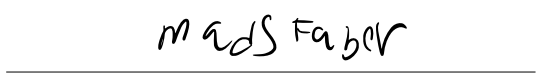
Offloading Computation to Mobile Phones

- Employers Dumping on Workers -

Project Report
SW711E20

Aalborg University
Computer Science

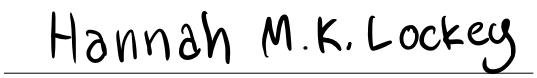
Laurits Brøcker



Mads Faber

Philip Irming Holler

Magnus Kirkegaard Jensen



Hannah Lockey

Computer Science
Aalborg University
http://www.cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Offloading Computation to Mobile Phones

Theme:
Distributed Computation

Project Period:
Fall Semester 2020

Project Group:
SW711E20

Participant(s):
Laurits Brøcker
Mads Faber
Philip Holler
Magnus Kirkegaard Jensen
Hannah Lockey

Supervisor(s):
Michele Albano

Page Numbers:
69

Date of Completion:
December 17, 2020

Abstract:

Mobile phones have become a necessary resource for for the average person. It is estimated that more that five billion people have some sort of of mobile device[1]. Even though most people today own a mobile device most are not fully utilized. This report proposes a system for utilizing the idle computational power by offloading from desktop computers to the mobile phones, in this case Android phones. The system includes both scheduling of jobs, consensus calculations for job results as well as proposing a system of trust for identifying possible possible malicious users. We tested two types of schedulers; FIFO and Economic with trust, and concluded that Economic with trust processed jobs slower, but had more correct results. We tested our system with a Raspberry Pi and a OnePlus 6, and concluded that with these devices our system is faster at processing jobs than standalone users.

Contents

Preface

This report is made by a group of 7th semester Software students from Aalborg University. This project involved creating a system for offloading computation to mobile phones.

Chapter 1

Problem Description

This chapter describes the reasoning that lead to the problem description and the problem description itself.

1.1 Initial Problem Analysis

In 2016 the network and communications company Cisco projected, that the workloads for cloud compute instances will more than double between 2016 and 2021[2]. This means, that more powerful data-centers have to be build to keep up with demand. This leads to both more power consumption, despite increases in efficiency, as well as a higher demand for server hardware. In this report we will discuss a solution for cloud computation using existing hardware, in this case mobile phones. The computational power of mobile phones is often not fully utilized, for example when the owner is sleeping. During such periods we could potentially leverage the unused processing power for cloud computing.

Additionally, mobile phones have become very energy efficient and computationally powerful. This can be seen in the cross platform CPU benchmark, Geekbench 5, by Primate Labs[3]. When comparing a current generation server CPU from Intel, the W2155, with the current generation flagship mobile System-On-a-Chip (SOC) from Qualcomm, the Snapdragon 865+, the efficiency of the Mobile chipset is clear. The results are summarized in table 1.1. The results vary slightly from device to device, so these numbers are the average results. Even though the Intel server CPU has almost three times the performance of the Snapdragon, it achieves this while pulling about 25 times the power.

	Snapdragon 865+[4]	Intel Xeon W-2155[5]
Cores and Frequency	8c/8t at 2.4-3.1Ghz	10c/20t at 3.3-4.5Ghz
Power Consumption	5 watt	≈128 watt[6]
Geekbench Score (single thread / multi thread)	≈900 / ≈3280[7]	≈1100 / ≈10200[8]

Table 1.1: Comparison of power efficiency

An important and common use case for cloud computing is Artificial Intelligence. For this area the hardware in mobile phones has also made big leaps in performance. According to Qualcomm their latest Snapdragon 865 SOC has a neural engine capable of up to 15 trillion operations per second, which is a five fold increase since 2017[9]. The neural engine can then be accessed through the so called Neural Processing SDK for AI[10]. This enables hardware acceleration using the neural engine as well as the graphics processing unit of the chip. Similar developments can be seen with the SOC developed Apple[11], Huawei[12] and Samsung[13].

