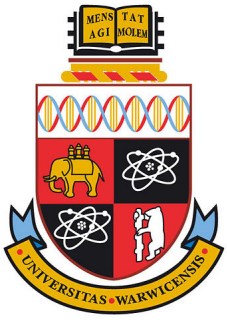
A Transparent Distributed Computation Toolkit

CS310 Computer Science Project
Final Report

Mikas Michelevičius



Supervisor: Dr. Adam Chester
Department of Computer Science
University of Warwick
2020-21

Abstract

The report discusses the design and implementation of distributed computation system, using the Python programming language and its built-in libraries for communication networks such as *socket* and *FTP*. The document also looks into the development of a web user interface using *Django* framework. The system accepts multiple job submissions from the user and distributes these jobs among the cluster of connected nodes for execution. The developed application contains features such as node selection for execution, queue for submitted jobs and scheduling algorithm. The report researches the progress of distributed systems as well as focuses on researching and comparing the similar existing systems. The final product is a functional prototype system of distributed computation system accepting user submissions containing command-line interface and web-based user interface.

Contents

Chapter 1

Introduction

Because of the recent changes and improvements in the information technology field such as decreased prices of microprocessors, speed improvements of the networks, terms computer cluster and distributed computing emerged. Together with that, the demand for efficiency, power and supercomputers is growing. In today's data-driven world, knowing that over 90% of the world's data was generated in the last several years [1], High-performance machines are being used widely in any field such as medicine, finance or especially computational science area working on quantum mechanics or weather forecasting.

With that being said, the development of computer clusters and distributed computation significantly improves the performance, scalability of what can be achieved on a single machine and is more cost-efficient compared to a single computer. A computer cluster is a number of computers interconnected using local area networks. Each node of the computer cluster performs the same task with the software controlling and scheduling it. Distributed computing allows machine in the cluster to share their resources with other machines. It lets one computer run an application and use CPU power, memory and/or storage of other computers. When working together, all the connected nodes can be seen as a single powerful computer, it can also enhance fault-tolerance of the system and reach many users.

A system using resources of different machines and improving performance is a single thing, however, another important aspect of distributed systems to pay attention to is scheduling. A scheduler is responsible for decision making when distributing jobs incoming to the system. It determines the order of the execution of jobs in the queue, selects the node for execution depending on required and available resources. Knowing the metrics of the machines in the distributed system and with the user providing relevant information about the job being submitted, the scheduler will aim to optimize the overall performance, it will utilize the resources of the cluster and improve the efficiency of the distributed system as much as possible.

1.1 Project Objectives

First of all, the primary goal of the project is to implement a prototype system which is a transparent distributed computation toolkit. The aim is to design and implement an API where the user is able of submitting multiple jobs to the system for execution. The system by itself consists of multiple interconnected nodes that are being managed by the server. The target is to develop such an application which, given some jobs, will queue and schedule them, depending on the user provided information and the history of user's system usage, and send the jobs for execution on different worker nodes and return the results once the execution of each job is completed, also providing some analytics on running and finished jobs. The goal is to develop both, command-line based and web-based user interface for the program.

Moreover, similar applications already exist, thus, a thorough analysis will be performed on few existing tools such as HTCondor - a high-throughput computing software framework and Ballista-Compute - a distributed computing platform (more details in section 2.1). Different aspects of each application will be compared such as the architecture of both programs, tools used for development and how the systems are used. The objectives include the understanding of the systems being analysed and how one or another aspect of the existing frameworks could help to develop a separate distribution system for high-performance computing.

In the end, since there exists a number of applications for distributed computing it will make it easy to evaluate the functionality of the distributed computation system that is being built against the desired outcome of existing tools. The project will also be evaluated against the initial requirements, the system will be tested using different methods with the results being analysed to create an overall view of the project.

1.2 Project Requirements

As it was stated earlier in section 1.1, the primary goal of the project is to develop a prototype system for the distributed computation platform. Having completed the research phase it can be stated that such application is considered to be a complex software system. Having this in mind, it is important to follow the best development practices. One of the standard procedures in software development is to explicitly define the project requirements so that during the development stage each feature could be easily tracked. Therefore, the section will present the core requirements of the project.

Following the commonly used software engineering principles, the requirements are split into two groups of functional and non-functional requirements where functional requirements describe *what* the final version of the software should do and non-functional requirements specify *how* the program should perform a certain task. Moreover, each requirement was prioritised into one of the four categories with respect to the MoSCoW method. The MoSCoW abbreviation stands for four classes for a requirement in the system that are: *Must* have, *Should* have, *Could* have and *Will* not have right now [3]. Such classification allows to determine the minimum success criteria and focus on the critical features of the software, helping to stay on track during the system development phase.

1.2.1 Functional Requirements

1. Job submission to the system. User must be able to submit one or more tasks to the system for execution. (M)
2. The system must return the results to the user after the job is completed. Provide an error log to the user if the execution was unsuccessful. (M)
3. The system given multiple jobs must execute them concurrently on different machines in the computer cluster. (M)
4. The system must, if all the connected nodes are busy, take the submitted job and add it to the queue until one of the nodes become available. Then the job must be sent for execution. (M)
5. The system must provide user with the details on the available machines in the cluster as well as provide the user with information on currently submitted running and queued tasks. (M)
6. The system must execute a submitted task with the dataset provided on one of the machines in the computer cluster and provide the user with the results of the execution. (M)
7. The application, providing all the specified information and features, must be available to use from the command-line interface. (M)
8. User should be able to select the worker node depending on the metrics of each machine in the distributed system. The system should assign a worker node by itself if none was specified by the user. (S)
9. The system should notify the user once the job is completed by sending an email notification to the address specified within the submission file. (S)
10. The submission of a job should be done by providing the system with a submission file that contains essential information about the job being submitted. Such as executable name, dataset name, etc. (S)
11. The system could state the time of the execution of the task. (C)
12. User could be able to log in and log out of the system. (C)

1.2.2 Non-Functional Requirements

1. The system should have a web-based interface where submission of tasks is possible and details of each machine in the cluster is displayed as well as the results returned after job execution. (S)
2. The system should be easy to use. If using command line interface, each specific action should take no more than one command line input. If using the web user interface, all features should be accessed by no more than 2 clicks away at a time. (S)
3. The project should have a user guide containing instructions and essential information on the system's usage. (S)
4. The code of the system should be maintained and well documented. (S)
5. The system could be extendable to add new computational problems the system can execute. (C)

Chapter 2

Background Research

2.1 Existing Systems

2.1.1 HTCondor

2.1.2 Ballista

Chapter 3

System Design

The chapter discusses and describes the design of the distributed computation system carried out after completion of initial background research phase of the project. The System Design chapter includes the system architecture section describing every component of the developed application as well as reviewing both command-line interface and web-based interface usage of the system. It also covers the core tools used to build the system.

3.1 System Architecture

The core area of design that is required for the project is the system-level architecture of the distributed system. For the project, the main concerns for the developing system are concurrency - how well can multiple machines execute jobs at the same time, scalability - can the performance of distributed system be enhanced by extending it with a greater number of machines, transparency - can the system make invisible the distribution aspects to the user, providing centralized view of the application. Another important concern is whether such architecture can be designed and implemented in a specified period of time. Taking into account these concerns, the Server-Client Model for the project is used since this model is more simple to design and implement compared to other architectures, has multiple clients (in this case - separate machines) that can run and execute jobs independently of others, new clients can be connected to the server to increase scalability and server is designed as a centralized system managing all clients and jobs which provides transparency [5].

Three base system components of such architecture are - Server, Client and Communication Networks. In this model, the server provides services that can be initiated by clients requesting them. In the project system, there exists one control client that is the entry, the user part of the application. Within this client, the user can submit jobs to the distributed system by requesting that from the server and get back the results after the jobs are executed. Then there are multiple work clients that are being managed by the server and once a job is submitted from the control client, the server requests the work client to execute the specified job which after the execution returns the result to the server. The server also acts as a client what makes the architecture multitiered (Figure 3.1). Clients and server communicate with each other using a request-response pattern. A type of inter-process communication where the client sends a request to the server, and the server returns a response. The server can receive multiple jobs for execution at the same time but the distributed system can only handle a limited number of tasks. To manage this situation, the server depends on scheduler which prioritizes and queues the job submissions from the client to maximize availability and enhance the overall performance of the system for the user.

3.1.1 System Components

Having the core system components identified that are Server, Client and Communication Network, the overall system architecture model with the remaining components can be defined. The Figure 3.2 displayed below provides a visual representation of the components in the distributed computation system that is being developed. As mentioned earlier, the Clients in the system are separated into two groups, one being a Control Node, which is a user interface client and multiple Worker Nodes whose task is to perform execution on the submitted jobs.

