform should use tools and frameworks that work on multiple hardware architectures and operating systems
- Extensible support for more existing cracking tools which allow for more hashing algorithms
  – the addition of new tools that support more hashing algorithms should be easy and should not have an impact on the whole system
- Platform resilience
  – the platform should be fault tolerant and resilient to changes in the execution environment

This project provides the theoretical and technical base for the actual production-ready implementation of this platform.

1.3 Research Question

After examining the project requirements, we formulated the following research question:

> *How can a scalable, modular and extensible middleware solution be designed for the purpose of password cracking, so that it is based on existing cracking tools and allows for the use of distribution and of a dynamic and adjustable cracking strategy?*

We identified the following subquestions, which help to answer the main research question:

- What are the best architectural and communication models to use in such a system?
- What are the functional requirements of such a platform?
- Based on the functional requirements, what is the best technical design for such a platform?
- *Optional:* Can a test implementation (proof of concept) be created based on this research?

1.4 Scope, Time & Approach

This project has a predetermined fixed duration of four weeks.

This project provides a complete description of:

- The logical components required for a distributed password cracking platform,
- The interaction between these components,
- A possible set of software packages and custom-built components that implement the identified logical components.

Due to a small time frame, the functional and technical specifications of some components are examined from a high-level point of view – most notably, the resource mapping and strategic components. However, the platform architecture is designed in a modular way, and the addition/extension of these components is possible in the future.

To be able to answer the research questions, we undertook a theoretical approach for the identification of the required logical components and the interaction between them. Theoretical knowledge was gathered by examining existing papers, and by taking into consideration current *de facto* standards and best practices in the fields of password hash cracking and distributed systems.

Although focusing on the creation of a new solution, we examined the features of existing distributed solutions and platforms to determine whether they can contribute to a solution of the project's requirements. We concluded that existing platforms cannot be used within the context of this project and confirmed that a completely new system is required (see Section 3.3).

In a distributed system, the architectural decisions and the communication paradigms are the most important aspects of the system's design. Therefore, we began examining architectural and communication concepts to identify suitable ones for such a distributed solution (see Section 3).

Having identified the most suitable architectural and communication models for this project, we proceeded with creating the functional and technical specifications for the new platform (see Sections 4, 5). With the functional and technical specifications completed, we started with the creation of a proof of concept (see Section 6).

1.5 Report Structure

This report covers the theory, the research and the outcome of this project. The report is structured as follows:

- *Chapter 1:* **Introduction** – the current chapter, introducing the project, its requirements and the research question
- *Chapter 2:* **Theoretical Definitions and Reasoning for Research** – providing the required theoretical definitions for this research and examining why this research is needed
- *Chapter 3:* **Theoretical Research** – examining existing distributed systems, communication paradigms and existing cracking tools for their applicability to this project
- *Chapter 4:* **Functional Requirements Specification** – providing the functional specification for the project
- *Chapter 5:* **Technical Design Specification** – providing the technical specification for the project
- *Chapter 6:* **Proof of Concept** – providing details surrounding the Proof of Concept implementation
- *Chapter 7:* **Conclusion** – the research conclusions and project outcome
- **Appendices**

2 Theoretical Definitions and Reasoning for Research

This chapter provides the necessary theoretical background for understanding the topics discussed in latter chapters. It also outlines the results of previous research regarding existing distribution tools and the reasoning behind the creation of a new middleware solution for the distributed password cracking platform.

Firstly, the most relevant definitions and concepts needed for understanding this document are described. Afterwards, in Section 3.3.1, we look into existing distribution tools and middleware, and explain why current distribution tools are not adequate for this project. The last section of the chapter explains what new knowledge is needed to accomplish the goals of this project.

2.1 General Definitions and Terms

This section explains the basic definitions and concepts needed for understanding this document. Firstly, the definition of hashes is given, along with the use of hashing in information security. Afterwards, password cracking and cracking attacks are discussed and Section 2.1.3 explains what are the qualities that make a password secure. The concept of distribution is discussed in Section 2.1.4, along with the theory behind cracking strategies and resource mapping. Lastly, relevant hardware available for password cracking is taken into consideration.

2.1.1 What Hashes Are

The output values of so-called cryptographic hash functions are called *hashes*. Cryptographic hash functions have the property of taking an input of arbitrary length and converting it to a fixed-length output value - the hash. Another property of cryptographic hash functions is that they are one way functions – it is nearly impossible to determine the input value by examining the output value.[4]

Hashes are useful in the field of information security because of their “one way” properties. In computing, hashes are mostly used to store a representation of a user’s password – the password hash, on the authenticating server. This allows for the server to authenticate the user without storing the plain-text password in its database. In the example below, the word “rabbit” is shown – hashed using the hashing algorithm MD5.

```
md5(rabbit) => a51e47f646375ab6bf5dd2c42d3e6181
```

Different Hashing Algorithms

Different hash functions use different mathematical functions for the creation of hashes. Since different mathematical functions are used, the computational power necessary for execution differs between hash functions. Also, some algorithms allow for a varying number of “rounds” (calculation repetitions), which also contributes to this difference. The more rounds there are, the more computational power is needed for producing a hash.

Hashing algorithms should have a very low probability of producing the same output value for two (or more) input values. The occurrence of this case is called a *collision*. In information security, collisions are not a desired feature for a hashing algorithm. These may allow an attacker to find an input value, which, after hashing, results in the same output value as the hash of a particular password, stored in an authenticating server's database, thereby allowing the attacker unauthorised access.

Other problems, related to hashing algorithms, include the possibility for side channel attacks, the existence of "shortcuts" in the computation of a hash and the speed of calculating the hashes. All of these may facilitate an attacker in calculating many of the possible hashes and identifying the stored password via an exhaustive search (see the following Section).

New hashing algorithms are needed when older ones are found to be flawed – that is, they exhibit the problems, listed above. This is the reason why new hashing algorithms are constantly being developed and perfected.

2.1.2 Password Hash Cracking

As mentioned in the previous section, user passwords should not be stored in their plain form on authenticating servers – passwords should be stored in their hashed form. *Password cracking* is an activity that, by examining the hash of a certain passwords, aims to guess or derive in some way the plain form of the password. Password cracking is usually attempted by malicious individuals who try to gain unauthorised access to a system or a network.

There are various methods for retrieving the plain password back from a hash. The most appropriate attack method for a particular password hash depends on the actual password being used, as well as on the information the attacker has about the password and about the mechanisms used for storing it. Several possible cracking attacks are explained below.

Exhaustive Search

An *exhaustive search* attack (commonly referred to as *brute-force* attack) is an attack, which methodically tries all possible combination of an alphabet, within a certain keysize, in an attempt to guess the password. This attack exhausts the key space that the password uses (see 2.1.3), making the attack 100% successful. However, as passwords get longer the time needed to perform an exhaustive search grows exponentially, which makes the attack impractical to use for long passwords and complex alphabets. [5]

Dictionary Attack

Dictionary attacks use a predefined list of common words or expressions in an attempt to guess the password. These attacks rely on the assumption that users use common words for their passwords – presumably, because these are easy to remember. As users are known to frequently use common words or number combinations as their password [6], the probability that this attack will be successful is high. The downside of this attack is that a small modification to the common word used may render the attack useless. [7]

Rainbow Tables

Rainbow tables are data structures that contain precomputed hash values and their corresponding key values in a table. The said values are stored in such a way, so that it is possible to perform lookups for known hashes and determine the plain-text values that resulted in these hashes.

Rainbow tables are limited to a particular key length and character set. Nevertheless, when working within the constrains a particular table is designed for, such a table may decrease the time needed for cracking a password hash dramatically. [8][9]

2.1.3 How to Make a Password Secure

How secure a password is depends on certain characteristics. These characteristics are explained and discussed below.

Password Length

Length is one of the most important characteristics of a password. It determines how hard it is for an attacker to try all the possibilities when trying to crack the password (see Section 2.1.2). The shorter the password is, the smaller the total number of possible passwords of that length is, which means that an attacker will need less time to try them all. This means that long passwords may take longer to guess by an attacker.

Password Character Set Used

Besides the length of a password, the *character set* used is also important. The larger the character set, the more password combinations there are. For this reason, using large character sets, which include special characters along with letters and numbers, is considered more secure. The most frequently used character sets include: [10]

- Digits (0-9), Set Size: 10
- Lower-case (a-z), Set Size: 26
- Upper- and Lower-case letters plus digits (A-Z, a-z, 0-9), Set Size: 62
- All standard keyboard characters, Set Size: 94

Total Key Space

The *key space* is the set of all available possibilities. If we consider a possible length of 6 and a character set of 36 characters, one can simply calculate that all possible combinations for this key space are $36^6 = 2176782336$. One can specify different key spaces when working with cracking tools. Cracking tools can calculate the number of possible combinations in the key space and then estimate the total time it would take to calculate and check all keys for a certain key length.

Salting a Password

Salting refers to the technique of a password being augmented by appending or prepending a string (called the *salt*) to the password before calculating its hash. It is possible that the added salt depends on some arbitrary ordering of the hashes.

Salting dramatically increases the keyspace for the password. Also, when cracking multiple hashes at the same time, it becomes impossible to test multiple passwords hashes with a single computed hash – even if the stored passwords are the same, their hashes will be different because of the salt. This makes all types of attacks impractical, since the password hash now represents the salted password. For dictionary attacks, this means that new dictionaries have to be created, that contain possible *combinations* of common words and possible salts. In the case of rainbow tables, a new set of precomputed hashes needs to be generated, which also takes the salt into consideration. Regarding exhaustive search, when salting is used, the key space increases and it will take longer before the plaintext password can be found.

2.1.4 Distribution and Cracking Strategy

Distribution, or *distributed computing*, is the concept of organising multiple machines in a way that allows for coordination and collaborative execution of a task on these machines. The classical approach for obtaining more computational power is to equip faster hardware on the used machine. However, this idea may eventually become limited by various factors – price, hardware availability, platform availability, geographical location, etc. Using a distributed approach allows for multiple computers to be connected to each other with the purpose of increasing performance. In that sense, a distributed system should be scalable and modular. In recent years distributed systems have gotten more popular and have been developed for various purposes. [11]

Distributed systems may be used successfully for computational problems – for example in the fields of password cracking, mathematics, bioengineering and astrophysics. In these fields, a lot of computational power is needed. Ideally, a distributed system can use any computer's computational power as a resource to gain more performance for a specific task. A distributed system with multiple connected computers for the purposes of computation is called a *cluster* [11]. It is within the distributed system software's responsibilities to control this cluster and all of the connected computers in a manageable manner.

What Cracking Strategy and Resource Mapping Are

The term *cracking strategy* refers to the sequence of steps undertaken to approach the cracking of a password hash. It depends on the combination of hash type, password keyspace (explained in Section 2.1.3), and on other knowledge regarding the calculation and storage of password hashes.

Resource mapping is a term that describes the allocation of different resources and the dispatching of tasks to these resources. As explained in Section 2.1.4, a system may make use of many computers for the purpose of increased computational power. A resource mapping module may be the part of a distributed system that determines which part of a computational task should be executed by a particular part of the distributed system.

For example, it could be the case that in a distributed system there are computers that differ in their capabilities. As tasks can have different computational requirements, it should be determined which computer(s) within the distributed system are best suited for executing that task. A distributed system's use of optimised resource mapping allows for a greater increase in performance since tasks will make use of the most suitable resource(s) available.

2.1.5 Hardware Available for Password Cracking

Different techniques for cracking hashes are possible, as described above. However, for retrieving a password, some techniques are more efficient when a particular hardware type is used. Below, the different types of hardware that could be used for password cracking are discussed.

CPU Hardware

A CPU is the main computing element within a computer system. It has general purpose computing capabilities, as it performs all the calculations required by the software on the system.

GPU Hardware

A GPU is the graphics processing unit within a computer system. The GPU is a specialised piece of hardware that is designed to perform graphics computations fast and in parallel.

Before 2006, GPU hardware could only be used for graphics-related processing – that is, the rasterisation of 2D and 3D vector graphics for the purposes of displaying them on a screen. However, in 2006 Nvidia® launched the Compute Unified Device Architecture (CUDA®) – a computing architecture making use of GPU hardware for general purpose computations, which was the first GPGPU solution.

While consumer CPUs have 1, 2 or 4 cores, consumer GPUs have as many as 1024 cores[12]. This makes them extremely fast, when compared to CPUs, in performing general purpose tasks that can be parallelised. However, the specialisation of GPUs also results in limitations when compared to CPUs – GPUs generally have a small I/O bus and limited memory available per core. This means that, while GPU hardware may be several times faster than CPU hardware for some tasks, for other tasks it may be several times slower than CPU hardware[2]. With relation to password cracking, it has been determined that GPGPU-based computation is best suited for exhaustive search attacks, while CPU-based computation is more suitable for other types of attacks, such as rainbow table or dictionary attacks [3].

Field-Programmable Gate Array (FPGA)

A FPGA is a type of integrated circuit, which can be reprogrammed after production. FPGAs can be programmed to implement any logical function or a sequence of logical functions that can be implemented with a regular Integrated Circuit (IC).

The advantages of using FPGA stem from the fact that computations are done in hardware and, thus, are performed much faster than when performed by software. FPGAs may outperform both

CPU and GPGPU hardware in terms of speed of computation. However, FPGAs have some limitations. Because of the way they operate, they need to be reprogrammed for different tasks. Applied to password cracking, FPGAs have to be reprogrammed for every algorithm that requires computation. Coupled with their higher price (when compared to CPUs and GPUs), they may prove to be an impractical solution in many cases.

Cell Architecture

Cell Broadband Engine Architecture (CBEA), commonly referred to as simply Cell, is a computer architecture, based on the RISC architecture, which uses a single conventional PowerPC core, acting as a controller, with 8 simple Single Instruction Multiple Data (SIMD) cores, acting as workers. Each of the worker cores has local memory.

Research has shown [13] that Cell can be from seven to sixty-five times faster at certain calculations when compared to conventional processors. Related to password cracking, Cell can lead to performance increases for the calculation of hashes mainly because it is based on the SIMD paradigm. However, there are several disadvantages to the Cell architecture, which make it impractical to use. Firstly, there is not much active development in this field, while other fields (notably – CPU and GPU hardware) see active development. Also, the price of Cell hardware is considered relatively high when comparing with the capabilities and price of recent CPU and GPU hardware. Finally, the availability of Cell hardware in the consumer market is quite limited when compared to regular CPU and GPU hardware. The combination of these disadvantages makes the Cell platform an impractical one for the purposes of password cracking.

2.2 Why Existing Distributed Systems Are Not Suitable

Existing distribution tools and middleware solutions are discussed in this section and we explain why these tools are not suitable for the current project. Information given about Berkeley Open Infrastructure for Network Computing (BOINC) and MPI is based on previous research. [3] Information about Jungle Computing and Cloud Computing is based on a paper study. [14]

2.2.1 BOINC

BOINC is an open source middleware system for volunteer and grid computing. [15]

> “Apart from all its advantages, a BOINC server for a custom computational project requires time and technical skills to be set up which makes BOINC not an out-of-the-box solution for password cracking. A BOINC project consists of a custom application which is spread along client nodes along with an input file (work-unit) for processing. When an input file which contains arguments to the aforementioned application is processed, a new one is dispatched by the scheduler server and this repeats until all client nodes have processed all work-units. Even though the BOINC Application programming interface (API) supports functions for managing the work load (work-units) to client nodes, the BOINC API lacks functions for managing individual projects. [3]”

As can be determined from previous research, BOINC is not suitable for the current project, as it brings a large deployment overhead due to its complexity.

2.2.2 MPI

MPI is a portable message-passing system that is used in a wide variety of parallel computers (for message-based communication see Section 3.2.4). MPI remains the dominant system used in high-performance computing today. [11]

> "When two processes want to exchange data over the network, they must ex-change messages because the processes do not share memory. MPI manages this time-consuming operation. Therefore, it is not suitable for thread synchronisation with shared memory. However, brute force password cracking can be implemented using only coarse grain parallelism as it is known as "embarrassingly parallel". MPI master would split up the searched key space into partitions and send these to client nodes which will process them independently. An MPI implementation can be compiled together with CUDA and Stream to support one or multiple GPU cards on client nodes. [3]"

It is shown that MPI can be useful for brute-force cracking. However, MPI is most useful when the MPI-controlled programs support MPI and are capable of collaborating. To achieve this, existing programs need to be modified, which may not always be possible due to some cracking programs being closed source.

It has to be noted that *John the Ripper* includes MPI support. [16]

2.2.3 Jungle Computing with Ibis

Jungle computing refers to the use of diverse, distributed and non-uniform high-performance computer systems to achieve improved overall performance. [17]

> "The Ibis platform [18] aims to combine all of the stated fundamental methodologies into a single integrated programming system that applies to any Jungle Computing System. Our open source software system provides high-level, architecture- and middleware-independent interfaces that allow for (transparent) implementation of efficient applications that are robust to faults and dynamic variations in the availability of resources.
>
> The aim of the Ibis platform is to drastically simplify the programming and deployment of Jungle Computing applications. To achieve this, Ibis integrates solutions to many of the fundamental problems of Jungle Computing in a single modular programming and deployment system, written entirely in Java.
>
> Despite the successes, and the fact that to our knowledge Ibis is the only integrated system that offers an efficient and transparent solution for Jungle Computing, further progress is urgent for Ibis to become a viable programming system for everyday scientific

> practice. One of the foremost questions to be dealt with is whether it is possible to define a set of fundamental building blocks that can describe any Jungle Computing application. [19]"

JavaGAT is used as the middleware layer for Ibis:

> "Java Grid Application Toolkit (JavaGAT) offers a set of coordinated, generic and flexible API for accessing grid services from application codes, portals, data managements systems, etc. JavaGAT sits between grid applications and numerous types of grid middleware, such as Globus, Glite, SGE, SSH or Zorilla. JavaGAT lifts the burden of grid application programmers by providing them with a uniform interface. [20]"

Jungle Computing can require a large development effort – especially when high-performance is needed. Ibis tries to provide basic functionality for efficiency and transparency, and overcomes most of Jungle Computing's complexities. However, Jungle Computing and Ibis are targeted at providing a development environment for new high-performance computing applications. As the current project aims at using existing password cracking tools in their current form, we consider Jungle Computing to be unsuitable for this project.

2.2.4 Cloud Computing

Cloud computing refers to a paradigm in which massively scalable IT-enabled capabilities are delivered "as a service" to external customers, using Internet technologies. Cloud infrastructures can be extended arbitrarily while they are operating, and are making use of principles from the field of distributed computing.

There are two types of clouds – public and private. Public clouds provide a shared infrastructure to their users. This means that all the users' data, although logically separated, resides on the same medium, which creates the possibility for a side channel attack. With private clouds, the infrastructure used for the could is provided exclusively to the user. Although, this mitigates side channel attacks to a certain extent, the infrastructure is still typically managed by an external party. [21]

Cloud computing allows for easy and cheap access to high-performance computing environments and is promising for cracking password hashes. It may certainly be feasible to explore this possibility in future research. However, for this project, targeted mainly at KPMG, it is not possible to use a cloud environment, as that would violate KPMG's data storage policies.

2.2.5 Summary

After examining several distributed systems and their capabilities, we concluded that, at the time of writing, there is no distributed system available that meets this project's goals for password cracking. A new distributed system needs to be researched, designed and implemented.

2.3 Which Research is Needed for This Project

In this chapter we introduced the basic theoretical concepts necessary to understand this project and provided an overview of the findings of previous research, which explained why existing distribution tools are not suitable for the current project. A new design for a distributed password cracking platform was needed. This section outlines the research that needed to be performed to achieve the project goals (see Section 1.2).

Completion of the rest of the project goals depended heavily on the chosen *system architecture* and *communication model*. Therefore, we needed to examine possible system architectures and communication models, and to determine which combination of the two would suit this project best. While investigating these problems, we needed to take into consideration design issues including distribution transparency, scalability, fault tolerance, reliability and modularity. Also, we needed to consider multi-platform tools and frameworks to allow for Operating System (OS) and hardware independence.