Assuming that mobile phones are inactive a big part of the day, clearly significant and efficient computational power is left unharnessed every day. This power is, however, distributed among many devices on many networks. Additionally, the mobile phone can be considered a somewhat volatile resource, since they can at any time be picked up and used for a different purpose. In this case we assume, that the user expects the phone to instantly be responsive, and that the external application stops consuming its computational power and energy immediately. This leads us to the problem of getting the data and programs distributed to the phones and running them. This leads us to the following initial problem description.

Initial Problem Description

- How can a framework be designed for utilizing the computational power of mobile phones?
- How can the workload be distributed to the phones?
- How can the problem of volatile resources handled?

1.1.1 Initial Research

In the initial research for solutions to the problem description, we encountered the problem of choosing a target platform for the phone application. There are two major players in this field, Android from Google and IOS from Apple. Development for Apples IOS requires a Mac and the IOS operating system is more limited

in capabilities, e.g. no direct access to the file system. This in combination with the fact, that Android has a significantly higher market share world wide led us to focus on Android[14].

Next we researched the machine learning SDKs with focus on Qualcomm, due to their very high market share. Qualcomm processors account for 32-40% of world wide smartphones in 2020[10]. The Qualcomm Neural Processing SDK for AI did, however, seem a bit underdeveloped. It requires a development environment running an old version of Ubuntu (14.04), which lost its long term support in 2019. Additionally, according to their own website, the SDK does not even support their newest Snapdragon 865. The list of supported chips is in general very limited. Since the focus of this report is on the system for distributing the computations we instead chose to look for alternative methods of computation on the phones, that does not involve hardware acceleration.

This led us to the Termux Android App. This app emulates a bash terminal with a Linux environment[15]. Within Termux a Python interpreter alongside other development tools like the CLang compiler are directly available. Furthermore, Termux has direct access to its own part of the Android file system. This allows us to download and store programs in persistent storage. From here we can then compile or interpret the programs and run them on the device, which fits our purpose perfectly. Using Termux might, however, limit which types of programs are runnable, and also doesn't support hardware acceleration like the neural engine SDK does. Since the hardware acceleration might be important for the performance of our system, we chose to test the Termux solution with a simple C program, which can be seen in appendix A.

The test consisted of two parts. Part one was calculating the factorial of 65 20.000 times. Part two consisted of randomly shuffling an array of length 10.000 a total of 1000 times. For the test we used a rather old and low end mobile phone to test the worst case performance. In this case the 2017 Huawei Honor Lite 9 running Android 8.0.0. The code was compiled using the standard clang compiler in Termux, i.e. version 10.1. After this the code was run inside the Termux app. As a point of reference a high end current laptop was used, where the code was compiled and run using the Apple clang version 12.0.0. The laptop ran MacOS 10.15.7 and had an Intel Core-i7-9750H 6-core CPU. The results are an average of 10 runs.

The results of the test can be seen in table 1.2. The laptop was significantly faster, but the difference is not as pronounced as we initially expected. The high end laptop is about 6 times faster. The difference is relatively bigger for the array shuffling test. This could indicate a possible cache or memory bottleneck.

	High-end laptop	Honor Lite 9 in Termux
Factorial test	34ms	101ms
Array Shuffling test	94ms	683ms
Total	128ms	784ms

Table 1.2: Performance comparison of high end laptop and low end phone running Termux

This proves that the overhead of running the programs inside Termux is not too much for even a low end mobile phone. Even with the overhead, Termux still allows for significant computation.

To simplify implementation we choose to use Termux over the Neural Engine SDK despite the inability to use hardware acceleration. We instead focus on the distribution aspect of the problem, and many of the presented ideas will be independent of the method of program execution on the phones. This leads us to our final problem description.

1.1.2 Final Problem description

- How can a framework be designed for utilizing the computational power of mobile phones through Termux?
- How can the programs and data be distributed to the phones?
- How can the volatility of the mobiles phones be handled?
- How can users interact with the system?

Chapter 2

Design

In this chapter we will discuss the requirement specifications for a system, that solves the problem description. Furthermore we will propose an architecture for fulfilling the requirements specifications as well discuss design decisions for the individual modules of the solution. Before continuing we will first introduce some terminology.

2.1 Terminology

A **user** represents a person registered in the system. A user may represent multiple workers but only one employer.

An **employer** is the role a user has when it uploads a job to be computed by the system.