The server part of the application consists of two server components, one is a Server Node in the Server-Client Model, another one is an FTP Server. FTP Server (File Transfer Protocol) is a part of the Server Node and is

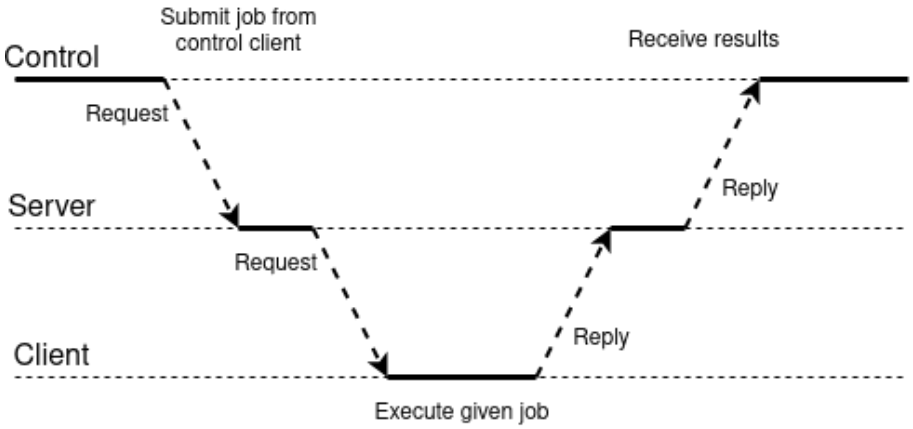


Figure 3.1: 3-tier server-client architecture

responsible for performing file transfers between the Control Node and Server Node, Worker Nodes and Server Node. When the user submits a job, all the required files have to be transferred to the server, in case all the Worker Nodes are busy and the user disconnects from the system, the server has the essential files stored and once a Worker Node becomes available, the files can be transferred there to be executed.

The arrows between the server components and Control Node or Worker Nodes stands for the Communication Network. As it can be seen in the Figure 3.2, the Control Node first has to request the Server Node to execute a job, the server from its part has to request one of the Worker Nodes to perform the task. The response with the completed task is sent to the Control Node via the server component.

The server is also connected to a database where it stores a piece of essential information about the system. For the prototype system, the database will contain a small amount of information such as the unique IDs for each user, user email for the notification, the unique task ID count. It also contains information on each user's system usage time as it will be used for queueing and scheduling of submitted jobs. The database can be extended as suitable for new features added to the system.

The last component of the system architecture that is visible in the Figure 3.2 is a web component. The web user interface is designed to provide the same functionality as the command line interface, therefore, the website is integrated with the Control Node program where the Control Node can be run either from the command line or by using the web user interface.

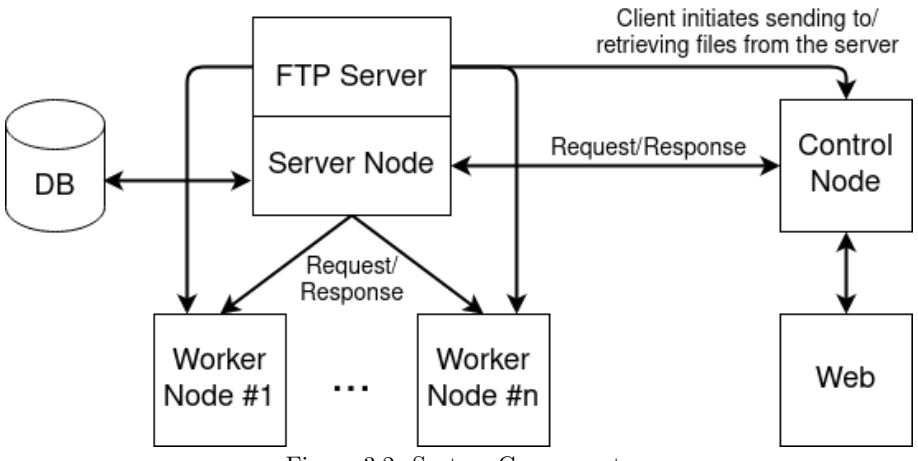


Figure 3.2: System Components

3.2 System User Interface

The user interface is probably the last fundamental component in the system to be fully implemented and prepared for usage as it does not perform any critical system functionality. On the other hand, it is important to define how the user might interact with the system in the early stages of the development process as it will be easier to define the internal components of the system architecture knowing how they will be accessed and used by the user.

3.2.1 Command Line Interface Usage

The project initially targets to implement the distributed computation system containing the command line interface usage of the software as most of the similar systems reviewed in section 2 contain such feature. Having outlined the functional and non-functional requirements of the system and targeting implementing a command line interface, the control panel for the system is defined. Having in mind that the system will also contain web interface usage, the command line interface is designed as follows. When a user connects to the system, it first receives a description of a command which will enable the command-line usage of the application:

```
To enable command-line usage - enter enable
```

Once the *enable* command is entered into the system, the complete control panel is displayed to the user. The command-line interface is designed in an easy and intuitive manner, containing very few commands, that will provide full functionality of the system for the user. Entering one of the first three commands will display the requested results in the command line. Entering the job submission command will first display the user the confirmation message that the system was successfully submitted together with its unique task ID. After the job completion, the system will display a message containing the name of the directory, where the results of the finished task are saved. The system will also send an email notification to the user after the job completion.

```
Connected clients - enter clients
Statistics of clients - enter statistics
Queue and running tasks - enter JOB-status
Task submission - enter SUBMIT filename.txt
To disable command-line usage - enter disable
```

The simplicity of the user interface has a major role in developing attractive software for the user. It is known that to develop a system to be usable on a daily basis it is important to keep the UI simple [2]. Designing such a minimal user interface provides ease of use and at the same time, it fully covers the functionality of the relatively complicated system.

Submission File

When using the command-line user interface of the application, user has to create a submission file to submit a job for execution. As discussed in section 2.1.1 reviewing HTCondor software, the submission file of the existing application was used as an example of how the submission file of the system being developed could look like. Following the existing example, the submission file for the system was defined. The submission file is a simple *.txt* file containing at most four lines that specify the relevant parameters. The first line, which is required at all times when submitting a job is a line specifying the name of the executable file that is being submitted. The name of the file has to be entered after the equality sign (=) following the *executable* variable. The remaining three attributes are specified in the same way as the first one, however, all three remaining attributes are optional and the lines can be avoided if not applicable. If the job to be executed does not contain a data file, the second line will be removed. If the user has no priorities when selecting the Worker Node, the line specifying the client unique ID can also be removed.

```
executable = executable.py
data = data.csv
email = your.email@gmail.com
client = 2
```

3.2.2 Web Interface Usage

As some people are not too comfortable with using different applications via command line user interface, it is vital to design a web interface for the system which provides the same functionality as the command line usage, and one

could say is more intuitive and easier to use. As defined in the Non-Functional Requirements in section1.2.2, all the features of the system should be accessed by no more than 2 clicks away at any time if using the web interface. Having this in mind, the UI component tree is first defined as adjacent components in the website in Figure 3.3. A node in the tree stands for a single component, a single HTML page in the website. The desired outcome of the web user interface is to be able to access every feature of the distributed computation system handily at any time.

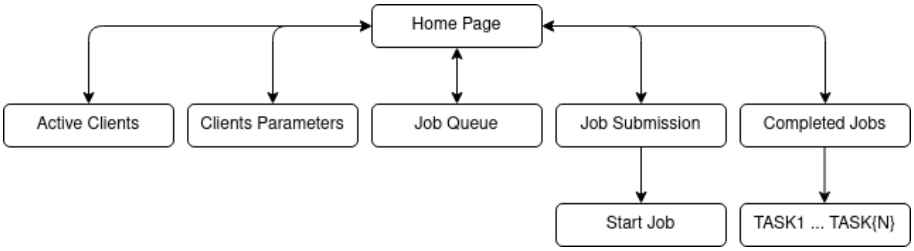


Figure 3.3: Web UI Component Tree

Once logged into the system, the user will land on the Home Page of the website (see Figure 3.4). From there on, just like using the command-line interface, we want to be able to access any system feature it contains. For that reason, the navigation bar of the landing page is designed. To be precise, the header bar will be part of the base HTML of the whole system so that each functionality can be accessed from every other component in the system.