In the next section we discuss architectural and communication possibilities and examine existing cracking tools, their capabilities and applicability.

3 Theoretical Research

In the previous chapter we examined existing solutions and explained why they are unsuitable in the context of this project and that a new solution is required. This chapter contains the theoretical research, required for the creation of a new system. Our research examines possible system architectures and communication models to identify the most suitable ones for this project. Having identified those, it is possible to create the functional and technical specifications needed.

Firstly, to determine the most suitable architecture for our system, we examine different types of system architectures, their advantages, disadvantages and applicability. Afterwards, in Section 3.2, we identify the the type of communication model that the new system is to use. Current existing cracking tools are discussed in Section 3.3. The chapter concludes with a summary of the research findings and outlines the architecture and communication models that will be used in the platform.

3.1 Distributed System Architectures

This section examines the different architectures that may be used within a distributed system. To be able to identify the most suitable architecture for our platform, we analyse the strengths, weaknesses and applicability of each architecture type. Firstly, we compare centralised and decentralised architectures. Afterwards, we take into consideration design issues such as scalability, modularity and concurrency. Finally, we conclude with a summary of the findings and an outline of the architecture of the distributed password cracking platform.

3.1.1 Centralised vs. Decentralised Architectures

The concept of centralisation deals with the functions that each computer system has within the distributed system. If all of the *nodes* (connected computers) within the distributed system have identical capabilities and can be interchanged freely, then the system is considered decentralised. If one or a few of the computer systems within the distributed system perform exclusive tasks (that cannot be performed by other systems), then such a system is called centralised. [11]

Centralised Architectures

Centralised distributed systems architectures are typically based on the Client-Server (C/S) model. In such an architecture, there are two distinct roles – that of the client and that of the server. The server is the component, that acts as a source of information. It may provide a service or act as an access point to an infrastructure. It may also act as a controller, issuing commands to the clients. Typically the server has no knowledge about the clients – the clients are expected to make themselves known to the server by sending a request message. As shown in Figure 1, the server component may consist of several physical computer systems.

Within a C/S architecture, *application layering* may be used. Application layering means that the flow of requests and responses is layered (see Figure 2a) – data can only go up and down between the different layers and is not shared between systems residing at the same layer. Also, skipping of

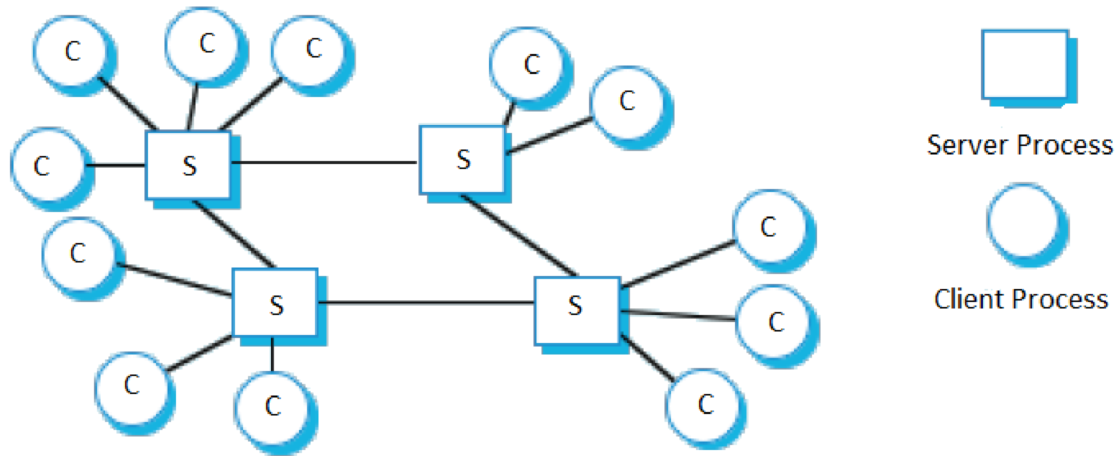


Figure 1: Client-Server architecture with multiple server systems. [22]

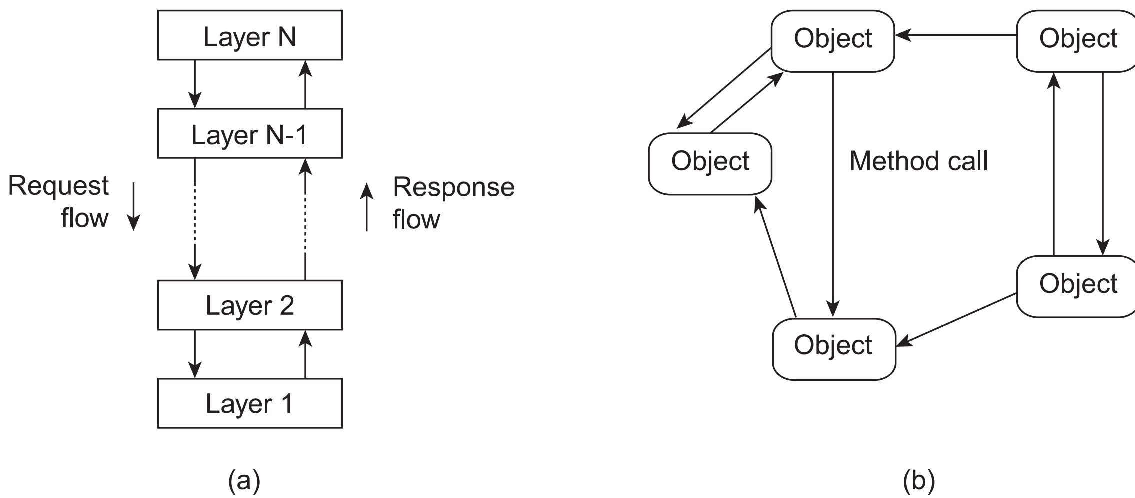


Figure 2: The (a) layered and (b) object-based architectural style. [11]

layers is not possible. One model employing application layering is the three tier model on which many web-based systems are based (see Appendix B).

Advantages and Disadvantages The main advantage of the centralised approach is its simplicity. There are clearly defined roles, which means that functionality can be divided and implemented separately. Also, this allows for more flexibility, as the server implementation may be modified without affecting the clients. The same holds for the client implementation.

The most significant disadvantage of centralised systems is the fact that a single component has functionalities, which cannot be handled by other components. This may lead to scalability limi-

tations – as more client nodes are added to the system, the server component will be expected to handle more requests simultaneously. Since the server component can only handle a finite number of requests simultaneously, there is a theoretical and practical upper limit to the number of clients that may be supported by a single server.

Another concern with centralised systems is the fact that the server component is a single point of failure. If the server system fails, the whole distributed system stops functioning, as no other system has the capability of handling the server's work.

Applicability The centralised approach is applicable in a wide variety of cases. Many systems use the centralised approach because of its simplicity. The main example is websites, where the servers handling the incoming request are the central part of their infrastructure. These requests are then passed on to application and database servers. Many applications and application protocols also make use of centralised architectures.

Decentralised Architectures

In decentralised architectures, all components in a distributed system typically have the same functionality and cooperate to achieve a task. [11] Compared to centralised architectures, using a decentralised architecture means that every computer system is both a client and a server, with different systems providing services to each other. Decentralised systems are more complex than centralised systems – as all components are the same, functionality cannot be divided and implemented separately. However, since all objects are the same and components can be interchanged, decentralised systems are very flexible. One example of a distributed architecture is Peer 2 Peer (P2P) (see Appendix C).

Advantages and Disadvantages The main advantage of a decentralised system, when compared to a centralised one, is the avoidance of the single point of failure problem. Since all components have the same functionality, components can be interchanged easily. This means that the failure of any single component will not lead to the failure of the distributed system as a whole. Also, since there is no central management component (such as the server in C/S architectures), the problem of scalability can be overcome – the addition of new nodes does not necessarily increase the workload of the other working nodes. However, this depends on the concrete system.

The main disadvantage of the decentralised approach is the complexity of design and implementation. Decentralised systems employ decentralisation-specific paradigms to allow for all components to operate correctly without a central control and coordination point. In a decentralised system, nodes both provide and request data from each other and act both as clients and as servers. Also, every node is expected to be self-controlling. This may lead to complex designs and implementations.

Applicability As decentralised architectures are a fairly new concept, they are used mostly to support newer technologies where scale or manageability can be an issue. The most widely used decentralised system is BitTorrent which uses P2P for exchanging files. Another example is Skype which make uses a hybrid P2P architecture.

3.1.2 Architectural Design Issues

This project's requirements include access transparency, scalability, concurrency and modularity. In the context of distributed systems, these issues relate to the system's architectural design. The following sections elaborate on each of the goals and explain how they can be achieved.

Access Transparency

One of the characteristics of a distributed system is to be composed of multiple separate computer systems, but to appear as a single coherent system to its users. This property of a distributed system is called *distribution transparency*. One specific aspect of distribution transparency is relevant to the current project – *access transparency*. [11]

Access transparency deals with the way the users of a distributed system access its resources/capabilities. A distributed system is said to implement access transparency if access to its resources/-capabilities is always provided at a single well-defined access point. If this is the case, the users perceive the system to be a single coherent system, while, in fact, it is distributed.

Access transparency can be achieved by using a centralised approach – a single system accepts all user requests and then designates other systems for processing the request. Another approach to access transparency is to map a single access point name to multiple systems' addresses. Then, when requesting access via the access point name, the users will transparently gain access through one of multiple access points.

Scalability

Scalability can be defined as:

> “... the ability of a system, network, or process, to handle a growing amount of work in a capable manner, or its ability to be enlarged to accommodate that growth. [23]”

In other words, for a system to be considered scalable, it must allow for the expansion of its capabilities.

When considering centralised distributed systems, scalability becomes a problem when the number of nodes that rely on the central system is so large, that the central component gets overloaded. One approach to improving scalability is to improve the capabilities of the central component. However, there is a theoretical upper limit to the capabilities of a single system. Another approach would be to reduce the workload of the central component. This can be done by setting up multiple identical servers that operate on shared data and make use of access transparency.

In the case of decentralised distributed systems, scalability can be a problem when there is broadcast communication between the nodes in the system. When the number of nodes becomes larger than a certain threshold, the underlying network can no longer support the volume of data being transmitted between the systems. In that case, scalability can be improved by decreasing the amount of broadcast communication, or by dividing the nodes in different sub-trees.

Concurrency

Concurrency refers to the simultaneous execution of different calculations on the same system. In distributed systems, concurrency may also refer to the simultaneous access by different users or programs to the same resources. Concurrency needs to be considered *before* a system is designed, as access to a resource may prevent other users from accessing that same resource.

Concurrency can be implemented in several ways. However, research has shown that one of these techniques, using threads, is the most suitable one for resolving concurrency problems [24]. Threads allow many users to connect to the same process by letting every user have her own "sub-process" – her own thread. With this mechanism, it is possible to have a single process serve multiple users without the users having to wait for each other to finish.

Architectural Modularity

A modular architecture is one, that has been designed and implemented using modular programming. Modular programming aims to break down a program's functionality into blocks, which implement a subset of the whole functionality – modules [25]. The functionality in these modules can be modified and reused, and as long as they communicate properly with the other modules in the system, the system will operate properly.

Modularity is a desired feature within a distributed system, as by using modules, it is possible to achieve greater flexibility. Parts of the system can be modified without affecting other parts of the system. Functionality can also be extended in a standardised way.

Modularity can be achieved by organising the desired functionality of the system into logically separate blocks. These blocks can later be implemented in software as modules.

Fault Tolerance

Another design requirement is fault tolerance, which refers to a system being able to continue functioning properly despite partial-failures. When a failure occurs, the system should recover and continue functioning. A fault tolerant system is characterised by high availability and reliability.

Fault tolerance is typically implemented through replication [11].

3.1.3 Architecture of the Distributed Password Cracking Platform

Two architectural paradigms were described – centralised architectures and decentralised architectures; we also described several design issues related to the project requirements – access transparency, scalability, concurrency and modularity. In this section, we cross-compare the architectural types with the discussed design issues and make a conclusion about the most suitable architecture for the distributed password cracking platform. In the comparison, we also include an extra factor – simplicity.

Design Issue	Centralised	Decentralised
Transparency	++	–
Scalability	+	++
Concurrency	+	+
Modularity	+	+
Stability	–	+
Simplicity	++	– –

Table 1: A comparison between centralised and decentralised architectures.

Access Transparency

Centralised architectures allow for access transparency by setting up the centralised component to handle all incoming requests and respond as if the whole system is responding – to the user of the system it appears that a unified response has been issued, while in effect, only the central component is responding. It is also possible to employ name-based access transparency, where the naming system points each request to a different access point.

With decentralised architectures, access transparency is only possible if implemented into the user interface – each client is aware of the whole system, often operating on a shared data space, but the user of the system may not be aware of using a distributed system.

Scalability

Centralised architectures are generally more susceptible to scalability issues – the central component with the addition of new nodes. However, research has shown that often there is no real need to support arbitrarily large systems at all times. Therefore, the growth rate of the system needs to be taken into consideration [26]. Thus, a centralised solution may be preferred, despite its non-scalable nature, if the system is not expected to grow quickly.

Decentralised architectures, on the other hand, are less susceptible to scalability problems when compared to centralised ones. Generally, depending on the amount of broadcast communication occurring, decentralised systems are considered a solution for scalability issues with centralised systems.

Concurrency

Concurrency can be achieved by using similar techniques (see Section 3.1.2), in both centralised and decentralised architectures.

Modularity

Modularity can be achieved by using similar techniques (see Section 3.1.2), in both centralised and decentralised architectures.

Stability

The stability of a system can be increased when fault tolerance is implemented. For a centralised design, this can be troublesome, since there is a centralised component which may fail. However, the centralized component may be replicated to function as a backup. Regarding the client systems in a centralised architecture, fault tolerance there is easier to achieve, as their role is not central to the functioning of the distributed system. When a client system fails, it may simply restart its processes and make itself known to the controller again.

With a decentralised architecture, fault tolerance is easier to achieve – as many decentralised architectures focus on nodes joining and leaving, the system can handle unexpected node crashes. Also, since all nodes typically have identical functionality, the system can continue its operation when any node fails.

Simplicity

With regards to simplicity, researchers agree that centralised architectures are far simpler to design and implement than decentralised ones [11]. This is because of the fact that centralised architectures allow for the separation of functionality – both in the design and in the implementation of the system. In decentralised architectures, each participant in the system is typically has all of the functionalities (although there are exceptions), which typically leads to complex designs and implementations.

Conclusion

After examining two architectural types and comparing them with regards to several design issues, we came to the conclusion that a centralised architecture is most suitable for the needs of the distributed password cracking platform.

When comparing both architectures, they are mostly equivalent with regards to the design issues discussed – centralised architectures lack in scalability and stability, while decentralised architectures lack in access transparency. However, simplicity is an important factor for us. The fact that centralised systems require much less effort to be designed, implemented, deployed and maintained was the deciding factor for our conclusion. We concluded that is important that the system works correctly is easily maintainable and a compromise – suboptimal scalability and stability, is acceptable.

3.2 Coordinator-Worker Communication Models

In the previous section we identified the most suitable architecture for the distributed password cracking platform. In this section, we examine several communication models and their advantages, disadvantages and applicability. We then determine the most suitable model for communication within the platform.

In a distributed system using centralised coordination and control, the communication between the Controller and the Worker Nodes is one of the most important aspects to understand. The communication channel needs to be clearly defined and the exchange of data needs to follow strict formulations for the communication to be possible.

In a distributed system, the middleware is responsible for conducting communication between the nodes of the system. Middleware communication protocols support high-level functionality and communication services that improve distribution transparency. [11]

The needs of the distributed system determine the communication paradigm and the set of protocols used. These needs typically include transfer of data, database synchronisation, coordination and service invocation.

In the context of distributed systems, several communication paradigms are well recognised [11]:

1. Remote Procedure Call (RPC)
2. Stream-oriented communication
3. Multicast communication
4. Message-oriented communication

The following sections examine each of them and determine the most suitable one for the current project.

3.2.1 Remote Procedure Call

Remote Procedure Call is a method, which allows the transparent execution of procedures on a remote machine. With RPC, the application that is calling the remote procedure does so transparently. Most communication details are hidden from the application and are handled my the middleware. RPC can be implemented both as a synchronous and as an asynchronous mechanism. It is possible to support the passing of parameters and data structures.

RPC Operation

RPC operates by making use of *client stubs* and *server stubs*. These "stubs" are method implementations, on the client-side and on the server-side of the communication channel. They are implemented in such a way, as to resemble local procedures, but have the capability of sending and processing requests on the network.

When the client stub is called, it accepts the passed parameters and transforms them in a process called *marshalling*. This transformation allows for the parameters to be transferred via the designated communication channel. The client stub sends the request and the parameters to the server and waits for the reply.

At the server side, the OS accepts the message and passes it to the server stub. The server stub unpacks the parameters and interprets them, performing the requested task. Once the server finishes with its computation, the server stub performs marshalling on the results and sends them back to the calling system.

When the client receives the server response, it unpacks the parameters and returns the result to the calling function as if the whole process has taken place on the local machine (not taking delay and outages into consideration).

RPC Applicability

RPC fits well within general-purpose middleware solutions and is most suitable for creating a distributed execution environment for client applications to operate in. It allows for the transparent remote execution of procedures. It is also suitable, in its asynchronous form, for submitting execution tasks, which take a long amount of time to complete.

3.2.2 Stream-Oriented Communication

Stream-oriented communication is a type of communication that deals with the transfer of non-independent data units. With streaming, the sequence of transferred data units is only meaningful as a whole. Examples of stream-oriented communication include online playback of audio and video.

In this context, both the *timing* of each transmitted data unit, as well as the *transfer delay* play a crucial role to the whole communication process. It may be necessary to subject communication to predetermined timing constraints to allow for the correct interpretation by participating parties.[27]

Asynchronous, synchronous and isosynchronous streaming

Stream-based communication generally relies on both sequential transmission/reception of data units and on specific timing constraints, regulating delay. This type of communication can be divided in three classes – asynchronous, synchronous and isosynchronous. [11]

Asynchronous stream-based communication requires only that the different data units are transmitted and received in the same sequential order. There are no specific requirements related to timing, as it is not considered critical for the communication to be successful. This type of stream-based communication is used when transferring discrete data streams – for a file to be transferred successfully the only requisite is that the bytes are ordered in the correct way upon reception.

Synchronous stream-based communication requires that communication is subject to a maximum transmission time constraint as well as strict ordering. It is important that data units arrive in a timely manner, but it is not important whether they arrive faster than required or just on time. This type of communication has a maximum end to end delay constraint.

Isosynchronous stream-based communication requires that three constraints are in place – data unit ordering, minimum and maximum transmission times. It is important that data units arrive precisely when they are expected – neither later, nor earlier. This type of communication is important when transmitting audio or video for immediate playback. The audio or video stream must be received and played back in a defined way for the playback to be correct.

Applicability of Stream-Oriented Communication

Stream-based communication is mostly used within multimedia systems that deal with the real-time transfer of large quantities of data. This data may include large files, audio, video or other types of continuous data. Stream-oriented communication is also suitable in the cases when there is no clearly delimited end to the data being transmitted. The applicability of stream-based communication is

only limited to these cases, as the constraints that surround it are not relevant when dealing with atomic messages or a request-reply model.

3.2.3 Multicast Communication

Multicast communication deals with sending identical data to multiple receivers. Multicast communication can be implemented on the network level with networking equipment dealing with determining the path and delivering the data in an optimal manner. In the context of distributed systems, however, multicast communication is most often implemented at the application level, with the data traversing a virtual middleware-supported network.

General Operation

Application-level multicasting relies on the concept of an *overlay network*. The overlay network is a middleware-supported virtual network, which re-organises the links between the nodes and abstracts the inter-node connectivity. Virtual links are used for communication between the nodes. The virtual links may be defined with, or without taking into consideration the underlying physical network (see Figure 3).