A **worker** is defined as a device that contributes processing power to the system by completing assignments. A worker is associated with one user. In theory a worker could be any device capable of running a modern version of the Android operating system, running Termux and having a connection to the internet.

A **job** is an entity in the system containing some job files to be processed by a worker. The job is created, when an employer uploads it to the system.

An **assignment** is the relationship between a job and a worker when a job is assigned to a specific worker. One job may be assigned to more than one worker, but a worker cannot have multiple assigned jobs at the same time.

2.2 Requirement Specification

In this section we discuss the requirements specification for a system capable of solving the problems in the problem description, see section 1.1.2.

1. The user must be able to offload Python programs by uploading them zipped to the system
2. The user must be able to see the status of his/her uploaded jobs
3. The user must be able to download results of finished offloaded jobs

4. The user must be able to enlist a mobile phone as a worker
5. The user must be in control of when his/her phone can process a job

6. It must be possible to create a user profile with a unique identifier and a password
7. The user must be able to acquire an authentication token from the server using their password and identifier
8. The user must have the option to sign in on the worker app using their password and identifier
9. The user must have their data regarding jobs accessible only by him/her

10. A worker phone should be usable while a job is running on the mobile phone
11. The user should be able to stop a job running on a mobile phone at any time
12. The user should be able to see statistics regarding job completion for his/her mobile phone
13. The user should be able to specify how many workers should process a given job to ensure the integrity of the results

The first requirement could have been expanded to running arbitrary code, but this is limited by the compilers and interpreters available in the Termux application. Both the Clang compiler and a Python interpreter are available, but for the sake of time we focus only on Python programs.

We envision the user uploading several jobs, and for this reason it would be advantageous for the user to be able to see all of the uploaded jobs as well as the

status of them. Since the point of uploading a job for computation is getting the result of the job, we included a requirement for downloading the result files.

In order to get some workers in the network for processing the jobs, there must be a way for the user to enlist a mobile phone as a worker. Since the user may want to user the mobile phone at any time, we add the requirement, that the user must be able to cancel a job running on his/her phone at any time. We imagine a user wanting to use his/her phone quickly, but without canceling the job, and for this reason we added the requirement, that the phone should still be usable while running a job.

In order to join the network in the first place a user must be able to create a user profile. This also allows for encapsulation of data specific to a user, so that the user only sees data relevant to him/her self.

2.3 Architecture

In order to create a system that fulfills the requirements, we first define an architecture of the system. In this section we will discuss the architecture.

For the design we came up with four main components. The first component is the worker client using Termux, which should run on the Android phones. Additionally, we need some entry point for users to upload jobs, i.e. an employer client.

If we simply connected these two directly, with e.g. socket programming, then it required for both the workers and employers to be online simultaneously when the job files are sent to the worker and when the results are sent back to the employer. This design is not well suited for a volatile resource like a mobile phone, i.e. the worker. Additionally, it would not be ideal to expect the employer to stay online while the worker processes the job. For this reason some module for storing both is needed. Additionally, the workers and employers need to be able to locate each other and be paired up. This process should preferably also be possible without both parts being online at the same time. For solving this problem, we propose having a server module, that can handle pairing up jobs from employers. In order to more easily pair up the employers and workers, we need some way of identifying them uniquely. Furthermore, if the jobs are temporarily stored in the server, we need some way of uniquely identifying them again, in order to locate them at a later point in time. This leads to the need for a database module connected to the server.

The server and database modules can also help with the requirement regarding encapsulating data, as we can use it to map jobs to a user, and to store user information such as username and authentication data. The server can handle the communication for creating accounts and encapsulating data send to the user, while the database can store the state and user credentials for a given user. This leads us to an architecture composed of four different modules: Employer clients, Worker clients, a server and a database.

In figure 2.1 we have created a flow diagram of how we envision the overall architecture. Here it can be seen how all communication is handled through the server.