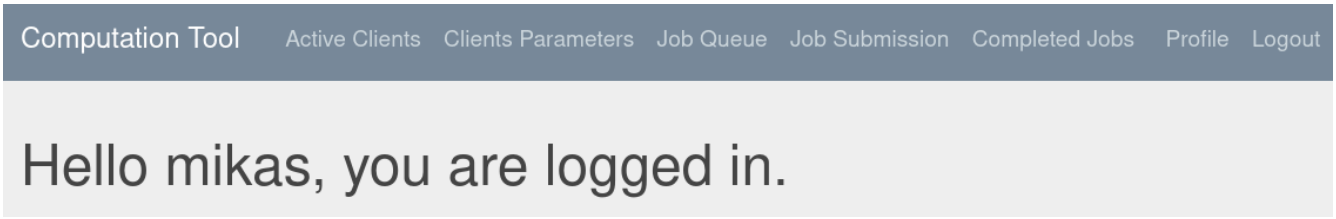
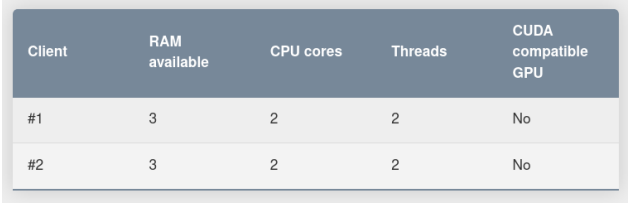


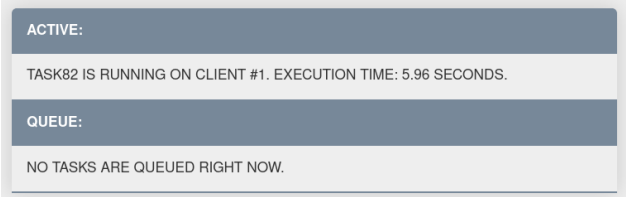
Figure 3.4: Home Page Component

Like in the command line interface usage, user can request for the list of connected clients in Active Clients (Figure 3.5(a)), list of connected clients and their respective metrics in Clients Parameters (Figure 3.5(b)) and see the current queue and running tasks in Job Queue (Figure 3.5(c)). The mentioned components of the web user interface provide the exact same information as given within the command-line interface, however, it is displayed in a more engaging way by using tables with a design.

(a) Active Clients



(b) Clients Parameters



(c) Job Queue

Figure 3.5: System Features

The major difference between using the command line interface and the web interface is the job submission part

of the software. Using the system from the command line requires writing a submission file by yourself. However, when using the web interface Job Submission component, the system will generate the submission file by itself, when the user only has to upload the files from his machine and fill in the required inputs. When the Job Submission component is accessed, the user receives several input fields (Figure 3.6) that have to be filled, including executable file, dataset, email for notification and select a Worker Node (can also be left with default input, then the system will assign a Worker Node randomly). Once the fields are filled, the user has to submit a job by pressing a Start Job button. This way of submission is definitely a more intuitive approach.

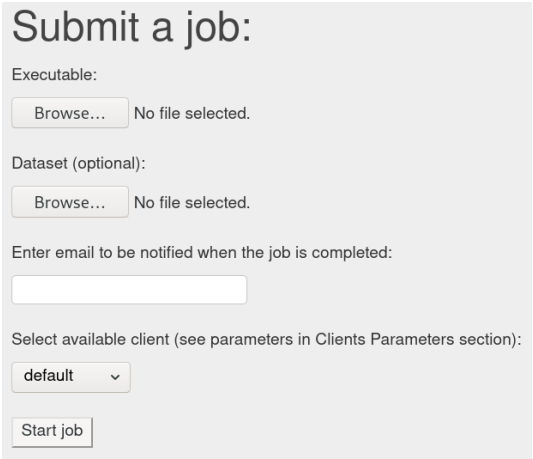
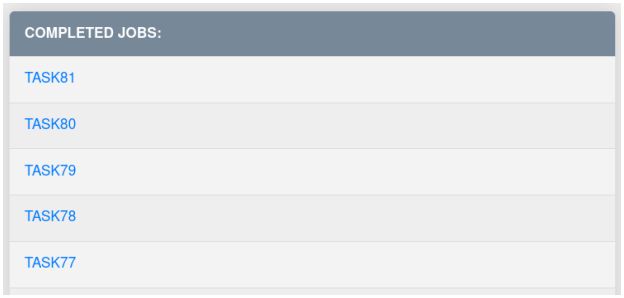
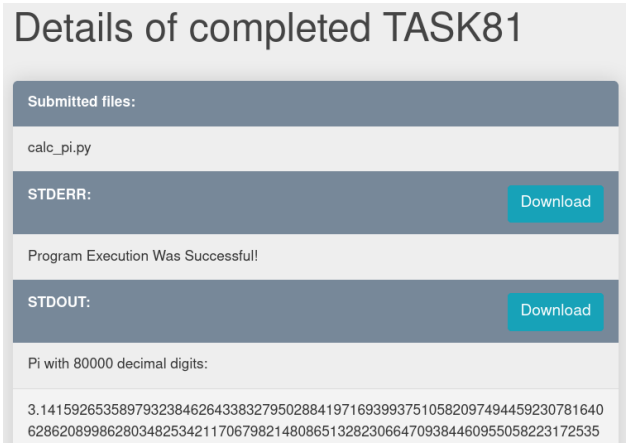


Figure 3.6: Job Submission

Another addition to the web interface of the system is that it lists all the user's tasks that were completed (Figure 3.7(a)). Selecting one of the completed tasks will land the user on another page which will provide the user with the results of the task completion (Figure 3.7(b)). The *stderr* and *stdout* of the job execution will be displayed, together with an option of downloading these files into a different directory in the machine. However, the same information can be still accessed in the task's directory in the machine.



(a) List of Completed



(b) Task Results

Figure 3.7: Completed Jobs

In the same way, as for command-line interface usage, the web interface was also designed as minimal and simple as possible. Avoiding unnecessary features, only containing the required functionality which can be accessed from any page at any time. The better one cannot be identified as it depends on the user's preferences, however, it is a good practice designing both types of user interfaces as it provides the user with the option of selecting one or the other.

3.3 Tools Used

3.3.1 Python Language

To build the distributed computation system Python programming language was used. A general purpose high-level programming language supporting multiple paradigms, primarily object-oriented programming, provides data structures that are highly efficient and at the same time simple to use. An elegant syntax of the language, together with an interpreter and dynamic typing makes it a great choice for rapid development and scripting of different applications, in this case, including the distributed systems programming. Due to the great community of Python, the programming language has a wide range of libraries that will accelerate the development process, there are also many frameworks available for python language, for example, Django, which is used for the web-based user interface of the project (more in section 3.3.5). Python's standard distribution includes a number of modules for internet protocols and network communication programming, one of them being *socket* module which is used for socket programming within the system [6; 7]. The communication is very easy and intuitive using Python, the following code sends a message using Python's socket communication:

```
import socket

HOST = 'node_addr'
PORT = 5002
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST,PORT))
s.send(str.encode("message_text"))
```

Considering the ease of use of programming language, developer productivity, prototyping and portability, one could say that Python is the best choice for a project such as a prototype of distributed computation system [8].

3.3.2 Python socket

Another area of research was the tools to be used for Communication Networks between server and clients. The concept of networking stands for connecting two or more separate devices in the computer cluster so that they could share resources between them. As mentioned above, Python's standard library contains a *socket* module used for networking and interprocess communication. The system uses TCP sockets for nodes communication where socket stands for IP and port of the network endpoint. The Transmission Control Protocol (TCP) is reliable and has in-order delivery of data, thus, there is a certainty that the packets sent across the network will not be lost nor the data arrived will be out-of-order.

Python socket contains several primary API methods that are required to set up the connection and perform communication between two endpoints in the network, the functions are socket(), bind(), listen(), accept(), connect(), send(), recv() and close(). The figure below provides a visual representation of the data flow and socket API calls sequence in the network (see Fig 3.8). The server creates a listening socket using socket(), bind(), listen() and accept() API calls. Now the server listens for the clients and when the client connects, the server calls accept() to establish a connection. In the meantime, the client calls a connect() method to initiate a three-way handshake. Now the handshake ensures that both endpoints are reachable in the connection. The middle part of the sequence is a section where the messages and data are being exchanged between server and client using send() and recv() API calls. After the desired communication is performed, both sockets are closed.

3.3.3 Python FTP and SMTP

Few other tools were required to build the system, to be exact, FTP which stands for File Transfer Protocol and SMTP short for Simple Mail Transfer Protocol. While TCP is a generic networking protocol running on the IP layer in the networking stack and it is used for reliable communication between the nodes in the network, File Transfer Protocol has to be used when an actual file has to be transferred from server to client or vice versa. Once again the Python's standard library is of use, containing *ftplib* module where the client side of the File Transfer Protocol is implemented using the following library [9]. The library contains multiple core methods used to connect to the server, navigate through the directories, fetch files from, send files to and delete files from the server. These function calls happen on the client side of the application.