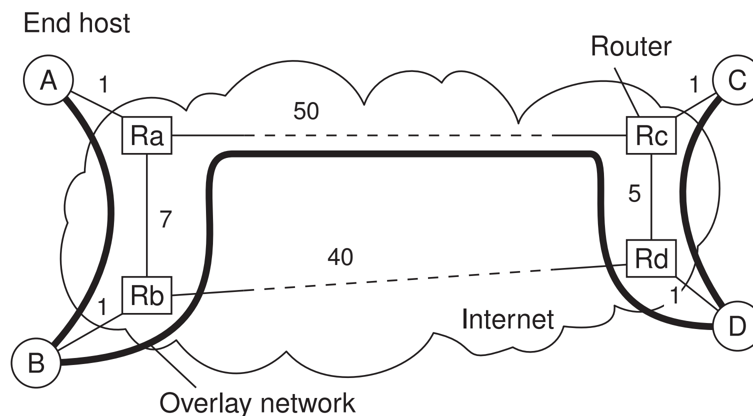


Figure 3: An example overlay network – both physical and virtual links are shown with weights. [11]

A classical approach to organising multicast communication is the tree-based approach. [28] The dissemination of data follows the principle of *subscribing* to a multicast tree and, subsequently, receiving data passed from other nodes that are part of the tree. With this approach, there is a source, one or more forwarders and one or more subscribers. Subscribers typically only receive data. However, when a subscriber receives a join request from a non-participating node, the subscriber becomes a forwarder for that node and is responsible to pass all received data to the new node as well.

A different approach to multicasting is the mesh-based approach. With this approach, each node is related to multiple other participating nodes and an algorithm is used to determine the best path to be used for information dissemination. While this approach is more robust and efficient, when

compared to the tree-based approach, it is also related to complicating the management of the system as a whole.

Applicability of Multicasting

Multicast communication is used mainly in the context of replication and synchronisation. It is most suitable when data needs to be transferred to a multitude of nodes. The advantages of this type of communication are felt most ostensibly when the number of participating nodes is large.

3.2.4 Message-Oriented Communication

Message-oriented communication models are considered simpler than the previously described models. Message-based communication deals with the transmission of messages, which carry data. All requests and responses passing through a communication channel are represented as messages.

General Operation

There are different types of message-based communication systems – transient and persistent. With transient message-based systems, the message needs to be delivered shortly after it has been sent with the time interval being dependent on a predetermined constant value. If delivery is not possible, the message is discarded. With persistent message-based systems, there is a mechanism of storing undeliverable messages until they can be delivered. In the context of this project, only the transient model is considered relevant.

Transient message-oriented systems rely on the possibility of delivery. After the message has been sent, the sender process usually waits until it either determines that the message has been delivered, or until a timer expires. The sender determines that the message has been delivered successfully if the receiver sends back a positive reply. If the message could not be delivered, the only option for the sender is to retransmit it.

One way of implementing message-based communication is to use *sockets*, which provide rudimentary communication facilities for the direct writing or reading of data to/from remote systems over a network. Although sockets are stream-based, with message-oriented communication, they are used as an IO-mechanism.

Message-Oriented Protocols

Message-oriented protocols are usually based on the request-response communication model. This model describes a message exchange pattern, in which the initiator of communication sends a request and expects a response from the receiver. Implementing protocols include the Hyper-Text Transfer Protocol (HTTP), the Simple Network Management Protocol (SNMP) and others. In all message-based protocols, a clear and strict message format can be observed and must be followed for communication to be successful.

Applicability of Message-based Communication

Message-oriented communication is generally applicable to a broad range of needs. It is best suited for use in systems, where the amount of data that needs to be transferred is small, or the exchanged data units are atomic in their nature. The request-response model is based on message-oriented communication.

3.2.5 Most Suitable Paradigm

The distributed password cracking platform aims to make use of the most suitable communication model for distributing of a cracking task on many computer systems. The communication requirements of the platform include:

- Control communication
- Task assignment
- Completion reporting
- Status reporting

With the exception of task assignment, the rest of the communication requires the transfer of small data units containing function names, parameters and status indicators. These data units are atomic, do not have a high bandwidth requirement and can fit well within the request-response communication model.

For the assignment of cracking tasks, one of the relevant data units is the hash values to be cracked. Such a data unit may impose bandwidth and latency requirements due to the amount of data that is to be transferred. However, such lists typically include less than 1000 entries, while the largest encountered lists have 150 000 entries. Typically, entries in these lists include a username and a password hash. Such lists are stored in plain-text files.

If we make the assumption that the average username is 8 characters long, the average password hash representation is 64 characters long and there are 2 extra characters acting as delimiters, then the filesize for 1000 entries is about 72 KiB, while the filesize for 150 000 entries is about 10.5 MiB (see calculation 1).

$$\begin{aligned} 1char &= 1byte \\ (8+64+2)*1000 &= 74000b = 72.27KiB \\ (8+64+2)*150000 &= 11100000b = 10.59MiB \end{aligned} \tag{1}$$

Request-response communication should still be able to handle this well. In the case of job assignments, the RPC model is more suitable than message-oriented communication – specifically, its

asynchronous variant, as that allows for non-blocking operation both at the server side and at the client side.

Having examined several popular communication paradigms and having outlined their operation and applicability, we can conclude that message-oriented communication based on the request-response model best suits the needs of this project. In the single case, where job assignment needs to be done, the RPC model is more suitable.

3.3 Existing Cracking Tools Overview

The concept of passwords and password hash cracking is not new. There is a plethora of existing software packages that deal with password cracking. However, a large part of those have been abandoned or have been merged into other projects. The most relevant non-distributed CPU- and GPU-based tools are discussed. Distributed password cracking tools are in their infancy, but are also looked into briefly.

3.3.1 CPU-Based Cracking Tools

CPU-based cracking tools have a longer history than GPU-based tools and are, thus, considered more mature with regards to stability, offered functionality and openness. There are various tools with various capabilities in existence. The most widely used cpu cracking tools include:

- John the Ripper
- Cain and Abel
- RainbowCrack

Each of these is briefly discussed and the advantages and disadvantages of each are outlined.

John the Ripper

John the Ripper [29] (commonly referred to as simply John) is one of the most popular password cracking tools available. It is open source and runs on multiple operating systems. John supports all of the most used hash types as well as many hash types, which are not frequently used. [30] John supports dictionary, brute-force attacks (see Section 2.1.2) and custom attacks. John's functionality can also be extended with modules.

There is an unofficial patch for John, which adds MPI support (see Section), which allows for distributing the task of cracking over multiple machines. KPMG uses the patched version of the program with limited success.

Despite its advantages, John the Ripper makes use of CPU-based computational power and has no standard support for using GPGPU. There is an unofficial patch available that adds GPGPU support to John, but the development quality of the patch is still too low, especially when it is to be considered as usable in a production environment.

Cain and Abel

Cain and Abel [31] is a security auditing and password recovery and cracking tool that is targeted only at the Windows OS which supports obtaining system hashes and cracking them. It can also work with user-provided lists of hashes. Cain and Abel supports dictionary, brute-force and cryptanalytic attacks, as well as rainbow tables attacks. It supports all of the commonly used hash types, as well as more obscure hashes. [32]

Cain and Abel is a proprietary closed source product, which is only available on Windows. Also, Cain and Abel does not have command-line control capabilities, does not include support for GPU-based password cracking, nor does it support distribution in any way. These disadvantages mean that Cain and Abel is only useful in specialised cases.

RainbowCrack

RainbowCrack [33] is a password hash cracking tool that uses the rainbow table attack (see Section 2.1.2). It requires user-supplied rainbow tables and does not discriminate on the hash type that is used for the generation of the rainbow table being used. The program supports multiple operating systems.

RainbowCrack has a version, which uses GPGPU; however, this version only supports Nvidia's® CUDA® and has no support for other graphics cards. Also, RainbowCrack does not support distribution and is proprietary closed source software.

3.3.2 GPGPU-based cracking tools

As GPGPU is a relatively new concept (see Section 2.1.5), the available cracking tools are still not as mature as tools using CPU-based computations are. However, some of these tools are already of production quality and have quite extensive feature support. Some of the most popular tools include:

- oclHashcat-plus
- IGHASHGPU
- Extreme GPU Bruteforcer

Each of these is discussed briefly and the advantages and disadvantages of each are outlined.

oclHashcat-plus

oclHashcat-plus [34] is considered the fastest and most advanced GPGPU-based password cracker available. It supports all of the widely used hashes as well as more obscure algorithms. The tool works with both Nvidia's and AMD's graphics cards and is available for multiple operating systems. It supports brute-force and dictionary attacks, and several custom attack modes.

oclHashcat-plus, however, does not support distribution. It is a proprietary closed source software.

IGHASHGPU

IGHASHGPU [35] is considered the first tool to make use of GPU computational power for the purposes of password cracking. It works on both Nvidia and AMD graphics cards.

IGHASHGPU, however, has limited support for hashes – MD4, MD5 and SHA-1. Also, it is only available for Windows, it is closed source proprietary software and commercial use is not allowed. Apart from that, at the time of writing, there has not been any development on the tool for several years.

Extreme GPU Bruteforcer (EGB)

Extreme GPU Bruteforcer [36] is a password cracking tool, which uses GPGPU computation. It supports all of the most used hash types and is optimised to work with salts and combinations of hash types. It supports dictionary, brute force attacks as well as other custom attacks.

This tool, however, is closed source proprietary software, it only works with the Windows OS and has restrictive licensing terms.

3.3.3 Combined CPU/GPGPU Tools

The set of tools that support both CPU and GPU computation include only one product - ElcomSoft's Distributed Password Recovery tool.

ElcomSoft's Distributed Password Recovery [37] is the only one that supports CPU and GPGPU computation as well as distribution. It has built-in support for multiple hashing algorithms and is of production quality. Also, it can work with different hardware platforms.

However, the licencing fees for this product are a major restrictive factor. Other disadvantages include the fact that it is closed source proprietary software and the fact that it only runs on the Windows OS.

3.3.4 Most Suitable Tools

This project aims to support multiple hardware platforms and different tools for the purposes of password cracking. It should be possible to use all of the described existing tools within the platform. However, since different tools have different capabilities, some tools may may be preferred over others. We focus on selecting one CPU-based tool, and one GPU-based tool which are to be used with the proof of concept implementation.

For CPU-based password cracking, the recommendation is to use John the Ripper. The table shows this is the most applicable tool for password cracking on CPU. Besides this KPMG already has setup an infrastructure using John, it should be possible to incorporate that infrastructure within the current project. Moreover, John is actively developed and additional features added may become more relevant in the future.

Summary Tools	John	C&A	RC	ocl	IGHASHGPU	EGB	ElcomSoft
CPU support	+	+	+	−	−	−	+
GPU support	?	−	?	+	+	+	+
Hash-type Availability	+	+	+	+	−	+	+
Brute-Force attack	+	+	−	+	+	+	+
Dictionary attack	+	+	−	+	−	+	+
Rainbow-table attack	−	+	+	−	−	−	+
Custom attack	+	+	+	+	−	+	+
Can run on Windows	+	+	+	+	+	+	+
Can run on Linux	+	−	+	+	−	−	−
Open source project	+	−	−	−	−	−	−
Can use Distribution	+	−	−	−	−	−	+
Free of usage	+	+	+	+	+	+	−
Command Line tool	+	−	+	+	+	+	−

Table 2: A comparison between existing cracking tools and their capabilities. (Meanings:+ is yes, - is no, ? is partial)

For GPU-based password cracking, the recommendation is to use oclHashcat-plus. Although it is closed source software, this tool is the most advanced password cracker with GPGPU capabilities. Also, it is actively developed and supported.

3.4 Summary

In this chapter we examined different architectural and communication paradigms, and provided an overview of existing cracking tools. We discussed the characteristics of the examined architectures and cross-compare them with regards to the required functionality of the system. We took into consideration the applicability of each architecture and how well it covers to project requirements. We came to the conclusion that a centralised architecture is most suitable for the distributed password cracking platform.

Afterwards, we examined the advantages and disadvantages of different communication models. We also took into consideration the expected amount of data that will be transferred for the system to operate correctly. We came to the conclusion that message-oriented communication is most suitable for the distributed password cracking platform. Only in the case of submitting a cracking job to a client node will RPC be used.

As for the existing cracking tools, we examined the capabilities of several popular tools. When considering the project requirements, we came to the conclusion that John the Ripper and oclHashcat-plus are the most suitable tools for this project. In the next chapter, we describe in detail the functional requirements and top-level design of the distributed password cracking platform.

4 Functional Requirements Specification

For the system description and project requirements, refer to Section 1.2.

In the previous chapter we made conclusions about the type of architecture and communication model that are to be used within the distributed password cracking platform. Continuing with our research questions, we examine the functional requirements of the system in this chapter. We describe the full functional requirements and design specifications for the distributed password cracking platform.

First, the system usage patterns are examined and described with use cases. Afterwards, in Section 4.2, the top-level functional requirements for the project are outlined. In Section 4.3 these requirements are further detailed.

4.1 System Usage Patterns

This section will focus on the identified intended usage patterns for the system. The need for the usage patterns comes from the fact that for the technical design the system needs to know what kind of actions a user can perform and so what functions the system should be capable of doing. A user is first introduced to the web-interface, where the user submits its credentials to login. When the user has logged in, five tasks can be chosen from:

1. New Job
2. Get Status
3. Stop Job
4. Show History
5. Delete Job

These use cases are shown in Figure 4 and are described in detail in the following sections.

4.1.1 New Job

The first possible task is to create a *new job* at the controller (see Figure 4a). The user may enter several parameters. The possible parameters are:

- Type of hash being used
- Cracking strategy to use
- Whether salting has been used
- Which key space to use

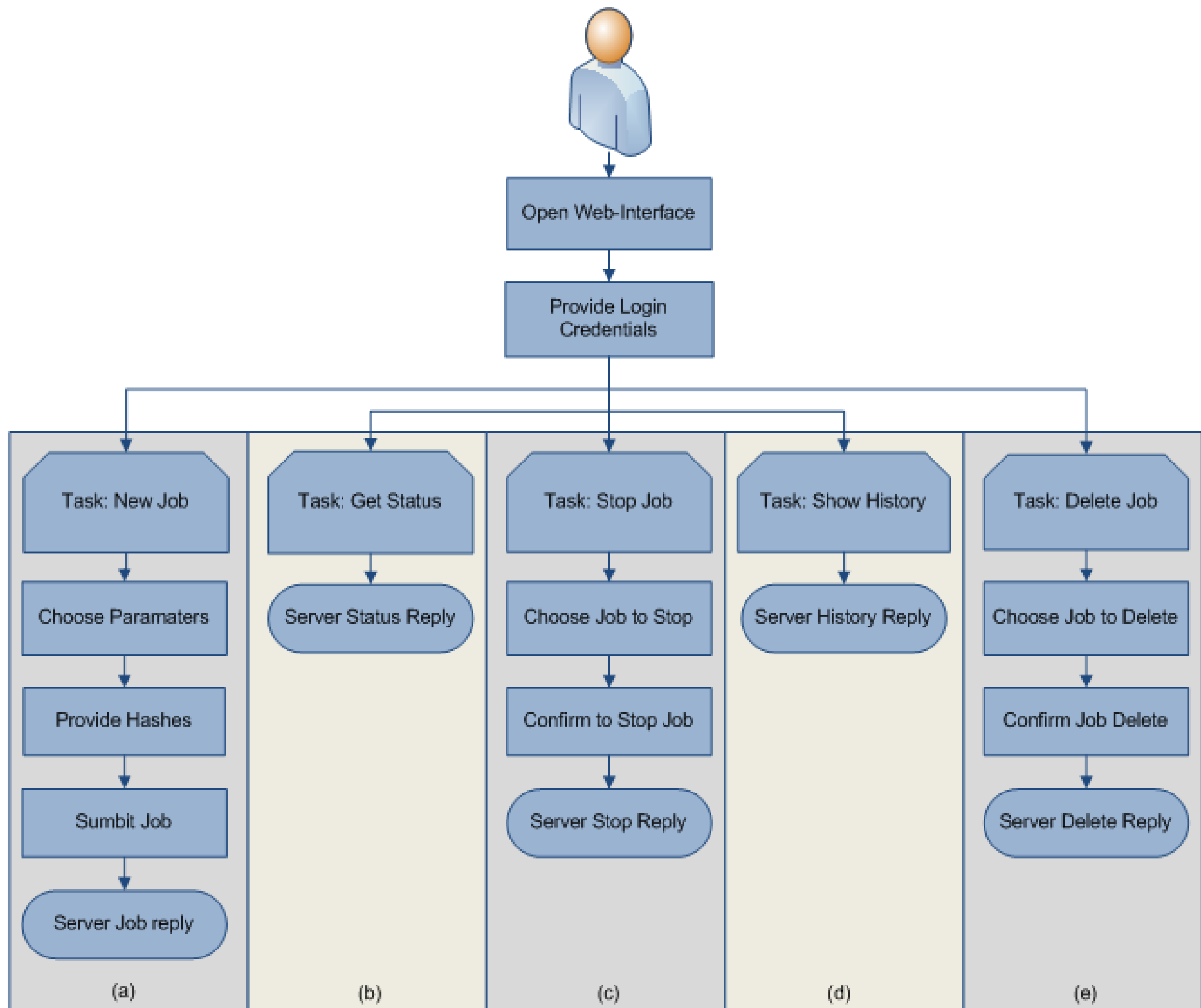


Figure 4: User Use-Cases.

- What the maximum running time of the job is

After setting the correct parameters, the list of hashed values needs to be provided. After the list has been input the job can be submitted. The server would then respond with a reply, containing the job ID along with additional information.

4.1.2 Get Status

The second task, *Get Status*, allows the user to view the current status of the system (see Figure 4b). The system would then return information including:

- A list with all queued jobs
- An estimated completion time for the current job and other running jobs
- The number of cracked hashes for all jobs
- Parameters given when the jobs were submitted (job ID, alphabet, etc.)

4.1.3 Stop Job

The *Stop Job* task stops a current running or queued job (see Figure 4c). The user can choose the job he wants to stop. After requesting confirmation, the platform stops the execution of the selected jobs.

4.1.4 Show History

Another task a user can use is *Show History* (see Figure 4d). This task allows the user to get a list of all previous jobs that have already been executed. This list contains information related to the completed jobs.

4.1.5 Delete Job

The last task the user can perform is *Delete Job* (see Figure 4e). With this task a user can choose a job he wants to remove completely. After requesting confirmation, the platform proceeds with the deletion. If the job is running, *Stop Job* is executed first to stop the execution. Afterwards, all content regarding the job is deleted.

4.2 Functional Requirements

Based on the project requirements (see Section 1.2), the project team identified the following general functional requirements. These apply to the whole new cracking platform being developed within the context of the current cracking infrastructure at KPMG. It should be capable of doing the following:

- Adaptable with regards to future technological changes within the KPMG infrastructure
- Capable of working with various existing cracking tools
- Be possible to extend the platform's capabilities with regards to algorithms and tools supported

- Be possible to be extend the platform by adding new worker nodes
- Be capable of following a predetermined strategy and, based on it, making decisions autonomously
- Support multiple users simultaneously
- Require little input or effort from the user's point of view to operate correctly

Based on the outlined requirements, a high-level system overview was conceived (see Figure 5).