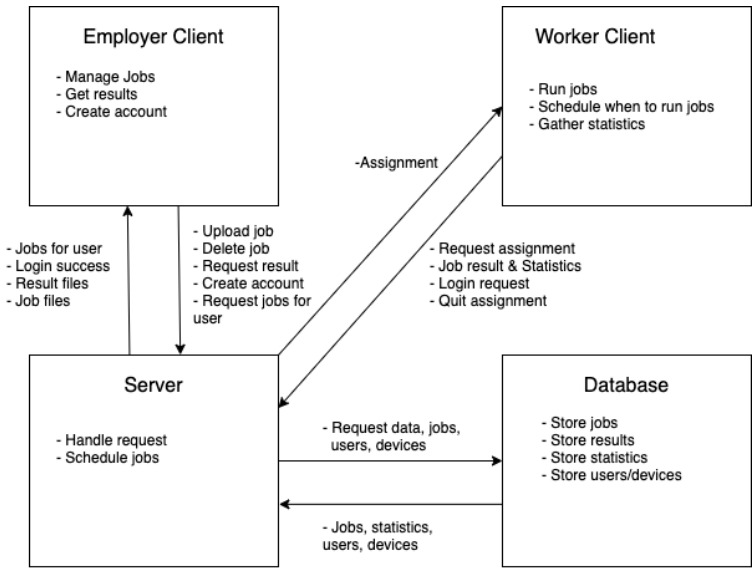


Figure 2.1: System architecture

In the next few sections we will discuss the design of each of the four components in more detail. The implementation of the four modules will be described in chapter 3.

2.4 Database

As described in section 2.3 the database was introduced as a solution for storing the state of jobs, as well as uniquely identifying jobs and users in the system. The database will only be accessed directly from the server. The employer client and worker client will only communicate with the database through the server.

A visual representation of the database can be seen in figure 2.2. A user table is introduced in order to uniquely identify a user as well as store the user credentials for the data access management. In the database a surrogate key called user_id is created as the primary, which is also used as a foreign key in the other tables when referencing a user entity. The user_id is an automatically generated unique id. The cpu_time column can be used for prioritization and scheduling of jobs. The design for this process will be described further in section 2.5.

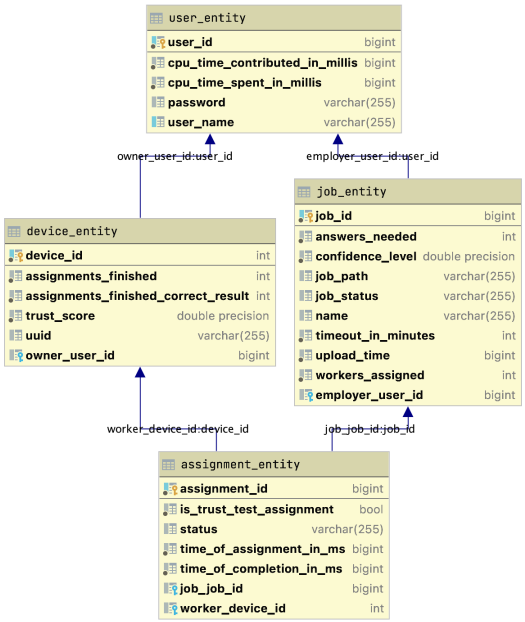


Figure 2.2: Table diagram with foreign keys for the database

In the requirement specification, see section 2.2, it was stated, that a user should be able to specify how many workers should work on a given job. This was introduced to ensure, that the user can protect itself against a single device returning the wrong result. In order to accommodate this in the database we introduce the assignment table in combination with the answer_needed column in the job entity. The assignment table has a many to one relationship with the job table, which allows for one job having many assignments. The assignment table also holds a foreign key to the specific device, that has received the assignment. Each assignment also has a status, which should hold the information regarding whether or not the worker has finished processing the job.

In order to locate the job files on the server, we also store the path to the job files in the database as a column in the job table. The job table also holds simple

information such as the job status, the name of the job and the upload time. These simple values can be used to display to the user some information regarding previously uploaded jobs. The job_id is a an auto incremented unique id. Each row in the job table also holds the status of the job. This status will be updated according to how the amount of the assignments that references the job and has the status *done*.

The device table holds an UUID number, a universally unique identifier that is pseudo randomnly generated, used to identify the specific devices in assignments and statistics.

2.5 Server

The server acts as a coordinator between workers and employers. Employers and workers connect directly to the server. The server is responsible for managing incoming jobs, assigning jobs to worker clients, and making processed results available to employers.