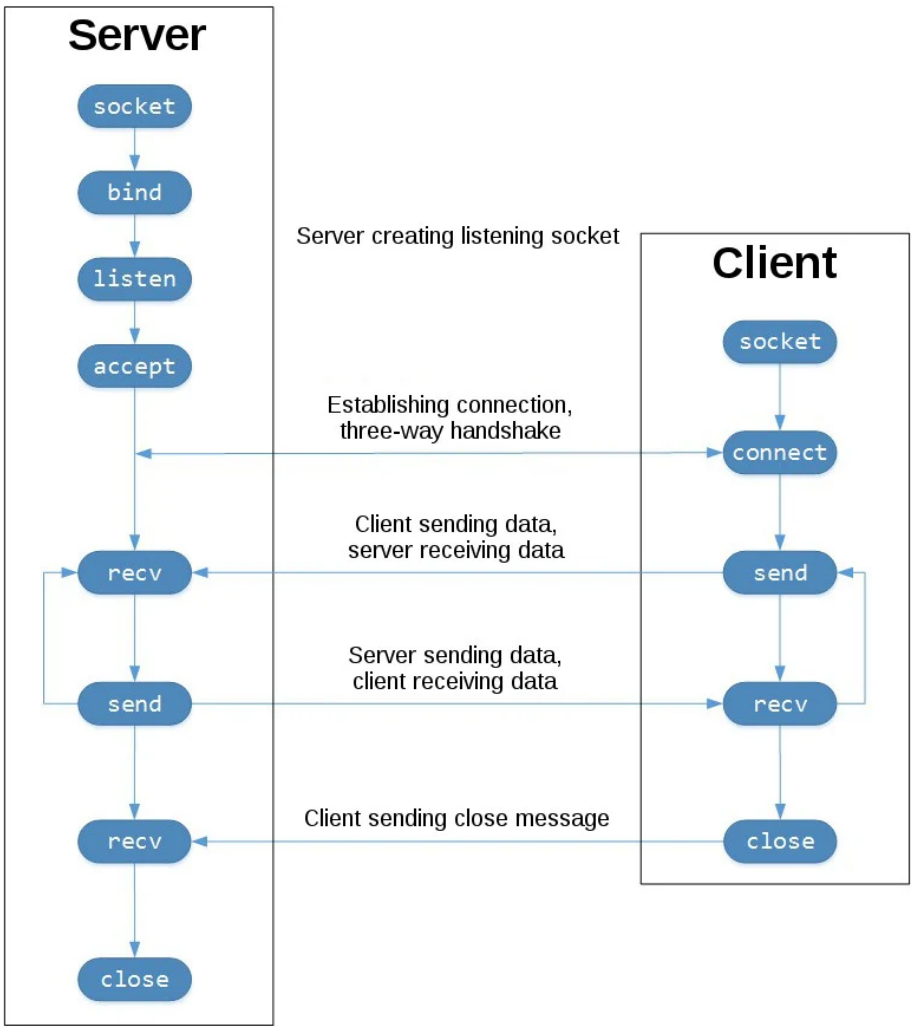


Figure 3.8: Socket API calls and data flow

Source:*https://commons.wikimedia.org/wiki/File:InternetSocketBasicDiagram_zhtw.png*

The system will also have to provide the user with notifications on running or completed jobs. Such a feature can be approached by using another existing Python standard module called *smtplib* [10]. SMTP session object is defined when using the library, which is used to send an email to any machine using an SMTP listener daemon. The module contains functions that will set up a local SMTP server, establish a secure connection with an email server and will send a personalized email, depending on the job outcome, to the provided address. While some hosts are struggling with delivering the emails, the SMTP is a great choice providing a secure and safe delivery, keeping the email records private and under the developer's control.

3.3.4 SQLite

Building a prototype system, SQLite is used which is a Relational Database Management System type database. The database was selected because of its ease of use, it is portable and reliable, it works great for the small amount of data. Thus, the server in the system architecture stores all the relevant information in the SQLite database. It is also said to be an appropriate use for the Server-side database [4]. However, the tool contains shortcomings, including that it should not be used in production since it does not support concurrency and it cannot scale. However, it is a practical fit to build a prototype system.

3.3.5 Django Framework

After doing research it was decided to use a Python based high-level web framework to build a web-based user interface for the system. The key aspect of the framework is that Django's main goal is to simplify the work to be done by developers. Building an application within the framework provides an easy way of testing and running the web app. Great support for both back end and front end as well as easy to use database API supporting most popular databases. Since the framework is Python-based, the extensive libraries will accelerate the development process. Django framework is based on MVT (Model-View-Template) architecture (see Fig 3.9). The communication between the server and the client happens using HTTP request and response objects. The HTTP objects with the contents are being returned from the app's views. In the views the data will be formatted, the Django models will be used to retrieve data from the database and the HTTP response object will be sent to the client.

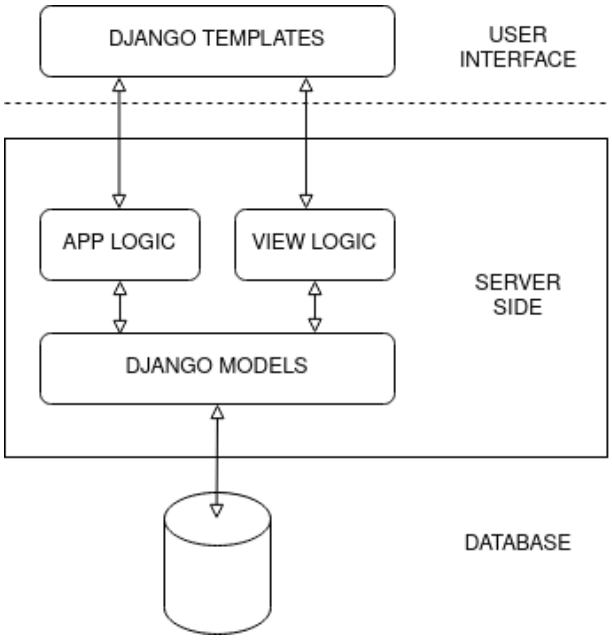


Figure 3.9: Django architecture

Chapter 4

Implementation

In the previous chapters, having completed the planning, background research and designing stages, the fundamental components of the distributed computation system were identified, therefore, the implementation part of the project can now be covered. The chapter will cover the software engineering and development practices implementing all of the earlier mentioned components of the system, following the selected development methodology, using tools that were selected during the research and design phases. In this chapter the development of low-level components of the system architecture will be thoroughly explained as it is important to understand how different components interact with each other in the system, how the data structures, algorithms and selected tools are utilised to make the application perform initially defined tasks. Separate components of the system architecture will be described step by step as it was implemented following the incremental development methodology during the project development. Breaking down project development into components provides logically consistent development and reasonable testing for each part.

4.1 Client-Server Model

To set up the base of the project simple client and server network is created. The server and client code will be located in two separate Python files. The server file will be later expanded to perform as a Server Node for the whole system, performing scheduling, queueing and distribution tasks, while the client connection code will be later used to implement a Control Node and the Worker Nodes of the system. The server side is first initialized, creating Python *socket* server object, specifying internet address family and the protocol to be used for message passing in the network, IPv4 (argument *socket.AF_INET*) and TCP (argument *socket.SOCK_STREAM*) respectively (line 1, Listing 4.1). Using API call bind(), the socket is associated with the specified network interface and port number. During the project development, the standard loopback interface address, localhost - '127.0.0.1' is used as a network interface. The server enters a listening state by calling listen() function and it is now able to accept an incoming client connection and establish a network. Once the connection is established, recv() reads whatever data the client sends and by calling send() function, can send the message back to the client (lines 5-8, Listing 4.1).

Listing 4.1: Server socket

```
1 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
2 s.bind((HOST, PORT))
3 s.listen()
4 conn, addr = s.accept()
5 while True:
6     data = conn.recv(4096)
7     if data:
8         conn.send(str.encode("Message back"))
```

On the other side client's *socket* object is created in the same way. Connection from the client's side happens calling connect() function specifying server host name and port number. At this point communication between server and client can begin, the message can be sent and received from client-side (lines 1-4, Listing 4.2).

Listing 4.2: Client socket

```
1 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
2 s.connect((HOST, PORT))
3 s.send(str.encode("Message from client"))
4 data = s.recv(4096)
```

In the system, the Server Node and both Control Node and Worker Nodes contain an infinite while loop constantly listening and waiting for new messages from another endpoint in the network. Two endpoints exchange string messages, however, Python *socket* sends data in bytes format, for this reason, str.encode() and data.decode() methods are used.