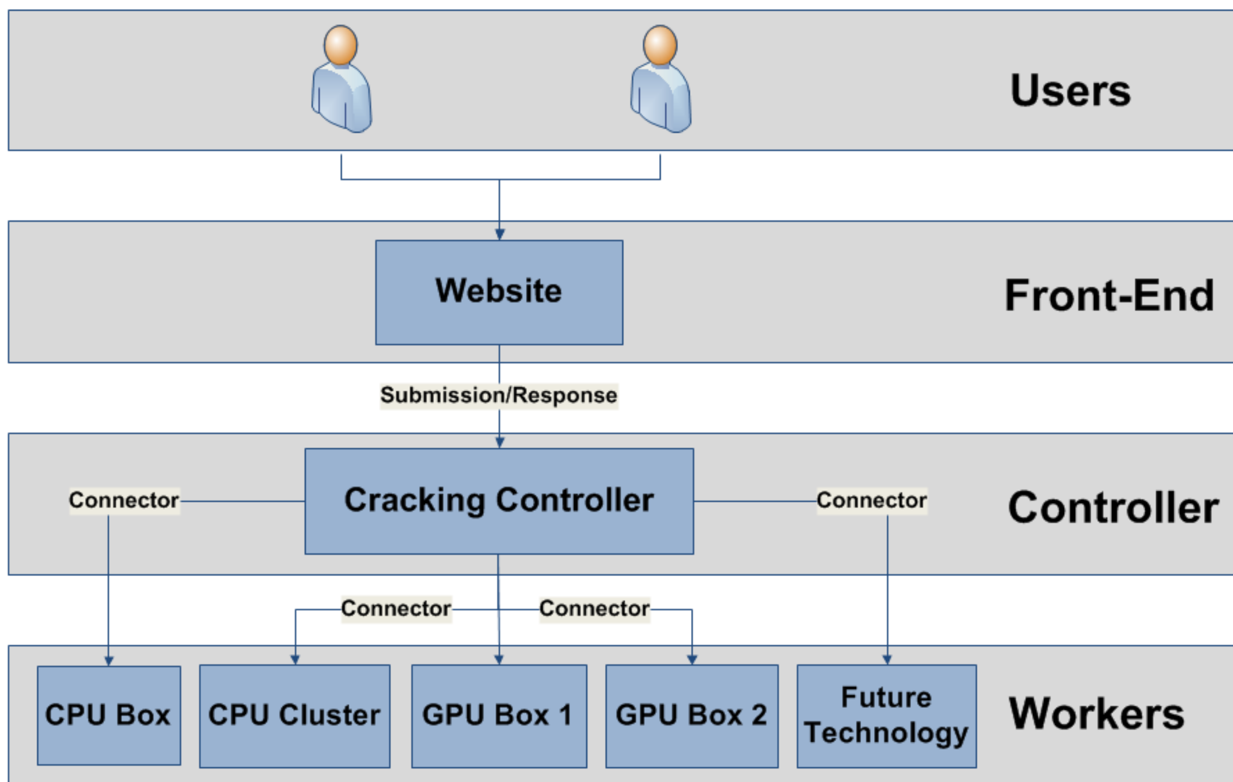


Figure 5: A high-level system overview.

The system will consist of the following major logical components:

- Website communication interface
  – the communication channel used between the website and the Controller node – responsible for passing the user interface parameters, job requests, status requests and stop requests to the platform

- Controller node
  – the core of the platform – responsible for handling user requests, controlling the worker nodes, providing status and notifying about the results of a submitted job
- Connectors
  – communication components – responsible for providing unified and reliable communication between the controller and the worker nodes
- Worker nodes
  – the computers performing the actual computations – responsible for handling parts of a job (subjobs), utilising the cracking tools and notifying the controller about the results of a subjob

4.3 Detailed Requirements

4.3.1 User Interface

While the developed platform itself will not have a Graphical User Interface (GUI), the actual GUI's capabilities will be determined by the platform's capabilities. Therefore, the functional requirements for that interface need to be taken into consideration.

Based on the functional requirements and on the current capabilities of the KPMG cracking platform, the following user interface functional requirements were outlined. A user is capable of doing the following:

- User input
  - Input (lists of) hashes for cracking
  - Specify the type of hashes provided
  - Specify additional data, related to the hash type and salting type
  - Specify the alphabet being used
  - Specify minimum and maximum key sizes
- Job status
  - Check the status of a job
  - See intermediate results
  - See errors when they occur
  - The system notifies the user when a hash has been successfully cracked
  - The system notifies the user when a job has finished and provide the overall job results
- Check their history
- Stop jobs
- Delete jobs

4.3.2 Controller Functionality

The Controller is the central part of the whole platform. It is the main coordination point, which should handle user requests, jobs dispatching and node management. The following functional requirements have been defined for the Controller component:

- User requests handling
  - Can accept job requests
  - Supports status requests
  - Provides historical data
  - Supports stop requests
  - Supports delete requests
  - Supports intermediate results requests
  - Supports all of the capabilities of the GUI
- User input handling
  - Performs input validation, taking into consideration the specified hash type
  - Stores the input for the duration of the cracking job
  - Supports a defined interface for a “pluggable” strategic module
  - Makes use of a strategic module for determining the appropriate cracking strategy and resource utilisation pattern
- Worker nodes control
  - Registers new nodes automatically
  - Keeps a list of active worker nodes
  - Can dispatch subjobs to worker nodes
  - Can cancel active subjobs at worker nodes
  - Can request the status of a worker node
  - Can receive status updates from nodes
  - Can receive the results of a subjob from a worker node
- User notifications
  - Notifies the user when a successful crack has occurred
  - Notifies the user when a job has been finished
  - Provides the results of finished jobs
- Resilience and Error Handling

  - Is capable of operating in a suboptimal environment
  - Is capable of recording errors and failures
  - Provides the Worker Nodes with the capability for submitting errors
  - Is capable of providing error information to the user
  - Is capable of handling failing Worker Nodes

4.3.3 Worker Node Functionality

The worker nodes are the systems that perform the actual cracking of hashes. These systems are controlled by the Controller and should register with it at startup. These systems should accept jobs from the Controller and then proceed with the execution. The following functional requirements have been outlined:

- Should register with the predetermined controller after startup
- Status request handling
  - Accepts status requests
  - Returns current system load
  - Returns current jobs
  - Returns cracking capabilities - CPU/GPU cracking, hashes supported, etc.
- Job processing
  - Accepts jobs from the Controller
  - Accepts job parameters from the Controller
  - Invokes the cracking tool specified and pass the parameters in an appropriate way
  - Provides periodic progress reports to the Controller when a job is active
  - Notifies the Controller when a hash has been cracked
  - Notifies the Controller when a job finishes
  - Provides the job results
  - Notifies the Controller about any and all errors occurring during the processing of a job
  - Allows for the cancellation of a job
- Cracking tool support
  - Interfaces an external cracking tool
  - Translates the platform parameters for passing to the cracking tool
  - Processes the tool's intermediate output for the purposes of progress reporting
- Resilience and Error Handling
  - Is capable of operating in a suboptimal environment
  - Is capable of submitting detected errors to the Controller

4.3.4 Platform Communication

As the platform consists of several distinct components, there must be sufficient communication capabilities integrated for the functionality to be realised. The following functional requirements have been identified:

- The website can submit requests to the Controller and can process its responses
- The Controller can receive requests from the website and the worker nodes and can respond to these requests
- The Nodes can receive requests from the Controller and respond to these requests
- The Nodes can provide status updates and job results to the Controller

4.4 Summary

This chapter examined the functional requirements of the platform. These were based on the anticipated use cases, as defined by KPMG. Based on the use cases and on the project requirements, we were able to detail all of the functional requirements of the platform.

After having identified these requirements, it was possible to proceed with the creation of the Technical Specification for the password cracking platform.

5 Technical Design Specification

After having a complete functional specification, it was possible to create a concrete technical specification for the project. This chapter starts by providing an overview of the system architecture. It then proceeds with providing workflow diagrams for each logical component in the architecture. These diagrams show the states different components can be in and also the actions that are taken when certain conditions are observed. The chapter then provides a definition of the communication protocol and concludes with the and overview of the platform operation.

5.1 System Architecture Overview

Based on the functional requirements defined in Sections 4.2 and 4.3, a system design was created (shown in Figure 6). The design shows the main functional components within the logical components of the system – the website communication interface, the Controller, the Worker node and the communication components.

5.1.1 Architecture: Website Communication Interface

The website communication interface is used to pass data and user requests from the website to the platform Controller. It is tasked with providing a means for the website to submit user input, specified parameters and additional data to the Controller. Also, the website should be capable of submitting status and stop requests to the Controller. The Controller should respond accordingly.

The development of the communication interface on the Controller side is a top priority. However, this project will not provide functional or technical specifications for the website components, as that is outside the scope of the project.

5.1.2 Architecture: Node Controller

The Node Controller is the central component in the distributed password cracking platform. Its tasks consist of handling user requests, deciding of the worker nodes. It also handles and processes user requests and provides status and job outcome information to the website. It is divided in two distinct logical components – the Communicator and the Job Dispatcher.

Communicator Overview The Communicator is the communication component within the Node Controller. Its task is to listen to requests to the Controller. These requests originate from either the Website or from any of the Worker nodes. It should provide a consistent interface for communication with the nodes, which is capable of passing all of the identified parameters in a uniform manner. Also, it should handle, validate and store the input of data – either from the Website or from any of the Worker nodes. When data input indicates that a job has been finished, the Communicator should notify the user associated with that job.

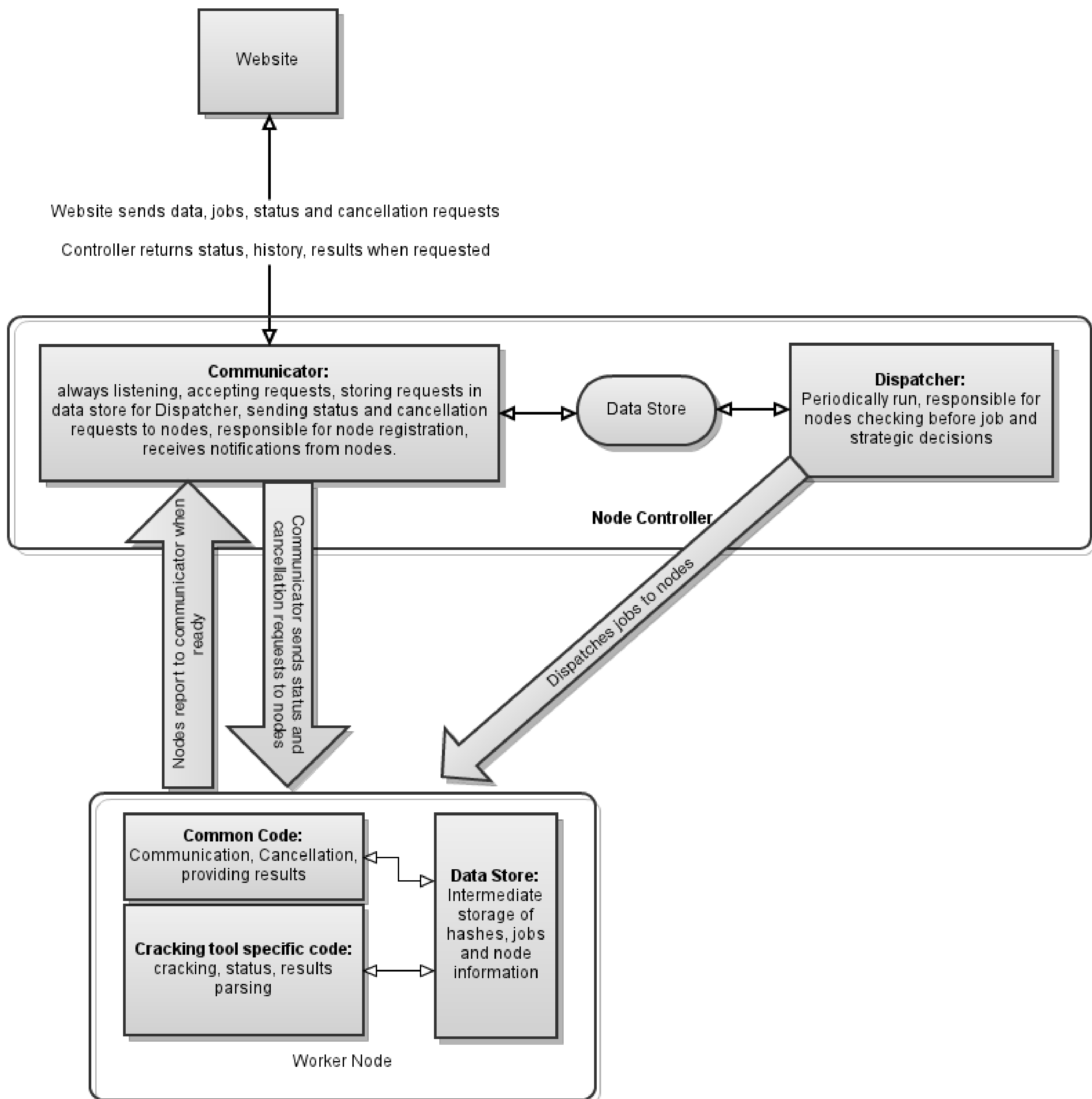


Figure 6: Functional system design.

Generally, the communicator does not contact the nodes directly, as that is handled by the Job Dispatcher (see below). In a few cases, however, the Communicator is responsible for contacting the nodes directly:

- when a registration request has been submitted by a node
- when a stop request has been submitted via the website
- when a node reports that a job has been finished and it is determined that other nodes should be stopped

Dispatcher Overview The Job Dispatcher is the component within the Node Controller that is tasked with the analysis of input data, the producing of a strategic decision and the dispatch of the subjobs to the Worker Nodes.

The Job Dispatcher runs periodically on the Controller system. When it runs, it checks whether there are any jobs queued for processing. If this is the case, the dispatcher starts by first checking whether all of the registered nodes are still active and available. After gathering that information, the dispatcher passes it along with the job parameters to the strategic module. The strategic module analyses the input and produces a list of nodes and node parameters for the subjobs to be dispatched to. Finally, the dispatcher contacts the nodes, submits the subjob requests and exits.

Data Store Overview The Data Store component provides persistent storage for any data that needs to be stored.

5.1.3 Architecture: Worker Node

The Worker Node is a component within the platform that performs the actual cracking of hashes. The Worker Node is dependent on the Controller, which submits job, status and cancellation requests to it. The node hosts an external cracking tool, which operates in a non-distributed manner (locally on the node) and passes the requests it receives from the Controller to the cracking tool.

The Worker Node's functionality can be divided in two parts – common part and tool-specific part.

Common Code Overview The common part of the Worker Node's handles the communication and job management functionality. It processes the different requests from the Controller and reacts accordingly. In the cases when the Controller submits a new job, the job input and parameters are inspected and the job is passed to the cracking tool. In the cases when the Controller requests the node's current status, the node's capabilities and load are returned. In the cases when the Controller requests a job cancellation, the responsible tool process id is retrieved and the tool process is terminated.

The common code part is also responsible for notifying the Controller about any errors that the Worker encountered during its operation.

Tool-specific Code Overview The tool-specific part of the Worker Node is tasked with interfacing the external cracking tool. This part of the Worker Node's code is specific to the tool used, as

well as to the tool version, the operating system and to the execution environment. This component passes the job parameters to the tool, parses the tool's intermediate results and identifies errors, failures and successful cracks.

5.1.4 Architecture: Communication Components

The communication components are not clearly identifiable as distinct logical components. Instead, they comprise the rules and conventions that the platform components should follow while communicating with each other.

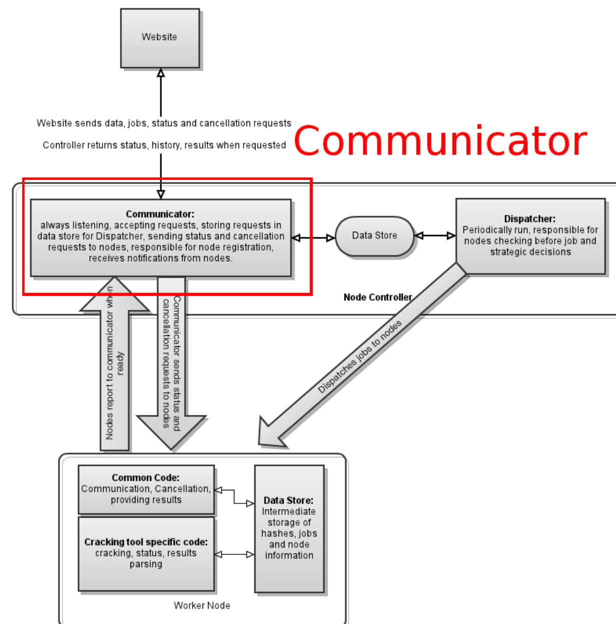
This section defined the top-level architectural components of the platform. The next sections provide details about the internal organisation and workflow of each component.

5.2 Controller Components – Models and Workflows

The Controller is comprised of three distinct components – the Communicator, the Job Dispatcher and the Data Store. Separate workflow diagrams have been produced for the first two. The model for the Data Store is described afterwards.

5.2.1 Controller: Communicator Workflow

The Communicator serves the purpose of handling all incoming requests to the Controller. These requests may be originating either from the front end, or from the worker nodes. The Communicator is shown below in the context of the top-level architecture (see Figure 6).



The Communicator component handles the communication capabilities of the Node Controller – it processes website and node requests and submits some of the control actions to the nodes. The Communicator passes through several states and its workflow is shown in Figure 7. Its states and actions are described below.

Startup State When the Communicator starts up it is in the *Startup* state. There is no operating knowledge about existing nodes at this point. The Communicator checks whether there is a record of any nodes being active in the Data Store. If there is such a record, the Communicator contacts each of those, requesting a status update. If the recorded nodes are still active, they should respond with their status. Once the controller has determined which nodes are active, it can transition into the Listening state.

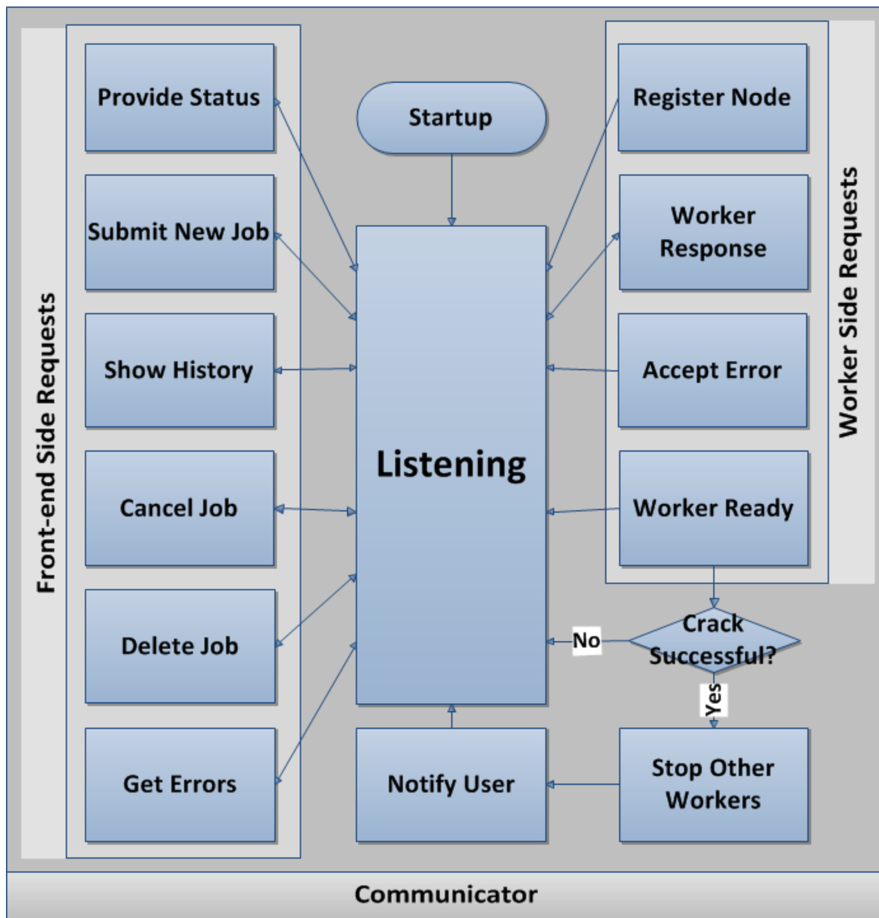


Figure 7: Communicator workflow diagram.

Listening State In the *Listening* state the Communicator expects incoming requests – either from the front end (user requests), or from the Worker Nodes. This state accepts the incoming requests and, based on the type of request, transitions the Communicator into one of its other states.