2.5.1 API Interface

The server should provide an interface that exposes jobs, results and assignments to the employer and worker clients. It should also allow for registration of users and login for authentication. This interface is depicted in figure 2.3 where it is represented as HTTP-style requests.

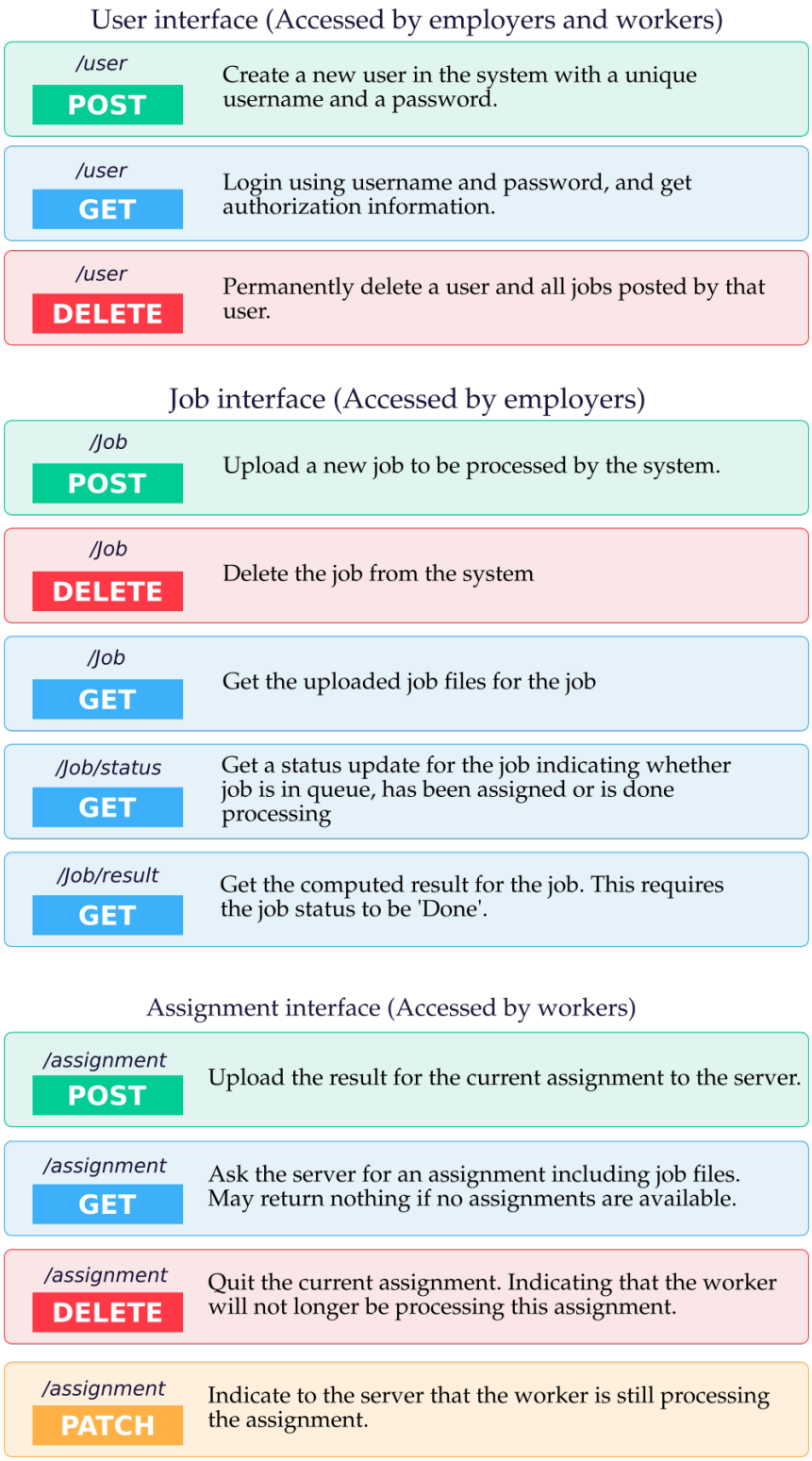


Figure 2.3: Interface design for the server

The user interface allows for creating new users, deleting old user and logging in. A user is not allowed to access other parts of the interface before being authenticated by a log in.