4.1.1 Control Node

In the multi-connection client and server, there is one dedicated Control Node in the system that provides the user interface of the program. When this client connects to the server, the server registers it as being connected and available. If the user connects for the first time, the server provides the user back with the unique ID, which is saved in the database within the Control Node, otherwise, the Control Node already contains its unique ID and provides the server with this information during connection. Therefore, each Control Node connected to the system is assigned with its unique identifier which is saved externally.

The Control Node, when connected, provides the user with a control panel, where the user can select one of the several features. These features are:

1. Get information about currently connected available and busy clients.
2. Get metrics of each connected client. This includes available RAMs, the number of CPUs and threads, whether a GPU is available.
3. Get information about currently running or queued jobs.
4. Submit a new job to the system.

The system contains an infinite loop, where the program waits for one of the two following - either a user input with a request for one of the specified system tasks or a message from the server.

User request

If the control client receives one of the first three requests from the user, the program will forward that request to the server of the system via an established socket connection. Depending on the message passed, the requested information will be generated in the server and returned as a string message back to the Control Node within a socket connection. Control Node side of the application handles differently the request to submit a new job for execution. Once the program receives a request to submit a job, the user provides the system with the submission file containing essential information about the job together with the names of the required files. Function *send_file()* is now called, giving a submission file name as an argument. In this function, the connection between the FTP server in the Server Node and the Control Node will be established. The executable file of the submitted job, the dataset and the submission file itself will be moved to the Server Node via the FTP connection using its API *storbinary()* call. Once the files are transferred, the FTP connection will be closed and the submitted files will be temporarily removed using the Python *os* library and its *remove()* function.

Requests from the server

At the same time, the Control Node is constantly listening for the incoming messages from the Server Node endpoint of the socket connection. Most of the messages coming from the server will contain the information that was requested from the user itself, for example, a list of currently available and busy nodes, metrics of each connected client and information about running and queued jobs. Some messages will contain information such as confirmation of successful user connection to the server, confirmation of successfully submitted job for execution, such information will be handled and displayed to the user. For example, once the user submits a job, the server generates a unique ID for the submitted task and will return a confirmation message containing the unique task ID number.

```
submission_file.txt SENT FOR EXECUTION AS TASK32
```

This information will be delivered to the user, which now receives the unique id, in this example, *TASK32*, which will be used to provide further information about the submitted job.

The server can also send a request with information that the job execution is completed, providing a unique task ID of the job that was finished. The Control Node will now call the *get_results()* function with the unique task ID as an argument. Once again the FTP connection between the server and the client is established but this time the Control Node will retrieve all the relevant files from the server after execution using FTP API function call *retrbinary()*. The retrieved files, containing standard output, standard error of the job execution will be presented to the user in the Control Node side of the application. The program will create a directory by the name of the unique job ID where all the retrieved files from the server will be stored.

4.1.2 Worker Node

Another part of the distributed system architecture is a Worker Node of the system. In the multi-connection client-server distributed application structure, the Worker Node stands for a client. Assuming a fully functioning distributed computing system, such system will have one or more Worker Nodes running at the same time that is connected to the Server Node in the network. The Worker Node is dedicated to performing one of the submitted tasks that were assigned by the server and run the computations by using the resources of the machine the Worker Node is running on and return the results to the server when the job is finished.

In the developed system, Worker Node contains a separate Python file, which will be run to connect to the distributed system. When the code is run, the client will connect to the server using a Python *socket* connection. At the time of the connection, Worker Node will send the Server Node its metrics which will be provided to the user when requested. This happens by calling *return_stats()* function in the client program which will retrieve the machine metrics by reading files of a Linux machine such as */proc/meminfo* and */proc/cpuinfo*. It will also get the information on whether the machine contains a CUDA compatible GPU. The GPU information will be collected by using TensorFlow config system call *list_physical_devices()*, which will return a list of available GPU devices on the machine. All this information will be formatted and sent to the server after connection, using the *socket* function *send()*.

The Worker Node program, like other socket endpoints in our system, will be constantly listening to the messages from the server. As mentioned earlier, the Worker Node's main responsibility is to execute the submitted task and returning the results back to the server. With that being said, the Worker Node will receive a message from the server containing the unique task ID, requesting the job execution for the specified task. Once the request is received, the program calls *execute_task()* function providing the unique task ID as an argument to the function. The connection between the FTP server in the Server Node and Worker Node is established and all the relevant files of the submitted task are being retrieved from the server (including executable, dataset). The Worker Node then uses a Python *subprocess* module to invoke a subprocess and calls *Popen* system call on the uploaded executable which executes a program in a new process on the Worker Node machine. *stdout* and *stderr* are obtained from the function call once the program is finished running.

The FTP connection will once again be established between the Server Node and the Worker Node. All the relevant information collected during the job execution will be transferred to the server via an FTP connection. The Worker Node will also send a message through the *socket* connection responding to the server about the job completion. The Worker Node will be marked by the server as available for another job and the Worker Node will receive another request to execute a task once a new job is assigned by the Server Node.

4.1.3 Server Node

The most important component in the system's architecture and in the client-server model is the Server Node of the application. The server is basically the logic of the distributed computation system, which has multiple responsibilities within the system. Some of the responsibilities are to manage and distribute submitted jobs, handle the traffic of the job submissions - queue and schedule them, process other requests from the clients, store and protect the data. The Server Node of a distributed system has to be developed carefully since with every decision there is a trade-off. The greatest difficulty of the distributed system is its complexity - the more Worker Nodes are connected, the more data and messages are being passed which, if not handled correctly, will result in network and

communication failure, management overload. Thus, the server component of the system was the most challenging part of the project to design and implement.

Connection management

First of all, the server has to keep track of the connected clients. This sounds somewhat intuitive, however, the system contains two types of clients that are Control Nodes and Worker Nodes that have to be separated. To tackle this problem, the server contains several data structures that are Python dictionaries. One for Control Nodes, another one for Worker Node connections. When the Control Node is connected to the server it will send a message within the *socket* connection containing information of its unique ID. The server then, using its unique ID as a key will save the client's *socket* connection object as a value in the dictionary, thus, the dictionary data structure contains all the connected Control Nodes with ID/connection object tuples as key/value pairs. Now whenever the server needs to send a message to one of the Control Nodes, it will access its connection object by using its unique ID.

In a similar way, the server assigns a unique identifier for each Worker Node connected to the system and saves its unique ID and *socket* connection object in a separate Python dictionary. The Server Node program also contains another data structure used to save information of which Worker Node is currently executing which task so that the system would not try to assign a task for the execution for a Worker Node if one is currently executing another task. Thus, every time a task is sent for execution to a Worker Node, the data structure containing currently busy connections is updated with a new node, and when the job is done, the Worker Node will be removed from the list of busy nodes.

Control Node requests

Most of the messages received by the server are coming from the Control Nodes. The Control Node will request one of the four features that the system contains, that were listed in section4.1.1. Every time the Worker Node connects to the server it sends its metrics to the server which again stores the details of each connected machine in another Python dictionary. Thus, when Control Node requests details of connected machines, the server will send a response with the information that was stored in the dictionary.

If Control Node requests information of the currently connected and busy nodes, the server will make use of earlier mentioned data structures that store the list of connected Worker Nodes and busy nodes and will send a response to the Control Node with such information.

The server also contains data structures that store the tasks that are currently being executed or the tasks that are waiting in the queue. Then, when Control Node requests for information of running and queued jobs, the information from these data structures will be formatted and sent back to the Control Node.

When the Control Node sends a request to execute a job, the server will first assign the job with its unique ID (which is basically an increment of the submitted tasks to the system), then *send_to_execute()* function is called with arguments containing the submission file name and the task ID. Then the confirmation message containing the assigned unique task ID is sent back to the Control Node. In the *send_to_execute()* function the program will create a directory for the submitted task where all the relevant files (submission file, executable, dataset, results) will be stored, this way, if there are no Worker Nodes available, the system will store submitted tasks in the server until they can be executed. The system will also use a Python dictionary to map the submitted task ID to the Control Node which submitted the task and the information is later used to return the results to the correct user. If the Worker Node was selected in the submission file, this information will be used to select the specified Worker Node connection object, otherwise, Worker Node will be selected randomly and the request to execute a task will be sent to the selected Worker Node. If no Worker Nodes are available, the *enqueue()* function is called where the task is put to queue and scheduled for execution (will be covered in section 4.2).

Also, when the Control Node from which a job was submitted earlier connects to the server it will request to return the results of the submitted job by calling *get_waiting_results()* function. If the job has already been executed, the server will send the Control Node the information of which files to retrieve from the server, that contains the results of the executed job.