Provide Status State The *Provide Status* state is entered when the Communicator has received a user request for the status of a job. The controller contacts the Data Store, extracts and manipulates the needed information and returns it to the front end.

Submit New Job State The *Submit New Job* state is entered when the Communicator has received a request that submits a new job. The Communicator processes the input data and transforms it into an internal format if necessary. The job ID is generated. This module then makes a request to the Data Store and stores the data into persistent storage. Afterwards, the system transitions back into the Listening state.

Show History State The *Show History* state is entered when the Communicator has received a show history request. The Communicator gets all jobs history from the user out of the Data Store and shows this back to the user. Afterwards, the system transitions back into the Listening state.

Cancel Job State The *Send Cancellation* state is entered when the Communicator has received a user request for the cancellation of a job. The Communicator sends a cancellation message to each node working on the specified job, removes all related data from the Data Store and returns to the Listening state.

Delete Job State The *Delete Job* state is entered when the Communicator has received a delete job request for deletion of a job. The Communicator removes all information regarding that job out of its Data Store and goes back into the Listening state.

Get Errors State The *Get Errors* state is entered when the Communicator has received a request for the list of errors. The Communicator extracts all recorded error data from its Data Store, returns it to the website, and goes back into the Listening state.

Register Node State The *Register Node* state is entered when a node has been started has sent a request for registration to the Communicator. The Communicator generates a node ID and returns it to the node. The node information provided with the request is stored in the active nodes list. Afterwards, the system transitions back into the Listening state.

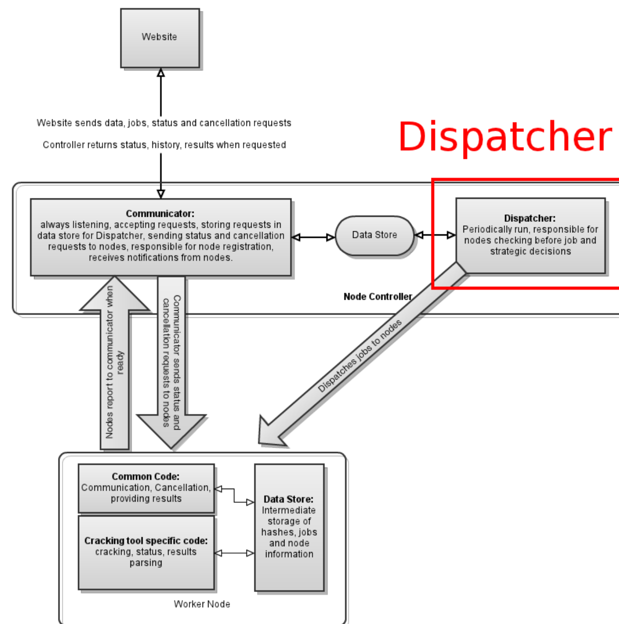
Worker Response State The *Worker Response* state is entered when the Communicator has received a progress update from a Worker Node. The Communicator stores the status data in the Data Store and returns to the Listening state.

Accept Error State The *Accept Error* state is entered when the Communicator has received an error notification from a Worker Node. The Communicator stores the error message and the related job data in the Data Store, and returns to the Listening state.

Worker Ready State The *Worker Ready* state is entered when a node reports that it has finished with a particular job. The data passed is being processed. If it has been determined that the completion of this particular job is enough for the whole job, other working nodes are notified that they should stop working (the *Stop Other Workers* state). If the finishing of this job completes a specific cracking task, the user responsible is being notified about the results (the *Notify User* state). The Communicator then transitions to the Listening state.

5.2.2 Controller: Dispatcher Workflow

The Dispatcher is responsible for analysing the cracking tasks and splitting them into several subjobs, which it then dispatches to the available Worker Nodes. The Dispatcher is shown below in the context of the top-level architecture (see Figure 6).



The Job Dispatcher is responsible for the allocation and proper utilisation of the computational resources available within the system's current state. It is run periodically, checking for pending jobs and determining the optimal distribution of the requested tasks (making use of the strategic module). It has several states (shown in Figure 8), which are explained below.

Startup State When the Job Dispatcher is started it enters the *Startup state*. It transitions immediately to the Contact Data Store state.

Contact Data Store State In the *Contact Data Store* state, the Job Dispatcher queries the Data Store, checking whether there are any pending jobs. If there are such jobs the dispatcher transitions to the Node Checker state. Otherwise, it transitions to the Exit state.

Node Checker State In the *Node Checker* state, the Job Dispatcher issues status requests to all of the known nodes. Upon the reception of all replies the Job Dispatcher updates the list of active nodes if necessary. Afterwards, it chooses a job to perform and passes the job parameters to the Determine Strategy state.

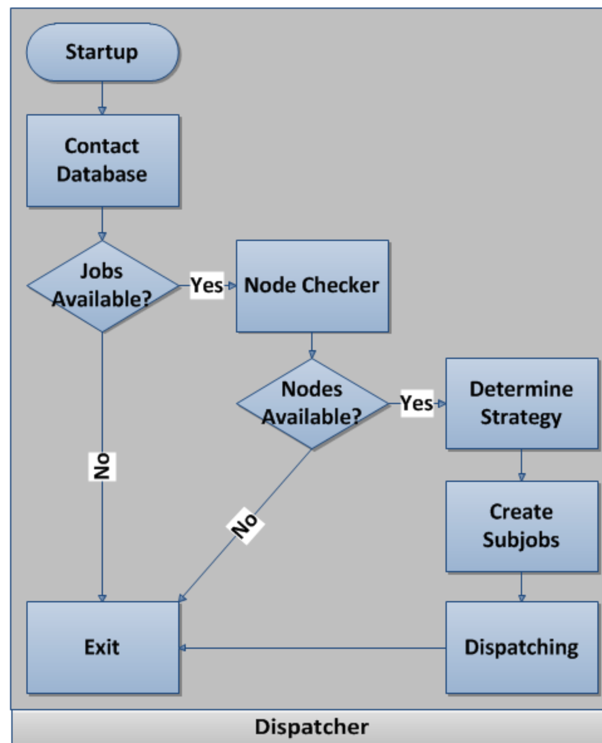


Figure 8: Job dispatcher workflow diagram.

Determine Strategy State It is the *Strategy Module*'s responsibility to examine the state of the platform, the selected job and the job parameters and to define the optimal distribution of this particular job over the Worker nodes.

The Strategy Module receives as input the parameters that the user has specified for this particular task. These parameters include, among others, the hashtype, the alphabet used and the keylength. Based on these parameters, the strategy module makes a decision about the way that particular task will be distributed – whether CPU or GPU will be used, whether the hashes will be distributed, or keyspace distribution will be used, etc.

After examining the input parameters, the Strategy Module splits the task into so called "subjobs", which contain the needed information for a Worker Node to start cracking. The subjobs are then handled and stored in the *Create Subjobs* state.

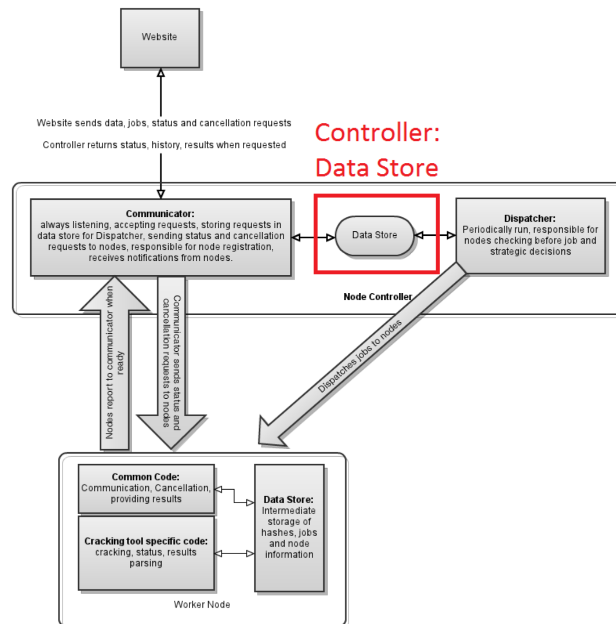
Create Subjobs State Based on the information received from the Strategy Module, the *Create Subjobs* state creates the different subjobs and stores them in the Data Store. Afterwards, the subjobs are dispatched to the Worker Nodes in the *Dispatcher State*.

Dispatching State In the *Dispatching* state, the system examines the created subjobs and constructs the corresponding request messages. These request messages are then sent to the corresponding Worker Nodes, assigning them jobs to work on. Afterwards, the Job Dispatcher transitions to the Exit state.

Exit State In the *Exit* state the Job Dispatcher stops its operation.

5.2.3 Controller: Data Store Model

The *Controller Data Store* contains all data which is needed for the controller to operate properly. In Figure: 9 the layout of the data store is shown. A explanation why the different tables are necessary is explained below:



Users Table The users table contains the usernames and passwords needed for the user to authenticate themselves. These are passed by the front-end, when users submit hashes and request statuses and are checked at the Controller against the values in this table.

Nodes Table This table contains information about the registered worker nodes. This information is used by the Job Dispatcher to determine what resources it has available. The table stores the nodes' IP addresses, names, whether the node uses CPU or GPU computational power, and which tool is being used. As CPUs and GPUs vary in their capabilities, it is possible to store more information detailed information in this table. However, the time constraints of this project did not allow us to extend this table more.

Jobs Table In the jobs table, the user-submitted tasks are stored. This table contains the parameters needed for creating subjobs.

Subjobs Table The subjobs table contains all created subjobs with the different cracking parameters, as well as the completion status.

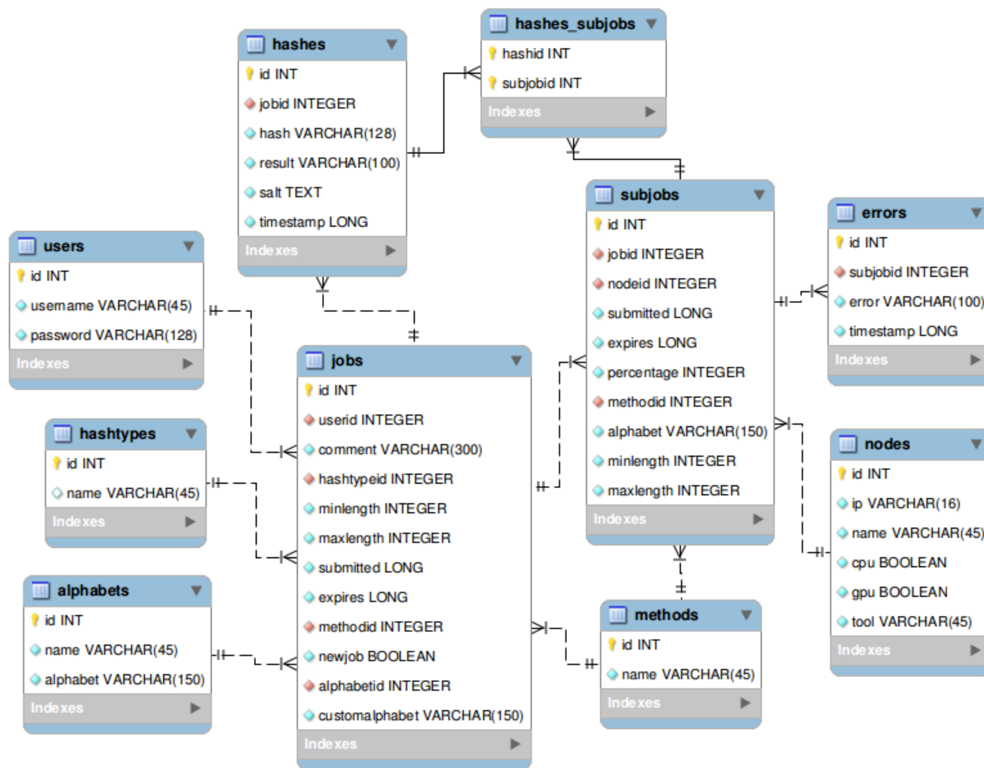


Figure 9: Controller Data Store.

Hashes Table All submitted hashes are stored in the hashes table. The hashes are related to entries in the jobs table. Apart from the hashes, the used salts and the plaintext results are stored in this table.

Hashes–subjobs Table This is a simple table which links the hashes with the subjobs. This way subjobs can have different hashes.

Hashtypes Table The hashtypes table contains a list of hashtypes, which need to be given as parameter when submitting a job.

Alphabets Table In the alphabet table the different possible alphabets or characters sets are being stored. The user can select one using a parameter when submitting a job.

Methods Table This table contains the possible cracking methods. Entries may include brute force attacks, dictionary attacks, rainbow table attacks and others. It is linked to both the jobs and subjobs, as those may only make use of the methods, supported by the platform.

Errors Table As the Workers may encounter errors during their operation, these errors must be reported to the Controller. Such information is stored in the errors table.

5.3 Worker Node – Workflow & States

The Worker Node is comprised of a single logical component. However, because of the modular design of the platform, we can identify two distinct code bases within the Worker – the Common Code, handling communication and control, and the Tool-Specific Code, interfacing the external cracking tool. The workflow for the Worker Node component is shown in Figure 10.

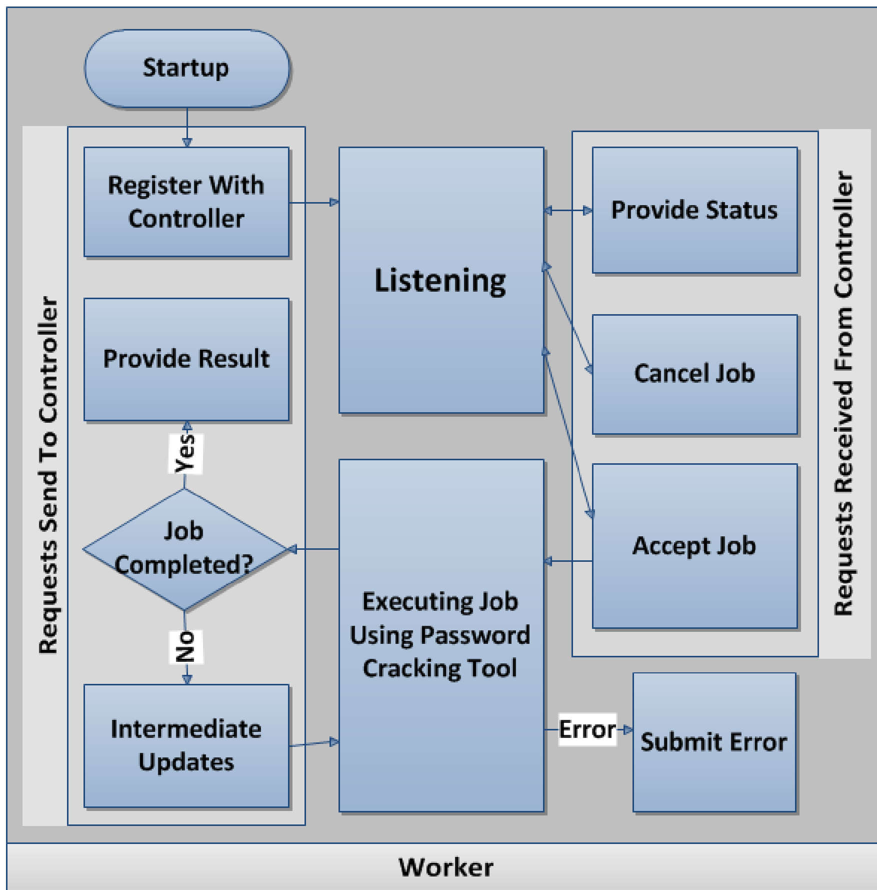
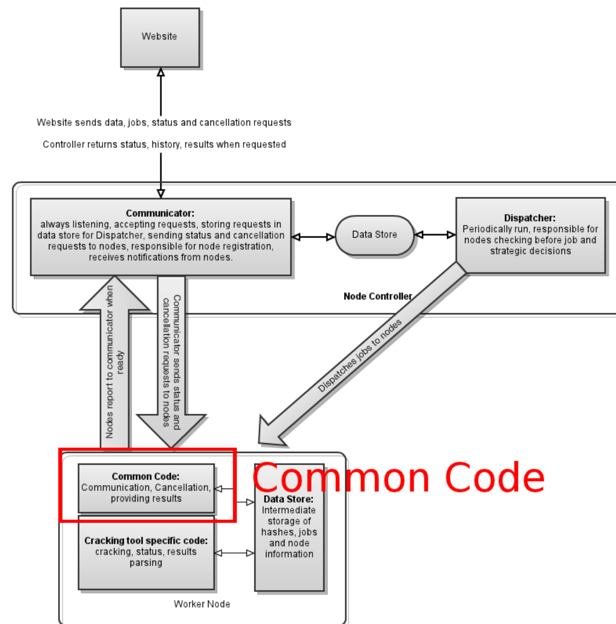


Figure 10: Worker node workflow diagram.

The Worker Node is the component that defines the computational capabilities of the platform as a whole. Multiple Worker Nodes are to be connected within the system. Each of the Worker Nodes is to be controlled by the Controller and tasked with the execution of particular cracking subjobs. As shown in Figure 10, the Worker Node has several execution states, which are described below.

5.3.1 Worker: Common Code

Below, the Common Code part of the Worker is shown in the context of the top-level architecture (see Figure 6).



The worker module can register a worker at the controller. When it is available and listening a subjob can be submitted from the controller. It will startup the specific cracking tool the worker is using. The worker sends intermediate status updates and will publish progress. Errors are also being sent back to the controller. The controller may cancel the worker's active task at any time. Explanations about the purpose and function of each of the Worker's states follow.

Startup State When the node starts up it enters the *Startup* state. At this state it should obtain information about the address of the controller. It then transitions to the Registration state.

Registration State In the *Registration* state the node sends a registration request to the Controller. The reply should contain the node ID. After receiving and storing the node ID, the Worker Node transitions to the Listening state.

Listener State When in the *Listener* state, the node expects requests from the Controller.

Provide Status State The node transitions to the *Provide Status* state when it has received a status request message from the Controller. It replies with its status information and returns to the Listening state.

Cancel Job State The node transitions to the *Cancel Job* state when it has received a cancellation request from the Controller. In that occasion the node identifies the process corresponding to the cracking tool, which is busy with the execution of the specified job and kills that process. The node transitions to the Listening state afterwards.

Accept Job State The node transitions to the *Accept Job* state when it has received a new job request from the Controller. It processes the passed parameters and converts it to an internal representation if needed. It then transitions to the Process Job with Tool state.

Executing Job State In the *Executing Job* state the Worker Node invokes the external cracking tool, passing it the appropriate parameters, and starts cracking.

Intermediate Updates State While in the Executing Job state, the Worker Node transitions periodically to the *Intermediate Updates* state. In this state the node submits the progress with the current assigned task to the Controller and returns to the *Executing Job State*.

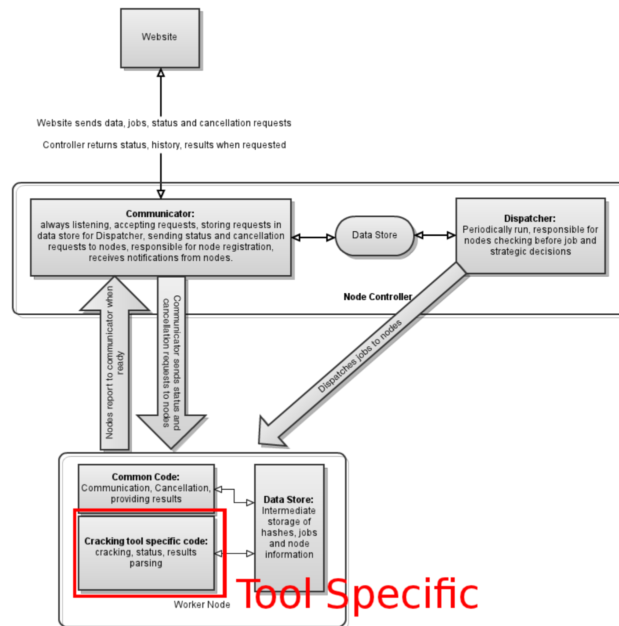
Submit Error State The node transitions to the *Submit Error* state when it has encountered an error while executing the cracking task with the external tool. The Node submits the error message that it obtained (either from predefined values, or directly from the cracking tool) to the Controller, and returns to the Listening state.