Worker Node requests

The server will first receive a message from the Worker Node when connected to the system, containing details about the machine as already mentioned earlier. The metrics of the machine will be stored in another Python dictionary in the program.

However, the only other message coming from the Worker Node to the server is a notification that the job that was submitted is now executed. Once such a message is received by the server, the *send_results()* function is called passing unique task ID and submission file name as the arguments. The function then retrieves the user ID and its connection object depending on the task ID. The function *send_email()* is called which notifies the user of job completion. If the Control Node that submitted this job is currently connected, the notification of job completion is sent to the Control Node so that it can retrieve the relevant files from the system. If the user is not connected, the task is added to the list of tasks that are completed until the Worker Node connects to the server and the results can be transferred to the user. Once the mentioned tasks are performed, the program calls function *execute_queued()* which, will assign a task waiting in the queue to the newly available Worker Node, if there are tasks waiting in the queue.

4.2 Queue and Scheduling

Another important aspect of the server in the distributed system to pay attention to is scheduling. A scheduler is responsible for decision making when distributing jobs incoming to the system. It determines the order of the execution of jobs in the queue, selects the node for the execution depending on the required and available resources. In general, it utilizes the resources of the cluster and improves the efficiency of a distributed system as much as possible.

In the earlier version of the software when the queue was first implemented, the system used a simple priority queue. When the job is submitted and there are no Worker Nodes available for the execution, the task would be added to the priority queue. Once a new Worker Node would be connected or a busy one would become available, the first submitted job from the queue would be removed and sent for execution.

Later on, in the development process, it was decided to implement a smarter way of distributing the submitted jobs, to develop a scheduler that will distribute jobs more equally from the user point of view. There was also an idea of implementing a way of submitting a job with the user specifying the desired resources of the potential Worker Node such as a number of CPUs, RAMs, GPUs and implementing a scheduler that would assign a Worker Node depending on the requested variables. However, due to the time constraints of the project, the selection of available Worker Node was implemented without specifying separate parameters. Yet, a smarter scheduling algorithm was implemented. In the final version of the software, a scheduler is implemented which will form a queue of submitted tasks depending on the user overall system usage time. The goal is to share the resources somewhat more equally between the users of the system, so that if one user submits many jobs for the execution, another user, who has submitted a single job would not have to wait all the time until all the jobs submitted by the first user are done executing. The system contains a database that is updated every time a job is completed in the Worker Node. The database contains a table of a user and the overall usage time of the system for the respective user ID. The server uses the Python *time* library and tracks the execution time of each job and updates the table with the number calculated after job execution. Each time when a job is finished, the queue is updated, depending on the values in the database. At the front of the queue will be a task submitted by the user which has the smallest usage time. The queue is also updated every time *enqueue()* function is called with a new task being added to the queue.

Such a scheduling method is far from being optimal, however, it is a great way, which is quite easily implemented, of distributing jobs among the cluster of nodes in the distributed system. It can definitely be improved in many ways as some of the scheduling methods were mentioned in section 2.

4.3 Web Interface

The web user interface of the system was implemented using the Django web framework. The web interface is an extension for the Control Node program, which is a command line user interface. It was developed to have the exact same functionality as the initial command line interface provides, only targeting to develop an alternative or

more intuitive usage for the application.

The framework groups its code into different files that handle different steps of the application. The website is waiting for the *HTTP* request from the web browser or another client in the system. Once the request is received, the application will redirect the request based on the URL and depending on the *POST* or *GET* data it will perform the required task and return an *HTTP* response object that will be displayed in an HTML template.

The sequence of how the requests are handled through the different files in the application are presented in Figure 4.1. Firstly, the application contains an URL file which basically is a mapper between the HTTP request and the appropriate view for the request. Whereas the view is simply a Python function is the core of the application. In the view file, when the HTTP request is received, the program will perform any required actions such as accessing the database, performing any required computations or rendering a template and many more. The view will then generate the HTTP response object and return it to the specified HTML template. Template in Django is a text file defining the layout and structure of an HTML page. The model in the Django framework maps to the database table in the system. With that being said, Django provides an API supporting the most popular databases for handling the data through the Python objects called models.

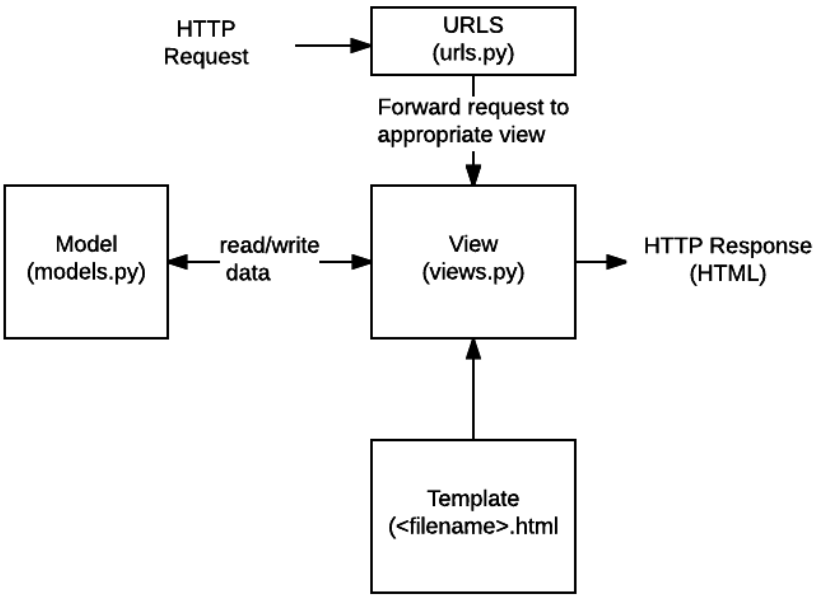


Figure 4.1: Django Requests Diagram
Source: *https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction*

4.3.1 Back-end

The back end of the application happens in the *views.py* file. From the browser, once any of the features in the system is selected (Home Page, Active Clients, Clients Parameters, etc.), an URL mapper will redirect the request to one of the respective functions in the views file. Once the request is received in the views function, the application will redirect the request to the Control Node of the distributed computation system. For example, if the Active Clients feature in the website is selected, the request will be redirected to the *clients()* function in the views file and the Server Node will be then requested for the following information. Once the information is returned from the Server Node, the views function will wrap the data containing the currently active Worker Nodes and return the HTTP response object to the Django template where the data will be displayed in the web browser for the user. Client Parameters and Job Queue request will be handled in a similar way.

When a user submits a job within the Job Submission page on the website, the *GET* method request is sent to the *submit()* function in the views file. The function then, depending on the provided inputs from the user, will create a submission file and will submit the file to the Control Node in the system. Whereas the Control Node will

then forward the request to the Server Node of the distributed system.

When the Completed Jobs page is accessed in the web user interface, the function *completed()* in the views file is called where the system will list all the directories of the completed jobs by using Python *os* library. When any of the jobs is selected, the request is sent to another *details()* function where it will wrap all the relevant information about the executed job and will send it within the HTTP response object to the template where the information will be displayed.

4.3.2 Front-end

The bootstrap styling library was used to develop a professional looking web user interface. The base HTML file was developed containing the navigation bar with the URL reference to any of the system features. Every other template for a different page in the website will extend the base HTML file, thus, every page in the system will contain a header bar. Apart from that, HTML files will make use of the data returned within the HTTP response object and display it using bootstrap styling tables. The Job Submission page in the system will collect all the user input data by using Django forms and will send the request containing the data once the user presses the Start Job button.

Chapter 5

Testing and Results

The chapter will go through the process of testing the application during the development process for different units of the application. Integration testing of the software will follow the unit testing, as well as the system testing, which will be performed to test the application as a whole.

5.1 Testing

As the core iterations of the development process was defined in section 6.1, using the incremental development methodology provided an opportunity to test each newly developed component of the system at each iteration before moving onto the next iteration. Since the system contains many different functions that are responsible for different tasks, during the development process, each function was tested to be sure it provides the desired functionality. However, integrating these multiple functions resulted in greater components of the system that were then tested, which are described below.

Client-Server Model

The initial stage of the development process was to develop a simple Client and Server Model as a base for the application by using Python *socket* library. Once the two separate components of client and server were created and a connection between them was established, to test whether the communication network works as it is supposed to, a simple Server and Client Echo application was developed. The created application tested whether the different endpoints in the network receives the messages sent from other components in the system. The application allowed to enter the input message from the client-side in the network and send it to the server. Once the server received a message, it would echo back a message notifying the client that the message sent was received. The established connection was functional and ready for further development.

Control Client Features

Once the Echo application was developed, the program was being extended by adding new features. The feature of the client allowing to enter the messages and send across the network was expanded to be used as a Control Client command-line interface of the application. Thus, the Control Client features of requesting server for the information about currently connected clients were being developed. The system features of returning the list of currently connected Worker Nodes and requesting information about the metrics of each connected node were tested. It was done by connecting a different number of client nodes, testing edge cases such as returning the information when no clients are currently connected or when there are many Worker Nodes connected to the system. The testing phase of these features helped to define the structure of the data being returned, establish the information that was essential to be provided to the user.

Job Submission

Before performing the job submission, the FTP server connection and the functions *send_file()* and *get_file()* in Control Node and Worker Node were tested. Testing these functions allowed to make sure that the file transfers between different components in the system are performed and that the submitted files will reach the wanted

destination where they will be executed and files containing the results will be returned back to the user. Once the mentioned functions and FTP connection were tested, the command line input usage was integrated to allow the user to specify what files to send for execution by using the submission file. The transferred files were then being executed on the Worker Node by using Python *subprocess*, collecting the *stdout* and *stderr* of the program that were returned back to the user. The job submission and execution was tested by submitting different types of Python executables. They were programs either containing a dataset and performing computations on it or some Python scripts with a long execution time. The programs containing errors were sent for execution to test whether the *stderr* is handled properly and returned back to the user.

Job Management

When the job submission feature was performing correctly on a single Worker Node, it was then attempted to distribute multiple jobs to multiple available Worker Nodes. When the first distribution feature was implemented, it was first tested by connecting two or more Worker Nodes to the system and submitting multiple jobs with a long execution time. This way the system was tested whether it can execute multiple jobs at the same time on different nodes. Having that done successfully, the queue and scheduling algorithms were being implemented. Since the algorithm schedules the submitted jobs based on the user system usage time which is stored in the database, to test whether the algorithm works properly, the database records were changed to creating multiple hypothetical users with their different system usage times. Multiple possible situations were tested knowing the expected outcome based on the database records and comparing it against the sequence of the system's scheduled jobs.

Web Interface

Having the software properly working by using the command-line interface, the web user interface was developed. Once the base of the website was created, the URLs and navigation of the website were first tested. All the buttons were tested, monitoring whether they redirect user to the appropriate page of the system, as well as login, register and logout functionalities, were tested. Once the website was integrated with the whole software, each feature that was functional from the command line interface was then tested by using the web user interface. The submission of multiple jobs was performed with multiple or no Worker Nodes connected. The information about the connected clients and submitted tasks was being requested.

5.2 Results

Testing of the application was a great part of the development of distributed computation system. At each iteration of the development process testing was a great tool to polish the functionality of the system since the testing provided an insight on the shortcomings of each feature and how can the system be extended or improved. The tests were created along the way, adjusting to the rapidly changing application and providing useful insights at each step. Having that in mind, the tests were created in a way that the application would fail, so that the features could be improved. However, if the same tests would be performed for the final version of the system, they would all pass.

Chapter 6

Project Management

For successful project execution, useful project management techniques were required. The chapter describes and justifies the choice of selected development methodologies used during the project development. It also explains how the time was managed and how different situations were handled, as well as mentioning the version control management of the project.

6.1 Methodology

As mentioned earlier in the report, to develop the project Incremental Software Development Methodology was chosen. One of the commonly occurring problems in software development is a trade-off between implementing the core requirements for a reliable system and delivering the software on time. Incremental development, compared to different development methodologies, allows handling both aspects very well [12]. The core of this methodology is the increment. Within each iteration, the next component of the system in the list of remaining tasks is challenged. Once developed, it is tested, reviewed and another increment is created. Using this methodology, even when timetable changes were encountered, did not cause any significant damage since it allows to adjust changes during the development. In parallel with each iteration of the system development, the research was being done from the beginning of the project till the end, meaning the ideas were likely to change. Again, due to the selected development approach, such changes were handled without any significant delay. Once the requirements were finalised the following main iterations of the project were defined:

1. Development of a simple client and server model as a base for the application.
2. Implementation of basic features of the system such as providing the user with information about the connected worker nodes.
3. Implementation of job submission feature of the system, returning the results after the execution.
4. Management of job submissions in the server - queue and scheduling algorithms.
5. Web user interface integration to the system.

Above mentioned core iterations consisted of multiple sub-tasks. To apply successfully the iterative approach for the project, regular weekly meetings with the supervisor had to be held. In such a way the core iterations and sub-tasks for each week were established, keeping the development focused on the key tasks of the project while strictly sticking to the plan.

6.2 Version Control

For version control management during the development of the project, Git was used. The tool allowed to revert to earlier development stages if some technical problems occurred that were hardly fixable. GitHub - a Git cloud-based platform was used, providing secure protection against losing the data as well as remote access to Git. Both the code and the report of the project were stored in a private Git repository on GitHub.

6.3 Time Management

Initially, the first steps taken when planning the project execution were outlying the main iterations of the development process and creating a Gantt diagram containing the main deliverables of the project. The very first diagram was defined and created for the Project Specification report at the beginning of the academic year at the end of 2020. However, due to the fact that the research phase was done after the Project Specification report was developed, the core development iterations of the project as well as the Gantt diagram have quickly changed. The more accurate version of the diagram was later presented in the Progress Report document.

The Gantt diagram provided a visual representation of the project development stages, as well as the multiple deliverable deadlines, were included, such as Progress Report, Presentation and Final Report. As the development of the project progressed, multiple changes were done in comparison to the initial diagram. Multiple new features were identified and added when executing the research phase. The diagram also provided a great way of tracking the progress also providing a clear picture of the development progress for the project supervisor, making sure the completion of different components of the system happens within defined deadlines.

Apart from the Gantt chart, another project management tool was used which is called Trello. Trello is a web-based application of the Kanban board [13]. The tool helped to define tasks of the project, dividing them into the categories of To Do, In Progress and Completed. This tool provided a great approach to track the weekly iterations and the objectives of each week. When the new tasks for the week were identified, they were assigned to the To Do category. During the course of the week, the tasks would move from To Do category to the Completed stage.

Chapter 7

Conclusions and Future Work

This chapter analyzes and evaluates the whole project upon its completion. The core of the chapter consists of evaluating the initially defined functional requirements of the system, outlined in Chapter 1. Due to the priority categories of each requirement, the requirements are compared against the resulting software and defined whether they were met, this way making sure that the minimum success criteria were achieved. After evaluating the requirements of the application, some limitations of the system are highlighted with propositions of how the issues can be tackled and how the system can be improved in the future research and development phase. Finally, the overall reflection on the project is discussed in the Conclusion section.

7.1 Requirements Evaluation

To estimate whether the final system of the project is a success, the final version of the distributed computation system that was developed will be compared against the main functional requirements that were outlined in the section 1.2.1. Analysing each of the requirements and whether the system delivered the desired features will conclude the success of the developed system. each requirement will be marked as MET if it was met and NOT MET otherwise.

1. **Job submission to the system. User must be able to submit one or more tasks to the system for execution.** (MET)

   The requirement belongs to the *must* category. The final version of the system successfully allows user to submit multiple jobs to the system for execution. If using the command-line interface, the user has to enter SUBMIT command specifying a submission file. When using the web interface, user has to upload the relevant files and fill the input fields.

2. **The system must return the results to the user after the job is completed. Provide an error log to the user if the execution was unsuccessful.** (MET)

   The requirement belongs to the *must* category. The final version of the system successfully returns the execution results to the user. The results are stored in a directory named by the unique task ID assigned for the submitted job. The results are also displayed when using the web interface of the system.

3. **The system given multiple jobs must execute them concurrently on different machines in the computer cluster.** (MET)

   The requirement belongs to the *must* category. As it was described in the section 5.1, the feature was explicitly tested and the system successfully performs multiple jobs execution concurrently on multiple connected Worker Nodes.

4. **The system must, if all the connected nodes are busy, take the submitted job and add it to the queue until one of the nodes become available. Then the job must be sent for execution.** (MET)

   The requirement belongs to the *must* category. The final system will, if there are no currently available Worker Nodes and the new job is submitted, add that new job to the queue where it will wait until one of the connected clients becomes available. The server will also successfully schedule the queue based on the user system usage time.

5. **The system must provide user with the details on the available machines in the cluster as well as provide the user with information on currently submitted running and queued tasks.** (MET)

   The requirement belongs to the *must* category. The final system contains properly working features of requesting system to provide information on currently connected Worker Nodes, their metrics and currently queued or running tasks. It can be performed by entering one of three commands in the command-line interface or selecting one of the features in the navigation bar of the web interface.