Provide Result State When the *Executing Job State* is finished the worker sends all it results back to the controller and start cleaning its own local Data Store and files. When this is done it can accept net job whit the *Accept Job State*

Notify Controller State When the external cracking tool exits – with an error, failure or success, the Worker Node transitions to the *Notify Controller* state. In this state the node parses the output information from the external cracking tool and reports the results to the Controller. Afterwards, the node transitions to the Listening state.

5.3.2 Worker: Tool-specific Code

As the platform is designed to make use of existing external password cracking tools, these tools need to be interfaced in some way. This is done by creating a module with tool-specific code. Below, the Tool-specific part of the Worker's code base is shown in the context of the top-level architecture (see Figure 6).



The tool-specific module handles the user-supplied parameters and converts them into a form, suitable for the tool in question. It also extracts the hashes to be cracked from the local data store of the Worker and converts them in a suitable format as well. The tool-specific code must then invoke the external cracking tool with the converted parameters and monitor its execution.

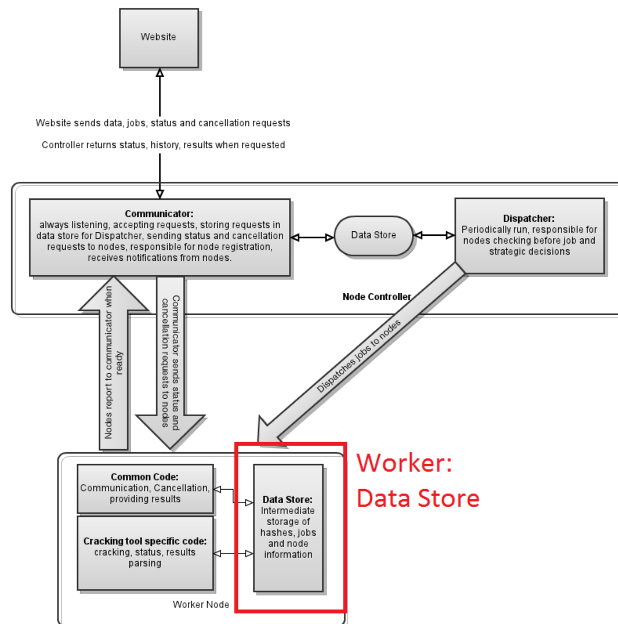
The module must have a way of recognising when the tool succeeds in cracking a hash. It must then notify the Controller about the cracked hash.

When the tool finishes its execution, the tool-specific module must notify the Controller that the assigned subjob has finished execution.

The code, handling the external tool is entirely dependent on the tool used, as it must take into consideration specific execution parameters and the hashes format. Therefore, more details cannot be provided.

5.3.3 Worker: Data Store Model

The *Worker Data Store* is needed only for temporary storage of the subjob and hashes, concerning the current task it is working on. The worker data store's model is shown in Figure 11. The tables below explain why they are needed and globally contain:



Jobs Table The jobs table contains the data needed for the current jobs it is working on. When it is not working is has no jobs. The parameters are basically the same as on the controller side. In addition it has a pid field which contains the pid number of the process running the current cracking tool.

Node Table In the node table the basic configuration data of the worker is being stored.

Hashes Table In the hashes tables all hashes are being stored. These hashes are coupled to the jobs table. Besides the hash also the salts and result of the crack hash are being stored.

Hashtypes Table The hashtypes table contains a list hashtypes which need to be given as parameter when submitting a job.

Alphabets Table In the alphabet table the different possible alphabets or characters sets are being stored. The user can select one using a parameter when submitting a job.

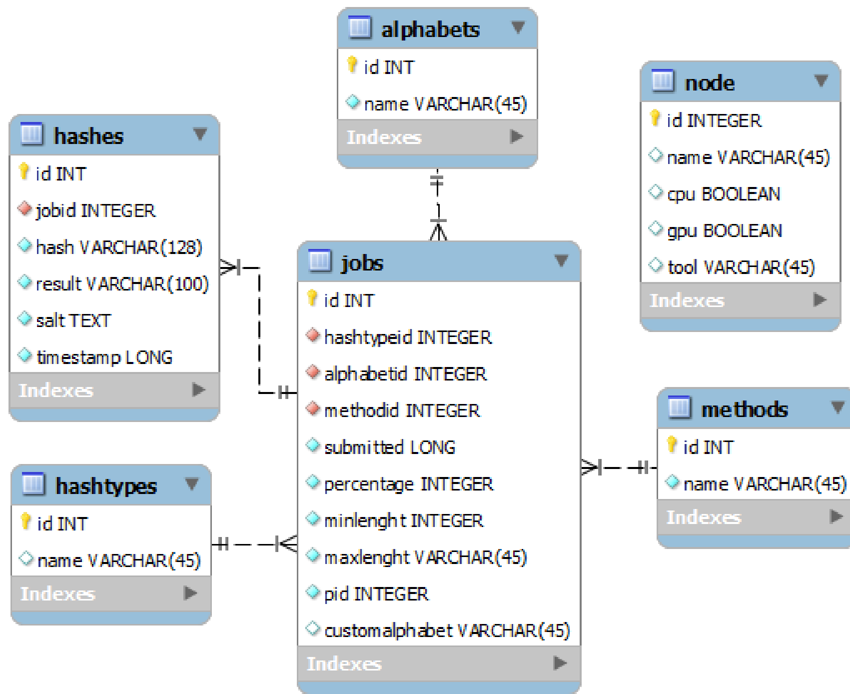


Figure 11: Worker Database.

Methods Table This table contains the strategy method that the users takes. It is linked to both the jobs and subjobs and this field decides which strategy is being used for creating the subjobs.

5.4 Platform Communication Protocol

This section describes the protocol used for communication within the cracking platform. It also describes the external interface of the platform, as visible to the website front end. The defined data structures, messages, remote procedures and related parameters are described.

5.4.1 Communication Model

As discussed in Section 3.2.5, the main communication channel within the distributed password cracking platform should be message-oriented and based on the request-response model. For the assignment of jobs to the worker nodes, an asynchronous RPC approach should be used.

It has been decided that the platform should take advantage of existing technologies for the implementation of the communication protocol. Therefore, instead of custom developments efforts, involving the implementation of communication via sockets, this project should use HTTP-based communication, supported by existing web server technology. Requests should be sent to the server by means of either HTTP POST, or HTTP GET. This decision is to be made by the implementing party.

Furthermore, as exchange of data is needed, there is a need for a mechanism that supports such exchange. For data that needs to be sent as part of a request, the integrated mechanism of HTTP POST/GET parameters can be used. POST/GET parameters are data units that can be transported with the corresponding POST/GET request. When data needs to be supplied as part of a reply, that data can be formatted in a number of ways – using XML, CSV, JSON or a custom format. These decisions are to be made by the implementing party.

In the case of job assignment, the Controller should perform a Remote Procedure Call (RPC) to the identified worker nodes, marshalling the parameters and data, and wrapping them within an HTTP POST method. The node should reply immediately, accepting the request, and start working on the assigned job. When the node finishes the required work, it is expected to notify the Controller about the outcome of the computation.

All requests and RPCs sent to any component within the system must be valid method calls, applicable to the component they are being sent to. Otherwise, the receiving component must answer with an negative reply.

In the cases, where there is no data to be passed for a certain required parameter(s) within the context of a request, the sending party is not allowed to omit the parameter. Instead, an empty string must be passed.

Requests should be made to URLs formatted in the following way:

```
http(s)://<SERVER-ADDRESS>/<REQUEST>
```

5.4.2 Protocol Methods

The Controller has two interface sides – facing the website front end and facing the worker nodes. Communication is required at both ends, with the Controller capable of receiving requests from both parties. The Worker Node, on the other hand, only faces the Controller and can only receive requests from it. Also, since RPC is used for the submission of cracking tasks to the Worker Nodes, the RPC methods are also included in the communication protocol.

An overview of the protocol methods, is given in Table 3. The table contains information about where the methods are implemented, what they are used for and which component uses them. For full method definitions, see Appendix E. For definitions of the data structures used with the protocol methods, see Appendix D.

Method	**Implemented**	**Used By**	**Purpose**
submitJob	Controller	Website	Submission of a cracking job by the user.
stopJob	Controller	Website	Stopping of a job by the user.
deleteJob	Controller	Website	Removal of a job by the user.
requestJobList	Controller	Website	Provision of a list of known jobs to the user.
requestJobStatus	Controller	Website	Provision of the status of a running job to the user.
requestSystemStatus	Controller	Website	Provision of the current *system* status to the user.
getErrors	Controller	Website	Provision of the encountered errors to the user.
register	Controller	Node	Registration of a new Worker Node.
publishProgress	Controller	Node	Periodic provision progress information by a Worker Node.
notifyError	Controller	Node	Provision of an encountered errors by a Worker Node.
stopJob	Node	Controller	Cancellation of a running job by the Controller.
requestStatus	Node	Controller	Provision of a Worker Node's current status to the Controller.
submitJob (RPC)	Node	Controller	Handles the assignment of a new job by the Controller.
notifyReady (RPC)	Controller	Node	Handles the notification about a completed job at the Controller.

Table 3: Protocol Methods Overview

The next section gives an overview of the platform's operation from a technical point of view.

5.5 Overall Platform Operation

This subsection covers the basic operations of the distributed password cracking platform and its workings. The user has a choice between several different usage scenarios, described in Section 4.1. Below, the different *use cases* are described on a technical level and are mapped to the *methods* described in Section 5.4.2.

New Job

When a user wants to crack a list of hashes, he can use the New Job function on the website. There she enters the parameters needed for the cracking and uploads the file with hashes to be cracked. When submitted, the website makes a request, using the *submitJob* method, to the Controller, which stores the task's data into its data store (creating an entry into its *Jobs* table).

The system runs its Job Dispatcher at specified time intervals (for example every minute). The Dispatcher checks if there are any Worker Nodes available for the cracking of hashes. If there are Workers available, it checks whether the job can be carried out by those worker, using its *strategy module*. If this is possible, the *strategy module* determines the cracking strategy and splits the *job* into several *subjobs*, which are also stored in the Controller's data store. The entries in the *Subjobs* table contain all the information about cracking of the hashes, related to a *subjob*. These *subjobs* are then dispatched by the Dispatcher module to the different Worker Nodes, using the Workers' *submitJob* method.

While the Workers are cracking hashes, they sends intermediate status updates to the Controller (using the *publishProgress* method). The status updates contain newly cracked hashes. It is also possible for the Controller to ask for status updates directly. When a worker finishes its task, it makes a request to the *notifyReady* method of the Controller. With this method, the node notifies the Controller that it has finished working. Afterwards, the *job* is being removed from the Worker's data store and the Worker becomes available for a new task.

When the controller notices all *subjobs* related to a task are finished, it notifies the user about the task results.

Get Status

The user asks for the status of a *job* using the web interface. A request is then made, using the *requestJobStatus* method, to the Controller. The Controller then returns the aggregated *job* status, based on the data in its data store. The data is accumulated when the various Worker Nodes submit status updated, regarding the *job*, by using the *publishProgress* Controller method.

Stop Job

A user can stop a job with the Stop Job function on the website. A request is made by the website, using the *stopJob* method, to the Controller. The Controller then sends *stopJob* requests to Workers working on this *job*. The Workers then stop cracking and remove all data, related to this job. After all Workers have stopped, the Controller replies positively to the website.

Show History

A user can view the task history by using the History functionality of the website. A request is made by the website, using the *requestJobList* method of the Controller. The Controller returns a list of known *jobs*, based on the data in its data store. The list includes both active and historical *jobs*.

Delete Job

A user can delete tasks by using the Delete functionality of the website. The website then sends a *stopJob* request to the Controller. If the *job* is actively being executed by Workers, the Controller sends a *cancelJob* request to every Worker. Afterwards, it deletes all data related to this job and replies positively to the website.

5.6 Summary

This chapter provided the technical design specification for the creation of a distributed password cracking platform. The architectural design was outlined in Section 5.1, with detailed explanations about the Controller component and the Worker components provided in Sections 5.2 and 5.3.3. The communication protocol was outlined in Section 5.4, and the overall platform operation was explained in Section 5.5.

With the functional and technical specifications, we were able to focus on the creation of a proof of concept implementation of the ideas outlined in this report. The next chapter provides an overview of the progress and level of completeness of this first implementation of the distributed password cracking platform.

6 Proof of Concept

As part of this project, we created a proof of concept implementation, based on the functional requirements and technical design described in the previous chapters (see Chapters 4, 5). The basic functions and methods were implemented. This chapter focuses on how this is done and what software was used for the implementation of the identified logical components.

Firstly, we provide a basic overview of the proof of concept and its workings. The chapter continues with setup instructions, explanations about the different files and functions, and ends with the technical workflow of the use cases in the context of this implementation.

The proof of concept code can be obtained at: https://github.com/dpcp.

6.1 Overview & Scope

A proof of concept implementation is created based on the technical requirements. This test environment for the proof of concept consisted of a Controller machine and two Worker Nodes. This test setup is shown in Figure 12 and works as the technical requirements describe. An overview of the development state of all logical components and their methods is given later in this section (see Table 4).

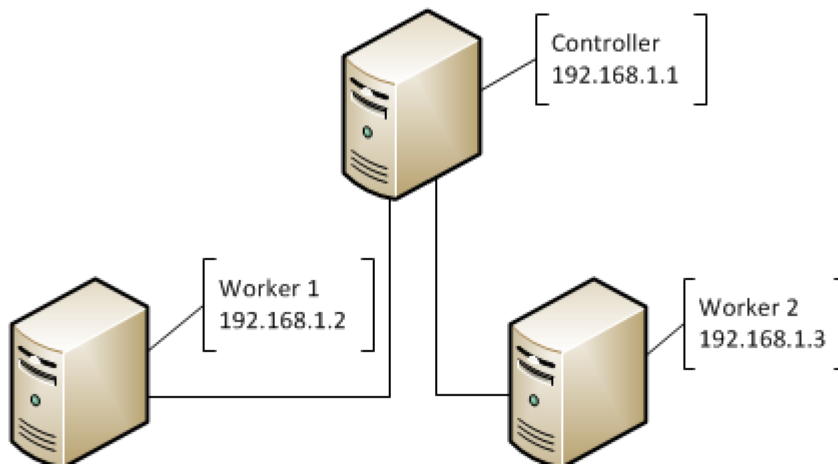


Figure 12: Proof of Concept setup

As this project was limited with regards to time, this first implementation of the distributed password cracking platform is a basic one. The main goal was to implement the communication components and one external cracking tool. This means that the Controller and the Worker components are implemented in a simple way, with a focus on communication and data storage. The front end is very simple and needs future development. Also, the cracking strategy was not within the scope of the project and is effectively not implemented.

For the Controller, the five actions a user should be able to make are implemented in the so called Communicator component. The Job Dispatcher is also implemented, but has uses a *very* simple cracking strategy.

The Worker Node is capable of running *John the Ripper* on Linux, but Joh-specific options have not been implemented. This means that the Worker can only handle raw MD5 hashes, using the default attack method of John (first dictionary attack, then brute-force). However, the communication functionality was implemented and, deriving from the existing John code, one can add support for more tools easily.

There are some software requirements for running the proof of concept code:

- Apache (or other PHP-supporting web server)
- PHP 5.2+
- PHP support for cURL
- MySQL (Controller only)
- PHP support for MySQL (Controller only)
- PHP support for SQLite 2 (Worker only)
- john 1.7.9 (Worker only)

Also, for the Worker Node, the control of the cracking tool uses Linux-only tools, such as *kill* and *pwd*. It should be possible to run the code on other operating systems with only small modifications. However, in the current state of the code, it only runs on Linux.

Component	**Method**	**Progress**	**Used**
Front-end:	Website	Very Simple	HTML
Controller: Communicator	listener	Finished	PHP
	requestJobStatus	Finished	PHP
	submitJob	Finished	PHP
	requestJobList	Finished	PHP
	stopJob	Finished	PHP
	deleteJob	Finished	PHP
	register	Finished	PHP
	publishProgress	Finished	PHP
	notifyReady	Finished	PHP
	notifyError	Finished	PHP
	requestSystemStatus	Not yet implemented	PHP
	database	Finished	PHP
	getErrors	Finished	PHP
Controller: Dispatcher	startdispatcher	Finished	PHP
	nodechecker	Finished	PHP
	strategy	Very simple needs strategies	PHP
	dispatcher	Finished	PHP
	database	Finished	PHP
Controller: Data store	Database	Finished	MySQL
Worker: Common Code	startup	Finished	PHP
	listener	Finished	PHP
	requestStatus	Finished	PHP
	acceptJob	Very simple (only john)	PHP
	stopJob	Finished	PHP
Worker: Tool specific Code	john	Simple (without parameters)	PHP
	Others	Not yet implemented	PHP
Worker: Data Store	Database	Finished	SQLite

Table 4: Proof of Concept development state.

6.2 Directory Overview and File Explanations

In this section, we give an overview of the different source files. For details about the individual files, refer to Appendix F.

6.2.1 Controller: Communicator

An overview of the Communicator directory contents is given below:

```
Controller
├── index.php
├── Includes
│   ├── functions.php
│   ├── database.php
│   ├── register.php
│   ├── submitJob.php
│   ├── stopJob.php
│   ├── deleteJob.php
│   ├── requestJobStatus.php
│   ├── requestJobList.php
│   ├── notifyReady.php
│   ├── publishProgress.php
│   ├── requestSystemStatus.php
│   ├── notifyError.php
│   └── getErrors.php
└── Front-End
    └── communicator.test.html
```

Explanations about each of the Communicator's files can be found in Appendix F.1.

6.2.2 Controller: Dispatcher

An overview of the Dispatcher directory contents is given below:

```
Dispatcher
├── database.php
├── startdispatcher.php
├── nodechecker.php
├── strategy.php
└── dispatcher.php
```

Explanations about each of the Dispatcher's files can be found in Appendix F.2.

6.2.3 Worker: Common Code

This section shows the common code part of the system. The Worker has some tool-specific files, which are discussed later in this chapter. Below, the whole *Worker* directory tree is shown. A detailed explanation about each file can be found in Appendix F.3.

```
Worker
├── db
│   └── worker.db
├── includes
│   ├── Connectors
│   │   └── john.php
│   ├── acceptjob.php
│   ├── requestStatus.php
│   ├── stopJob.php
│   └── functions.php
├── test
│   └── worker.test.html
├── working
│   └── files when tool is running
├── config.php
├── startup.php
└── index.php
```

6.2.4 Worker: Tool Specific Code

This section explains how the tool-specific code works. As the current implementation is basic, this part of the proof of concept will be of interest for future research – a lot of work can be done here by adding more modules/tools or expanding the functionality of what is already implemented. For information about adding a new tool refer to Section 6.3.4.

john.php The *john.php* file is called by the *acceptJob.php* file. It first reads all information it needs out of the SQLite database of the Worker. Then, it gets all the hashes and puts them in a format, which is recognised by John the Ripper. Then the command for starting John is stored in the variable *$command*. Afterwards, the John process is started and the main file descriptors are kept in an array.
While running an external process, that process is monitored in a while loop. This loop sends the HUP signal to the running John process, which causes John flush its current status back to a file. This file is then read in the while loop, and if there is any new result, the result is sent to Controller. If the process crashes or something unexpected happens, a *notifyError* message is sent to the Controller. If everything went well and the process exits normally, the PHP script sends a notifyReady message to the Controller and deletes all locally stored data about that job.

Working directory The Worker has a *working* directory. The worker stores John-related working files in this directory. These files contain (cracked) hashes, or intermediate progress information. These files are removed after the John process exits normally.