6. **The system must execute a submitted task with the dataset provided on one of the machines in the computer cluster and provide the user with the results of the execution.** (MET)

   The requirement belongs to the *must* category. The final system is able to successfully execute a submitted job that contains a dataset in the submission file. The feature was tested as it was mentioned in section 5.1.

7. **The application, providing all the specified information and features, must be available to use from the command-line interface.** (MET)

   The requirement belongs to the *must* category. The system was first designed and developed for command-line interface usage. Thus, the final version of the distributed system contains a command-line interface that provides the user with a control panel ready containing all the features of the system.

8. **User should be able to select the worker node depending on the metrics of each machine in the distributed system. The system should assign a worker node by itself if none was specified by the user.** (MET)

   The requirement belongs to the *should* category. The system successfully allows user to select one of the currently available Worker Nodes. The system will assign a unique number for each connected node and the user can specify the node's id in the submission file, if using the command-line interface, or select an available client when filling in the inputs in the Job Submission window when using web user interface. The system will then assign a job to the specified Worker Node.

9. **The system should notify the user once the job is completed by sending an email notification to the address specified within the submission file.** (MET)

   The requirement belongs to the *should* category. The final system, once the submitted job is completed, will send user an email notification using Simple Mail Transfer Protocol. The email can be specified in the submission file or entered in the email field when using the web interface.

10. **The submission of a job should be done by providing the system with a submission file that contains essential information about the job being submitted. Such as executable name, dataset name, etc.** (MET)

    The requirement belongs to the *should* category. In the final version of the system, job submission happens with user submitting a submission file via the command-line interface. The submission file contains information of the executable name, dataset name, selected Worker Node and email address. When using the web interface, the user has to upload the files and fill in the remaining fields, however, the same submission file is generated by the system and it is submitted.

11. **The system could state the time of the execution of the task.** (MET)

    The requirement belongs to the *could* category. The final version of the system states the execution time of the job when requesting information on currently running tasks. The complete running time will also be reported to the user via email notification after the job completion.

12. **User could be able to log in and log out of the system.** (NOT MET)

    The requirement belongs to the *could* category. The system does not contain the feature. The user will connect to the server by simply running the Control Node program. However, no authorization will be performed. The website contains a prototype login to the system, however, the website does not connect to the server of the distributed system as it is only a web interface for the Control Node.

7.2 Project Limitations and Further Work

As admitted in section 7.1, the primary goal was achieved - the project was successfully done by developing a prototype system of distributed computation system. Almost all of the core functional requirements of the project, except for one requirement of the *could* category, were met, implying the success of the project. However, due to the time constraints as well as learning the fundamentals of the area during the development process, the system contains several limitations and issues. Moreover, it is important to highlight the existing limitations of the system, since the outlined issues might be a great starting point of possible future research and development which could result in a more perfect distributed computation system.

First of all, it is essential to outline the main issue of evaluation of the project, which is the functional requirements of the system. The requirements of the project were set out having in mind the limited time, resources and rationally assessing the capabilities and skills of the project executor. It means that the project requirements were defined to be met at the end of the project. Developing an actual distributed system would contain many more, more complicated functional requirements, as for the prototype system of distributed computation system the requirements outlined were conceivable, thus, challenging enough for the third year project.

Another, probably the greatest limitation of the developed system is that currently, the system executes only the Python executable files. The prototype distributed computation system is not very optimal as it will not execute programs of different programming language or the job will fail if the new libraries and configuration files are used, where the machine does not contain such tools. However, the software could and should be extended by employing *containerization* to tackle this issue. Containerization encapsulates the application for computation in a different computing environment. The program is being packed with its required libraries, system tools, configuration files that are required for efficient execution in another environment. One of the most popular containerization tool used is Docker, which is standard, lightweight, secure and is available for both, Windows and Linux applications [14].

In the current system, the user can select the desired Worker Node on which its application will be executed. The rationale behind it is that the user can select the Worker Node with the machine's specifications that satisfy the user's needs. However, for a better version of such an application, the user should be able to enter the required resources for its task execution, such as specifying the desired number of RAMs, CPUs and whether the GPU is required while keeping the nodes of the application transparent for the user. With that being said, the software should be extended to be able to accept a submission file containing new fields of the resources being requested. Following that, further work needs to be done to improve the scheduler. The scheduler should be able to decide which machine should be assigned for the execution based on the requested resources. Also, the current priority queue algorithm should be improved as it will not be the most efficient and optimal for the system in many cases.

The final version of the project software contains a web user interface of the system. The web interface is an integration of the Control Node of the application based on the client side of the Client-Server Model application. The website allows user to access the same features of Control Node only by using the web interface. However, the system should be improved by creating a centralized website on the server side of the software.

The server currently stores only a small amount of the data on the database required for queueing and scheduling algorithms. Since the current version of the system returns the results of executed job to the user and discards them from the server, the software should be further extended to store the history of executed jobs for each user. The database in the server should store the results of each completed task so that the user could access that information from any other machine, rather than the one where the results were first returned. For this case, instead of using SQLite database, the system should make use of a more scalable, secure and robust database such as MySQL database [15].

7.3 Conclusions

To sum up, the primary goal of this project was to build a prototype system of distributed computation tool that is able to distribute multiple jobs to the cluster of connected nodes for execution upon user job submission. Also, implement other features such as node selection, queue and scheduling algorithms of submitted jobs or develop the attributes of the system that provide the user with the relevant information about currently connected worker nodes or currently running tasks. The system should contain command-line interface and web-based interface usage. As

discussed earlier, this type of application is rather complex if attempting to implement a complete product that is fully functional and ready to use in practice since it would require a great amount of resources and a thorough understanding of the field, also considering the time constraints. Taking into account the fact that the field is widely advanced and many similar systems already exist, another no less of a goal of the project was to research the existing systems and overall explore the area of the distributed computing systems.

Having this in mind, two sequential phases of the project were established: the stage for theoretical research of the area and existing systems whose goal was to develop a foundation of the main concepts of distributed systems to be able to design the system itself and the stage of system development, whose goal was to implement the desired tool that was designed. To achieve this purpose, during the project, multiple existing shared resources systems were researched and studied. Many scientific papers on the distributed systems were read to gain a profound understanding of the design of distributed systems, to find out the most common approaches for distribution algorithms for resource sharing and overall to learn the core concepts of the distributed systems. After the research phase, the project was set out into the key iterations of project development. Having the robust timeline with the core tasks defined and sticking to the plan in addition to the appropriate incremental software development methodology, the system was developed.

In conclusion, the project is considered to be a success, since the area of shared systems was thoroughly researched and the concepts of the distributed systems were learned. Also, the functional prototype system of distributed computation tool was developed that met the core requirements which were outlined prior to the development phase. Some limitation of the system were identified due to the time constraints and complicated nature of the field, however, at the same time, it proposes further work opportunities to improve the currently developed system.

Bibliography

[1] SINTEF, *Big Data, for better or worse: 90% of world's data generated over last two years*, May 22, 2013, `https://www.sciencedaily.com/releases/2013/05/130522085217.htm`.

[2] Kang S., Kim W., April 2007, 'Minimalist and Intuitive User Interface Design Guidelines for Consumer Electronics Devices', *Journal of Object Technology*, Vol. 6, No. 3

[3] *MoSCoW Prioritization*, 'https://www.productplan.com/glossary/moscow-prioritization/'

[4] *Appropriate Uses For SQLite*, 'https://www.sqlite.org/whentouse.html'

[5] Weijia J., Wanlei Z., 2005, *Distributed Network Systems: From Concepts to Implementations*, Boston: Springer Science+Business Media

[6] socket - Low-level networking interface.
`https://docs.python.org/3/library/socket.html`

[7] Socket Programming HOWTO.
`https://docs.python.org/3/howto/sockets.html`

[8] Eggen, R., Eggen, M., 2004 January, *'Efficiency Considerations of PERL and Python in Distributed Processing'*

[9] ftplib - FTP protocol client.
`https://docs.python.org/3/library/ftplib.html`

[10] smtplib - SMTP protocol client.
https://docs.python.org/3/library/smtplib.html

[11] Computing with HTCondor.
`https://research.cs.wisc.edu/htcondor/`

[12] Claes W., "Improving through an Incremental Approach", *Department of Communication Systems,Lund Institute of Technology, Lund University*

[13] 'https://trello.com/en'

[14] *What is a Container? A standardized unit of software*, 'https://www.docker.com/resources/what-container'

[15] *SQLite vs MySQL – What's the Difference*, 'https://www.hostinger.com/tutorials/sqlite-vs-mysql-whats-the-difference/'