6.3 Getting Started with the Proof of Concept

This section describes how you can get started with the current implementation of the platform. Information about what software is required and where permissions have to be changed is given.

6.3.1 Setting up the Controller

Before one can start, she has to make sure she has the software listed in Section 6.1 installed.
The first thing that has to be done is to import the database structure (found in the *database* folder) into MySQL. Also, the contents of the *controller/includes/database.php* file need to be adapted accordingly.
Afterwards, the Controller's communication-related code needs to be set up. This is done by placing the contents of the *controller* folder into the web-root of the running web server.
Lastly, the Controller's dispatching and strategy-related code needs to be set up. This is done by adding a cronjob on the system, which runs the *dispatcher.php* file through the PHP command-line interpreter every N minutes.
The Controller needs the two directories called *controller* and *dispatcher*. The first one, *controller*, needs to be the accessible through the web server, running on the Controller machine. The *controller* folder contains the communication-related code for the Controller. The *dispatcher* folder contains the strategy and dispatching capabilities of the Controller.
You'll need to setup the database by creating your own or importing the one that was already designed. After this you need to modify the *database.php* file with the correct parameters for this database.
As the Controller's components currently do not write a log file, no special permissions are needed. However, it is recommended to transfer ownership of the files to the user, running the web server.

6.3.2 Setting up the Worker

Before one can start setting up the Worker, she has to make sure she has the software listed in Section 6.1 installed.
Firstly, the Worker's code needs to be placed in the web-root of the running web server on the Worker machine. The Worker code is located in the *worker* folder.
Afterwards, the Worker needs to be configured. This is done by modifying the parameters in the *config.php* file.
Then, the correct ownership/permissions need to be set. The Worker needs read/write permissions for the *working* and *database* directories and their contents.
Finally, the Worker needs to be registered with the Controller. This is done by running the *startup.php* file through the PHP command-line interpreter.

6.3.3 Using the System

The system can currently only use John the Ripper with MD5 hashes with the default cracking strategy of John – it first uses a dictionary attack and then continues with a brute-force attack.
To use the system, we created a simple web-interface in the *front-end* directory. That interface serves mainly for data input and obtaining the status of workers. One can invoke the different methods of the Controller using that interface. However, the interface does not parse the output of the Controller, which is pure JSON.
To check whether everything works as expected, one can run the *startdispatcher.php* script from the command line. If there are any jobs in the database, that script should process them, create subjobs, and dispatch them to the registered Worker Nodes. When the dispatcher exits, one can either check the working of the system and the Worker Nodes' by invoking the appropriate status commands, or wait for hashes to be cracked and reported back to the Controller.

6.3.4 Adding New Tools to the Worker

The system has the ability to add functions to current tools or adding new ones. Functions are being called by the *acceptJob.php* file, this file should start the so called: 'cracking tool specific code' part of the worker. Currently only john is simple implemented. The acceptJob file should determine on what function should be used this can be accomplished by a simple read from the database and making decisions based on that. The difficult part is creating the tool specific code by making it able to send intermediate update back to the controller, so you should first be focusing on implementing it without those intermediate updates. Below the few essential lines for the current john implementation are described:

```
$command = JOHN . " --nolog --config=" . dirname (JOHN) . "/john.conf hashes-job-{
    $_POST['id']} --pot=hashes-job-{$_POST['id']}.pot --format=raw-md5 --session=
    hashes-job-{$_POST['id']}";
```

This line calls the john application in a particular directory. It takes the john config file and hashes file. We also specify which format the hash is in. For adding more supported hashes this should be changed to a variable and read from the database. You can also add different parameters like how the tool should function and what dictionary it should use.

```
$process = proc_open ($command, $descriptorspec, $pipes, $cwd);
```

This line starts the line of code explained above together with a pipe. By using a pipe we can read and write to the process while its running. It also keeps the process ID so that we know if it's still running and can kill the job easily if it is needed.

```
exec ("cd $cwd && kill -s HUP {$pstatus['pid']} && " . JOHN . " --status=hashes-job-{
    $_POST['id']} 2>&1", $output);
```

John does not output intermediate updates easily we had to send a kill hup command to the john process for outputting the current status of the running john process. This output is read and

scanned for new hashes that have been cracked and the result is send back to the controller. This is maybe not needed with other tools since they may have a easier way of outputting the intermediate status.

6.4 The PoC Workings through the Different Use-Cases

The workings of the PoC will be explained through the different use-cases a user can take which are explained. (see Section 4.1). This way the whole use-case with technical workflow and files used will be explained. For more detail you can download the code here: [LINK] or e-mail us for further questions.

6.4.1 New Job

The first task a user should take is adding a new job into the system:

1. User submits new job request through the website (index.html)
2. Message goes to the listener of the controller (index.php)
3. Communicator is listening and accepts the submitted job when credentials are correct. (submitJob.php)
4. Communicator Stores the new job information into the database (submitJob.php)
5. Waiting for dispatcher do be executed (every 1 min for instance - startdispatcher.php)
6. When dispatcher runs it checks for available jobs or subjobs and selects the newest (nodechecker.php)
7. Dispatcher checks for nodes which are available (nodechecker.php)
8. Determines based on this and the job method a cracking strategy (strategy.php)
9. The cracking strategy creates different subjobs for a job (strategy.php)
10. Now the dispatcher does the real dispatching of the subjobs to available workers (dispatcher.php)
11. Worker starts cracking subjob with tool available (connectors/<tool>.php)
12. Worker sends intermediate updates to the communicator hashes cracked and progress percentage (publishProgress.php)
13. When the worker is finished it sends the whole cracked list to the controller (notifyReady.php)
14. The worker cleans everything up (connectors/<tool>.php)
15. If the worker cracked all hashes the communicator stops all other workers (stopJob.php)
16. When a user asks for the status it returns it (requestJobStatus.php)

6.4.2 Get Status

While the job is running the user can ask for the job status:

1. User submits get status request through the website (index.php)
2. Communicator is listening and accepts the status request when credentials are correct. (requestJobStatus.php)
3. Communicator returns the status back of the jobs select (requestJobStatus.php)

6.4.3 Stop Job

The user can decide to stop a running job:

1. User submits a stop job request through the website (index.php)
2. Communicator is listening and accepts the stop job request when credentials are correct. (stopJob.php)
3. Communicator gives status/response back (stopJob.php)

6.4.4 Show History

The user can show its own job history:

1. User submits a show history request through the website (index.php)
2. Communicator is listening and accepts the show history request when credentials are correct. (requestJobList.php)
3. Communicator returns the job history back (requestJobList.php)

6.4.5 Delete Job

The user can delete old or running jobs:

1. User submits delete job request through the website (index.php)
2. Communicator is listening and accepts the delete job request when credentials are correct. (deleteJob.php)
3. Communicator runs cancels currently running subjobs on workers (stopJob.php)
4. Communicator returns the result back to the user (deleteJob.php)

6.4.6 Worker Registration

In addition to the different use-cases there is one other scenario possible for a user this is the first registration of a worker to the controller:

1. The admin/administrator first changes the configuration file needed for the worker (config.php)
2. He can then start the startup script at the worker to register itself with the controller (startup.php)
3. A register request is send to the communicator (register.php)
4. Worker response with a OK message when registering is finished (startup.php)
5. Worker updates its own database with worker information (startup.php)
6. Worker stays idle in the listening state till the dispatcher or communicator send a request (acceptJob.php, requestStatus.php, stopJob.php)

6.5 Summary & Advice

We outlined the Proof of Concept overview and scope with a short description of the functions of each file. After this we discussed whats needed to setup the distributed password cracking platform and use it. Lastly the different use-case workings are explained.
Having developed the Proof of Concept we can say that adding a new tool can be time consuming. However, the basis of the system is ready and an example (John the Ripper) is implemented. Implementing the intermediate update functionality for John took a lot of time and work until it functioned correctly. This may be easier with other tools. The real challenge in improving this platform lies in adding support for more tools, a proper cracking strategy and interpretation of the tool parameters.

7 Conclusion

This project aimed to answer the following research question:

> *How can a scalable, modular and extensible middleware solution be designed for the purpose of password cracking, so that it is based on existing cracking tools and allows for the use of distribution and of a dynamic and adjustable cracking strategy?*

During our research, we worked towards answering the main research question by answering the identified subquestions. Thus, this section restates the research subquestions and summarises the findings of the project.

7.1 Theoretical Research – Architectural and Communication Models

Subquestion 1: What are the best architectural and communication models to use in such a system?
To identify the most suitable architectural model for the Distributed Password Cracking Platform, we examined the two main types of system architectures – centralised and decentralised. We outlined the strengths and weaknesses of each and looked into their general applicability and usefulness in the context of this project. We compared centralised and decentralised architectures, taking into consideration several system design criteria (see Section 3.1.3). Based on that comparison, we concluded that a centralised architecture will be more suitable for this project than a decentralised one.
For identifying the most suitable communication model for the platform we undertook a similar approach – we examined four widely used communication models – message-oriented, stream-oriented, multicast and RPC. Again, we outlined their applicability and usefulness in the context of the project, comparing them on several criteria (see Section 3.2.5). Based on the comparison, we concluded that message-oriented communication is the best model to undertake for this project. However, we identified one specific case, in which Remote Procedure Call will be used.
We also made an analysis of the cracking tools available for use within such a platform. Analysing and comparing them with regards to their cracking functionality, we identified two tools that will be used within the password cracking platform – John the Ripper for CPU-based password cracking and oclHashcat-plus for GPU-based cracking (see Section 3.3.4).

7.2 Functional Requirements

Subquestion 2: What are the functional requirements of such a platform?
After researching the possible architectural and communication models, we created the functional specification for the platform. Examining the project requirements and the platform's use cases, we were able to identify the main functional requirements of the platform (see Section 4.2). Based on these, we described in detail the functional requirements of all components of the system (see Section 4.3).

7.3 Technical Design

Subquestion 3: Based on the functional requirements, what is the best technical design for such a platform?

The technical design of the platform was based entirely on the research findings and on the created functional specification. Applying the research findings of this project, we designed a centralised system architecture, which makes use of message-oriented controller-worker communication. For the execution of tasks by the workers, Remote Procedure Call was used.
We described the top-level system architecture (Section 5.1), its logical components and the interconnections between them, as well as the the workflows for each of the components (Sections 5.2, 5.3.3). We proceeded with defining the communication protocol, the data structures and the messages needed for communication between the nodes in the system.

7.4 Proof of Concept

Subquestion Optional: Can a test implementation (proof of concept) be created based on this research?
We created a test implementation of the platform to serve as a proof of concept. The test implementation was created using PHP, MySQL, Apache and SQLite. The code implements John the Ripper as a cracking tool and allows only for brute-force CPU-based cracking of hashes. This implementation uses a very basic cracking strategy – it creates identical subjobs for each available worker. Also, it only supports raw MD5 hashes. We tested the code on Ubuntu Server, versions 10.04.3 and 11.10, and on Debian Lenny. We used John the Ripper, version 1.7.9-jumbo5.

7.5 Future Work

The current system architecture, communication and proof of concept code lay the foundation for the cracking platform as a whole. However, due to time constraints, we examined certain parts of the platform from a high-level only. Below, we give our recommendations for future research.

Communication Protocol It may be interesting for future research to extend the communications protocol. This depends on the capabilities of the tools used and whether they support features, that were not taken into consideration in the current research – features like collaborative computation, keyspace distribution, etc.

Cracking Strategy The cracking strategy refers to the decisions made by the platform, regarding the usage of resources for the execution of a job. These may include whether to use CPU or GPU, whether to assign the task to a single node, or to multiple nodes, whether to distribute hashes or the keyspace. The cracking strategy was out of the scope of this project and needs more work. As the strategy is crucial to the working of the whole platform, it is necessary that it is examined in future research.

Proof of Concept The test implementation that we were able to produce is a rudimentary one – it has limited capabilities with regards to available cracking tools, available hashtypes, checking and security. All of these aspects need to be addressed in future work. Other areas that require additional work include:

- Benchmarking
- Scalability testing

- Testing on multiple operating systems
- Securing the communications protocol
- Developing a front-end for the platform

A Acronyms

API Application programming interface

BOINC Berkeley Open Infrastructure for Network Computing

CBEA Cell Broadband Engine Architecture

CPU Central Processing Unit

C/S Client-Server

CUDA® Compute Unified Device Architecture

DHT Distributed Hash Table

FPGA Field-Programmable Gate Array

GPGPU General-Purpose Graphics Processing Unit

GPU Graphics Processing Unit

GUI Graphical User Interface

HTTP Hyper-Text Transfer Protocol

IC Integrated Circuit

ISP Internet Service Provider

JSON JavaScript Object Notation

KPMG Klynveld Peat Marwick Goerdeler

MPI Message Passing Interface

OS Operating System

P2P Peer 2 Peer

RMI Remote Method Invokation

RPC Remote Procedure Call

SIMD Single Instruction Multiple Data

SNMP Simple Network Management Protocol

SoC Seperation of Concern

UvA University of Amsterdam

B Three-tier model

The three-tier architectural model is used within many web-based systems. The basic organisation of the model is shown in Figure 13. The three tier architecture uses the following layers:

- *Tier 1:* The presentation layer
  The presentation layer is used as a user-interface an usually is a website and can act as the managing part. A technical tool for this is for example *Apache*.
- *Tier 2:* The business logic layer
  The business layer is used as the smart/logic part. It usually contains the software that handles the computations needed at the presentation layer. Tools used for this are often languages like: *Perl, Python, PHP, Ruby etc.*
- *Tier 3:* The data layer
  The data layer contains the actual data that is used by the business layer. Its the place where everything gets stored. For the data layer usually the tool *MySQL* is used as database.

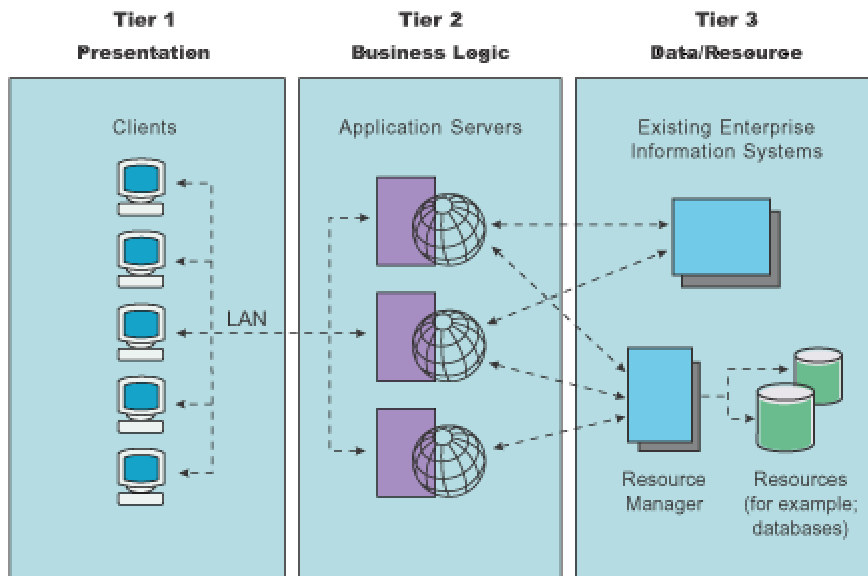


Figure 13: Three tier model. [38]

C Peer 2 Peer (P2P)

The P2P architecture is a type of decentralised architecture. Typical to decentralised architectures, the purpose of P2P is to act as both a client and server. P2P systems make use of overlay networks, which allow new nodes to easily join and leave a network in a structured way. There are three kinds of P2P systems – structured, unstructured and superpeer-based.

- Structured P2P systems
  A structured P2P system makes use of a overlay networks with a so called DHT system. A example of such a system is *Chord* which organises nodes in a logical manner, this can for example be the distance between nodes. Another possible way for a structured P2P system is making use of a d-dimensional space for organising new nodes in.

- Unstructured P2P systems
  As described there are also unstructured P2P systems which make use of randomised algorithms for creating an overlay network. A example of such a algorithm is *gossiping*. The idea of a unstructured P2P system is to be randomly select other nodes for creating a network and no structure or organisation of nodes is needed.

- Superpeer systems
  For unstructured P2P systems it can become problematic when the network becomes larger. When the network is large the routing between the nodes is far from optimal. There for something called a *superpeer* architecture is created. (See Figure 14) With such a system some nodes in a network become a superpeer. Superpeers have usually more performance and availability. New nodes will join a superpeer, when too many nodes are connected to a superpeer a new one gets created by promoting a regular peer into a superpeer.

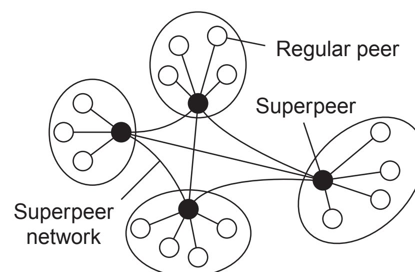


Figure 14: Superpeer architecture. [11]

D Data Structures

D.1 Hash

The Hash data structure contains information about a single subjob that has already been dispatched to a worker.

Parameter	**Type**	**Meaning**
hash	string	The hash value
result	string	The recovered value
timestamp	long	The timestamp of recovery (Unix timestamp format)

Table 5: Job Data structure

D.2 Job

The Job data structure contains information about a single subjob that has already been dispatched to a worker.

Parameter	**Type**	**Meaning**
id	int	The identifier of this subjob
username	string	The name of the hashtype used
comment	string	The name of the hashtype used
hashtype	string	The name of the hashtype used
minlength	int	The minimum length of the password
maxlength	int	The maximum length of the password
submitted	long	The time of submission (Unix timestamp format)
expires	long	The time of expiry (Unix timestamp format)
method	string	The name of the cracking method used
alphabet	string	The name of the alphabet used
customalphabet	string	The custom alphabet used
crackedHashes	array(Hash)	An array which contains the hash and hash result
percentageComplete	int	The percentage of completed checks

Table 6: Job Data structure

D.3 Subjob

The Subjob data structure contains information about a single subjob that has already been dispatched to a worker.

Parameter	**Type**	**Meaning**
id	int	The identifier of this subjob
hashtype	string	The name of the hashtype used
method	string	The name of the cracking method used
alphabet	string	The name of the alphabet used
submitted	long	The time of submission (Unix timestamp format)
percentage	int	The percentage of completed checks
minlength	int	The minimum length of the password
maxlength	int	The maximum length of the password

Table 7: Subjob Data structure

E Method Definitions

E.1 submitJob

Method Description

This method is part of the website-facing interface of the Controller. It deals with the submission of a cracking job to the Controller.
When a request message for this method is submitted by the website, a list of hashes has to be supplied. Also, all of the necessary job parameters need to be supplied – type of hash, additional information, regarding salting, hash generation, cracking strategy, etc. The Controller should accept the job, storing the parameters and data into its local database, and should reply with a Reply data structure, returning no data.
Note: The parameters alphabet and customalphabet are mutually exclusive; i.e. exactly one of those must be specified.

Method Parameters

Parameter	Type	Description
username	string	The username of the user requesting cracking
password	string	The SHA-256 hash of the password of the user
hashtype	string	The name of the hashtype to use while cracking
minlength	int	(Optional) The minimum length of the needed plaintext password
maxlength	int	(Optional) The minimum length of the needed plaintext password
expires	long	(Optional) Unix timestamp of the expiry time of this job
method	string	The name of the cracking method to use
alphabet	string	The predefined name of the alphabet to use while cracking
customalphabet	string	The characters of a custom alphabet to use for cracking
file	file	The file, containing the hashes to crack

Response

Parameter	Type	Meaning
status	enum(OK, FAIL)	Whether the request succeeded or failed
error	string	The error string (if applicable)
result	mixed	Empty for this request.

E.2 stopJob

Method Description

This method is part of the website-facing interface of the Controller. It deals with the stopping of a job.
When a request message for this method is submitted by the website, the identifier of the job to be stopped needs to be supplied. The Controller should notify all working nodes about the cancellation and reply positively, returning no data. Note, that with this method, the job data is not removed from the Controller, but is kept in the local data store.

Method Parameters

Parameter	Type	Description
username	string	The username of the user requesting cracking
password	string	The SHA-256 hash of the password of the user
jobid	int	The ID of the job to stop

Response

Parameter	Type	Meaning
status	enum(OK, FAIL)	Whether the request succeeded or failed
error	string	The error string (if applicable)
result	mixed	Empty for this request.

E.3 deleteJob

Method Description

This method is part of the website-facing interface of the Controller. It deals with the removal of a job.
When a request message for this method is submitted by the website, the identifier of the job to be stopped needs to be supplied. The Controller should first stop all nodes working on this job, then remove all job-related data from the local data store. Then, the Controller should reply positively to the website, returning no data.

Method Parameters

Parameter	**Type**	**Description**
username	string	The username of the user requesting cracking
password	string	The SHA-256 hash of the password of the user
jobid	int	The ID of the job to stop

Response

Parameter	**Type**	**Meaning**
status	enum(OK, FAIL)	Whether the request succeeded or failed
error	string	The error string (if applicable)
result	mixed	Empty for this request.

E.4 requestJobList

Method Description

This method is part of the website-facing interface of the Controller. It deals with the provision of a list of submitted active and historical jobs.
When a request message for this method is submitted by the website, the Controller should reply positively, returning an array of Job data structures – all the jobs it currently has stored in its local data store. The list should include both active and historical jobs.

Method Parameters

Parameter	Type	Description
username	string	The username of the user requesting cracking
password	string	The SHA-256 hash of the password of the user

Response

Parameter	Type	Meaning
status	enum(OK, FAIL)	Whether the request succeeded or failed
error	string	The error string (if applicable)
result	array(Job)	An array of Job datastructures

E.5 requestJobStatus

Method Description

This method is part of the website-facing interface of the Controller. It deals with the provision of the current status of a running job.
When a request message for this method is submitted by the website, the Controller should reply positively, providing the current status of the processing of a job.

Method Parameters

Parameter	**Type**	**Description**
username	string	The username of the user requesting cracking
password	string	The SHA-256 hash of the password of the user
jobid	int	The ID of the job, for which status is needed

Response

Parameter	**Type**	**Meaning**
status	enum(OK, FAIL)	Whether the request succeeded or failed
error	string	The error string (if applicable)
result	Job	A Job datastructure, containing Job information

E.6 requestSystemStatus

Method Description

This method is part of the website-facing interface of the Controller. It deals with the provision of the current system status.
When a request message for this method is submitted by the website, the Controller should reply with a SystemStatus data structure, containing the current state of the platform. The reply should include the number of working and idle nodes, the number of active jobs, the number of hashes to be cracked, the number of cracked hashes so far and the amount of time that the Controller has been active.

Method Parameters

Parameter	Type	Description
username	string	The username of the user requesting cracking
password	string	The SHA-256 hash of the password of the user

Response

Parameter	Type	Meaning
status	enum(OK, FAIL)	Whether the request succeeded or failed
error	string	The error string (if applicable)
result	SystemStatus	A SystemStatus datastructure, containing system status information

E.7 getErrors

Method Description

This method is part of the website-facing interface of the Controller. It deals with the provision of the list with errors.
When a request message for this method is submitted by the website, the Controller should reply with a list of all recorded errors until now.

Method Parameters

Parameter	Type	Description
jobid	int	The ID of the job/subjob related to the error
error	string	The encountered error message

Response

Parameter	Type	Meaning
status	enum(OK, FAIL)	Whether the request succeeded or failed
error	string	The error string (if applicable)
result	mixed	Empty for this request

E.8 register

Method Description

This method is part of the node-facing interface of the Controller. It deals with the registration of a new node.
When a request message for this method is submitted by a new node, the node should also supply the necessary parameters, including the node's capabilities (see Appendix E). The Controller should record the node's existence and capabilities in its local data store, issue a unique identifier for the new node and reply positively, returning the new node identifier in the result section of the reply.

Method Parameters

Parameter	Type	Description
name	string	The hostname of the new node
cpu	boolean	Whether the node has CPU capabilities
gpu	boolean	Whether the node has CPU capabilities
tool	string	The name of the tool the node uses to crack

Response

Parameter	Type	Meaning
status	enum(OK, FAIL)	Whether the request succeeded or failed
error	string	The error string (if applicable)
result	int	The new node ID

E.9 publishProgress

Method Description

This method is part of the node-facing interface of the Controller. It deals with the periodic provision of node progress to the Controller.
When a request message for this method is submitted by a node, the node should supply the necessary parameters – its ID, the identifier of the job/subjob that the node is working with, the percentage of completeness of the job and the estimated time until the job/subjob is complete. If there have been new cracked hashes since the previous progress update, the node should also include those in the request. After receiving the request, the Controller should process and store the provided data, and reply positively to the node.

Method Parameters

Parameter	Type	Description
id	int	The node ID of the node making the request
jobid	int	The ID of the job this node is working on
percentage	int	The progress on the current job
time	int	Estimated time until completion (in hours)
hashes	string	A serialized array of Hash data structures, containing cracked hashes

Response

Parameter	Type	Meaning
status	enum(OK, FAIL)	Whether the request succeeded or failed
error	string	The error string (if applicable)
result	int	The new node ID

E.10 notifyError

Method Description

This method is part of the node-facing interface of the Controller. It deals with the provision of encountered errors to the Controller.
When a request message for this method is submitted by a node, the node should supply the necessary parameters – its ID, the identifier of the job/subjob that the node is working on and the encountered error message. The Controller should reply positively, storing the error in its data store.

Method Parameters

Parameter	**Type**	**Description**
id	int	The node ID of the node making the request
jobid	int	The ID of the job this node is working on
error	string	The encountered error message

Response

Parameter	**Type**	**Meaning**
status	enum(OK, FAIL)	Whether the request succeeded or failed
error	string	The error string (if applicable)
result	mixed	Empty for this request

E.11 submitJob

Method Description

This method is part of the node's RPC provisioning. It handles the assignment of a new job by the Controller.
When a call for this method is made by the Controller, the call should also pass the required job parameters, related to keyspace, hash type and restrictions. The list of all targeted hashes should also be supplied. The node should process all parameters and data and should store them in its local data store. It should determine whether it is capable of handling the requested task and reply to the Controller appropriately. Afterwards, the node should invoke the external cracking tool, starting the process of executing the task.

Method Parameters

Parameter	**Type**	**Description**
id	int	The node ID of the node making the request
hashtypeid	int	The ID of the used hashtype
alphabetid	int	The ID of the used alphabet
customalphabet	string	All characters of the alphabet to use for cracking
minlength	int	The minimum length of the plaintext password(s)
maxlength	int	The maximum length of the plaintext password(s)
methodid	int	The ID of the cracking method to use
hashes	string	A serialised array of Hash data structures

Response

Parameter	**Type**	**Meaning**
status	enum(OK, FAIL)	Whether the request succeeded or failed
error	string	The error string (if applicable)
result	mixed	Empty for this request

E.12 notifyReady

Method Description

This method is part of the Controller's RPC provisioning. It handles the notification about a completed job.
When the node makes a call for this method, it should provide the necessary parameters, containing information about the execution and success status of the job in question. If there are cracked hashes, that have not been submitted before, those should be included as well. The Controller should process the received data and store it in its local data store. Afterwards, the node should remove all stored data related to the job in question.

Method Parameters

Parameter	Type	Description
id	int	The node ID of the node making the request
jobid	int	The ID of the job this node is working on
hashes	string	A serialized array of Hash data structures, containing cracked hashes

Response

Parameter	Type	Meaning
status	enum(OK, FAIL)	Whether the request succeeded or failed
error	string	The error string (if applicable)
result	mixed	Empty for this request

E.13 requestStatus

Method Description

This method is part of the node communication interface. It deals with the provision of the node's capabilities and current load.
When a request message for this method is submitted by the Controller, the node should reply with its current status, this is its load and a ready parameter which is a simple reply data structure message with OK. The reply should contain the node's current assigned and running jobs, the load of the system and the node's capabilities.

Method Parameters

None

Response

Parameter	Type	Meaning
status	enum(OK, FAIL)	Whether the request succeeded or failed
error	string	The error string (if applicable)
id	int	The ID of the node
load	string	The load string of the node (Unix-style)
cpu	boolean	Whether the node uses CPU computational power
gpu	boolean	Whether the node uses GPU computational power
tool	string	The name of the tool used for cracking
jobcount	int	The number of jobs this node is currently processing
joblist	array(Job)	An array of Job data structures

E.14 stopJob

Method Description

This method is part of the node communication interface. It deals with the cancellation of a running job.
When a request message for this method is submitted by the Controller, the identifier of the job/-subjob to be cancelled should also be supplied. When the node receives this message, it should kill all associated instances of the cracking tool process and discard any stored data regarding the job/subjob. The node should then reply positively to the Controller and cleanin itself up.

Method Parameters

Parameter	Type	Description
jobid	int	The ID of the job this node is working on

Response

Parameter	Type	Meaning
status	enum(OK, FAIL)	Whether the request succeeded or failed
error	string	The error string (if applicable)
result	mixed	Empty for this request

F Proof Of Concept File Explanations

F.1 Controller: Communicator

index.php The *index.php* file is the entry point of the Controller's Communicator. It determines which method is used with an incoming request and loads the appropriate handling file (from the *includes* directory).

functions.php The *functions.php* file is loaded automatically in the *index.php* file, regardless of the type of request being handled. It contains basic communication and authentication functions that may be used at any point in the code. The functions are given below.

- *simpleReply*: The basic reply function. It prints JavaScript Object Notation (JSON)-encoded data and ends the script's execution when called.
- *post*: Convenience wrapper around cURL.
- *checkCrendentials*: Convenience function for checking user credentials against the database.

database.php The *database.php* file is loaded automatically in the *index.php* file, regardless of the type of request being handled. It contains the database credentials and performs the actual connection call.

register.php The *register.php* file handles Worker Node registrations. It is loaded by *index.php*. The code in this file processes the passed parameters and adds them to the database (creating an entry in the Nodes table). It replies with the newly generated ID and exits.

submitJob.php The *submitJob.php* file handles job submission requests from the website. It is loaded by *index.php*. The code processes the job parameters and stores them in the database.

stopJob.php The *stopJob.php* file handles job-stopping requests from the website. It may be included either by *index.php*, or by *deleteJob.php*. The code sends a *stopJob* request to all Workers, working on a particular job.

deleteJob.php The *deleteJob.php* file handles job deletion requests from the website. It is loaded by *index.php*. The code first includes the *stopJob.php*, stopping the job if it is running, and then removes all related data from the database.

requestJobStatus.php The *requestJobStatus.php* file handles job status requests from the website. It is loaded by *index.php*. It aggregates job information from the database, and returns a list of cracked hashes (if any) and current job status (ETA, percentage done, etc.)

requestJobList.php The *requestJobList.php* file handles job history requests from the website. It is loaded by *index.php*. The code returns a list of all known jobs with their parameters (so both active and historical jobs).

notifyReady.php The *notifyReady.php* file handles Nodes' "ready" notifications. It is loaded by *index.php*. It sets the subjob progress to 100%, which effectively marks the job as finished. If there are any newly cracked hashes supplied with the request, those are entered into the database.

publishProgress.php The *publishProgress.php* file handles intermediate updates from Worker Nodes. It is loaded by *index.php*. It processes intermediate results and newly cracked hashes, and stores the data in the database.

requestSystemStatus.php The *requestSystemStatus.php* file handles system status requests from the website. This functionality is currently not implemented. The code should return a list of active Worker Nodes, their status and load, the uptime of the platform and other relevant information.

notifyError.php The *notifyError.php* file handles error notifications from Worker Nodes. It is loaded by *index.php*. The code stores the errors in the database.

getErrors.php The *getErrors.php* file handles error list requests from the website. The code returns a list of all recorded errors in the database.

communicator.test.html The *communicator.test.html* is a simple front-end, which can be used for testing and debugging. It sends POST requests to the Communicator. All parameters are pre-set for testing purposes. The URL of the Controller may need to be changed.

F.2 Controller: Dispatcher

database.php The *database.php* file contains the database credentials and the connection variables for making a call to the database. It is identical to the file with the same name within the Communicator's directory tree.

startdispatcher.php The *startdispatcher.php* file is the entry point for the Dispatcher. It needs to be run periodically every 1-5min. This file is a simple wrapper of the functionality defined in *nodechecker.php*.

nodechecker.php The *nodechecker.php* file begins with defining basic functions – *simpleReply* and *post*. Afterwards, the NodeChecker function is defined. These functions take care of the following tasks (in sequence):

1. looks for non-processed jobs;
2. looks for subjobs, which are ready for dispatching;
3. if any jobs/subjobs are found, it checks which nodes are available;
4. it creates a list of available nodes, showing their capabilities;
5. the list, along with the job/subjob parameters are passed to the strategy module.

The *strategy.php* file is loaded, followed by the *dispatcher.php*. Finally, execution ends.

strategy.php The *strategy.php* file determines the cracking strategy – how the jobs should be divided between the different workers and, in general, how the cracking of hashes should take place. In the current implementation, this is done in a very simple way – every node gets to do all the work, associated with a job. However, it is possible to create your own strategy module, which handles this task in a more sophisticated way. The file also has a *createsubjob* function which will create the subjob and send it to the *dispatcher.php* file.

dispatcher.php The *dispatcher.php* file requires a subjob and a node parameter. Based on those, it reads all the information from the database which is needed for the dispatching. The dispatcher output is a request to the node containing the subjob and all hashes it needs to crack.

F.3 Worker: Common code

worker.db The *worker.db* file is an SQLite database which can store the data which is needed for the worker.

config.php The *config.php* file contains the worker configuration parameters. Parameters in this folder are: controller-ip, worker-ip, node name, db-location, cpu, gpu, tool being used and the location where tool is located. These parameters are being used by the other functions.

startup.php The *startup.php* file sends a registration request to the controller using the parameter stated in the *config.php* file. After this is done successfully the worker can receive jobs from the worker (if all other requirements are set properly).

index.php The *index.php* file functions just as the controller as the listener file. All requests from the controller are first send to this file by the web-service (apache for example).

functions.php The *functions.php* file contains the global functions used by the worker. Those functions are: simpleReply, post, async_post, getDatabase, send_error, publishProgress, notifyReady. The first two are already described at the controller. *async_post*: this function is used by the *submitJob* case in the *index.php* file. Its purpose is to send a message back to itself; *getDatabase*: opens the SQLite database; *send_error*: sends as the name suggests errors to the controller; *publishProgress*: publishes intermediated updates back to the controller; notifyReady: Sends an ready message back to the controller when the worker is done.

acceptjob.php The *acceptjob.php* file does not do much at the moment. In this file the tool specific function should be called. Since we only have John the ripper implemented it can only call this file in *connectors/john.php*.

requestStatus.php The *requestStatus.php* file is used when the controller asks for the worker status. The worker sends its information back to the controller with parameters: current load, job count and tool used.

stopJob.php The *stopJob.php* file can be called when the controller requests for the worker to stop its current workings/ hash cracking. The worker deletes all its local content of the job and can be used for cracking again.

worker.test.html The *worker.test.html* file is used for testing purposes; for sending easy and fast requests back to the controller. It has some test parameters which are already filled in.

G Bibliography

[1] "About GPGPU." http://gpgpu.org/about.

[2] M. Bakker and R. van der Jagt, "GPU-based password cracking," Master's thesis, Universiteit van Amsterdam, 2010. http://staff.science.uva.nl/~delaat/rp/2009-2010/p34/report.pdf.

[3] A. Kasabov and J. van Kerkwijk, "Distributed GPU Password Cracking," Master's thesis, Universiteit van Amsterdam, 2011. http://staff.science.uva.nl/~delaat/rp/2010-2011/p11/report.pdf.

[4] F. Bauspiess and F. Damm, "Requirements for cryptographic hash functions," 1992.

[5] D. Reid and C. Knipping, *Proof in mathematics education: Research, learning and teaching*. 2010.

[6] D. Florencio and C. Herley, "A large-scale study of web password habits," 2007.

[7] R. Shirey, "RFC2828: Internet security glossary." https://tools.ietf.org/html/rfc2828.

[8] M. Hellman, "A cryptanalytic time – memory trade-off," 1980.

[9] "Rainbow tables." http://en.wikipedia.org/wiki/Rainbow_table.

[10] "Random password strength." http://www.redkestrel.co.uk/Articles/RandomPasswordStrength.html.

[11] A. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*. 2007.

[12] NVidia, "GeForce GTX590 specifications." http://www.geforce.com/Hardware/GPUs/geforce-gtx-590/specifications.

[13] S. Williams *et al.*, "The potential of the Cell processor for scientific computing," 2006. http://www.lbl.gov/Science-Articles/Archive/sabl/2006/Jul/CellProcessorPotential.pdf.

[14] F. J. Seinstra, J. Maassen, R. V. van Nieuwpoort, N. Drost, T. van Kessel, B. van Werkhoven, J. Urbani, C. Jacobs, T. Kielmann, , and H. E. Bal, "Jungle computing: Distributed supercomputing beyond clusters, grids, and clouds," 2010.

[15] Wikipedia, "Berkeley open infrastructure for network computing." http://en.wikipedia.org/wiki/Berkeley_Open_Infrastructure_for_Network_Computing.

[16] Openwall, "Parallel and distributed processing with john the ripper." http://openwall.info/wiki/john/parallelization.

[17] Wikipedia, "Jungle computing." http://en.wikipedia.org/wiki/Jungle_computing.

[18] "Ibis official website." http://www.cs.vu.nl/ibis/.

[19] F. J. Seinstra, J. Maassen, R. V. van Nieuwpoort, N. Drost, T. van Kessel, B. van Werkhoven, J. Urbani, C. Jacobs, T. Kielmann, , and H. E. Bal, "Jungle computing: Distributed supercomputing beyond clusters, grids, and clouds." http://www.few.vu.nl/~jui200/papers/jungle.pdf.

[20] Ibis, "Ibis project page." http://www.cs.vu.nl/ibis/projects.html.

[21] N. I. of Standards and Technology, "The nist definition of cloud computing." http://docs.ismgcorp.com/files/external/Draft-SP-800-145_cloud-definition.pdf.

[22] I. Sommerville, "Distributed systems," 2004. http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/Presentations/PDF/ch12.pdf.

[23] A. B. Bondi, "Characteristics of scalability and their impact on performance," 2000.

[24] R. von Behren, J. Condit, and E. Brewer, "Why events are a bad idea (for high-concurrency servers)," in *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, (Berkeley, CA, USA), pp. 4–4, USENIX Association, 2003.

[25] J. Haas, "Definition: Modular programming." http://linux.about.com/cs/linux101/g/modularprogramm.htm.

[26] D. G. Feitelson, "On the scalability of centralized control," 2005.

[27] R. Steinmetz and K. Nahrstedt, *Multimedia Systems*. 2004.

[28] A. El-Sayed, V. Roca, and L. Mathy, "A survey of proposals for an alternative group communication service," 2003.

[29] "John the Ripper official website." http://www.openwall.com/john/.

[30] pentestmonkey.net, "John the Ripper hash formats." http://pentestmonkey.net/cheat-sheet/john-the-ripper-hash-formats.

[31] "Cain and Abel official website." http://www.oxid.it/cain.html.

[32] M. Montoro, "Cain and Abel hash formats." http://www.oxid.it/ca_um/topics/password_crackers.htm.

[33] "RainbowCrack official website." http://project-rainbowcrack.com/.

[34] "oclhashcat-plus official website." hashcat.net/oclhashcat-plus/.

[35] "IGHASHGPU official website." http://www.golubev.com/hashgpu.htm.

[36] "Extreme GPU Bruteforcer official website." http://www.insidepro.com/eng/egb.shtml.

[37] "Elcomsoft's products page." http://www.elcomsoft.com/products.html.

[38] IBM, "Three-tier architectures," 2005. http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=%2Fcom.ibm.websphere.express.doc%2Finfo%2Fexp%2Fae%2Fcovr_3-tier.html.