The job interface is used by the employer clients. Jobs can be uploaded by posting job files along any necessary metadata. Once a job has been uploaded the employer can download the job files again or delete the job from the server. Furthermore the employer can retrieve the current status of the job, which indicates whether the job is waiting in queue, being processed or is completed. When a job is completed the employer can retrieve the result through a get request. Note that all job requests relating to a specific job can only be accessed by the user that posted that job.

The assignment interface is used by the worker clients. The worker can request an assignment, which may yield a set of job files to compute, depending on whether the server has an unprocessed jobs available.
A worker should periodically send a heartbeat-request with a patch method to the assignment interface, in order to let the server know, that it is still alive and working on a given assignment.
It is possible for a job to be completed while a worker is still working on it, and in order to abort a worker who is doing work on a job that is already done, the worker client uses the response from the heartbeat-request to determine whether the job has been completed or not.

2.5.2 Scheduling

When a worker requests an assignment from the server, it is important to consider what job it should be assigned to maximize the effectiveness of the system and induce fairness. In this context the fairness depends on the measurement chosen when implementing the scheduler. For example, one might consider a FIFO queue fair, while others think that the contributed computation time should determine the priority of the jobs from a given user. There are many possibilities when designing a scheduler. In this report we focus on just three.

FIFO scheduler

+In order to get a baseline for a naive approach, we propose the FIFO scheduler. Using this approach, the oldest job in the system is the first one to be assigned to a worker. This scheduler only takes into account when a job was uploaded. Employer id, timeout and answers needed are not taken into account.

Economic scheduler

To provide an incentive for contributing computational power to the system, we introduce the economic scheduler. In addition to incentivizing contribution of computation, the economic scheduler also makes use of locality in the sense, that it always tries to give assignments to the employers own workers first. This is done, since it does not cost anything in terms of CPU time spent. Additionally, the risk of manipulation of the results will be diminished, since a user would not ruin the results of its own job. In case the user has no available workers the economic scheduler will prioritize users with the highest banked CPU time. This is what the word economic refers to. Banked CPU time is how much CPU time the user has earned subtracted by how much it has spent. In this case that means how much time the users workers has contributed with its workers subtracted by the time other users have spent on this user's jobs. This feature should incentivize the users to enable workers to compute in order to get ahead in the queue, when they upload a job. After sorting by banked CPU time, the scheduler picks the oldest job from the user, i.e. FIFO picking. If multiple users would have the same banked CPU time then the oldest job between the users will be chosen.

Trust scheduler

The trust scheduler is an extension to the economic scheduler. This means, the prioritizing of the employers own workers as well as the incentives for contribution workers cpu time are still present. In addition we introduce trust as a measure. For the trust feature we took inspiration from a big distributed network, the Bittorrent network, where trust in peers is also used. Bittorrent uses something called tit-for-tat with optimistic unchoking[16]. Tit for tat means, that you need to contribute in order to receive. This is in line with the economic scheduler. Optimistic unchoking refers to not completely ignore bad peers, i.e. peers that, in the Bittorrent context, do not upload. Upload spots are assigned to the best peers for a given file, but with a random interval the algorithm chooses a random peer, which allows for bad peers to increase their trust score, if they choose to upload.

For our system we do something similar in the sense, that all workers get a trust score. The trust score is a floating point value between 0 and 1 and is initialized at 0.5 when a worker joins the system. Whenever a worker is in the majority for result comparison, i.e. is likely to have the correct result, we increase the score by some value, for example 0.05. However, in order to quickly punish a malicious worker, we divide the trust score by 2, whenever a worker is in the minority for

results. The trust scheduler can then choose to not assign real jobs to the worker. This can, however, completely starve the worker for jobs in the system, which we are also not interested in, since it might have been a bad coincidence. For this reason, we introduce our version of Bittorrents optimistic unchoking. A user, that the scheduler does not trust, can still receive an assignment. This assignment is called a *test assignment* and will not count towards the amount of workers assigned to the job, nor be used for fetching the final result for the job, when all workers have submitted a result. The test assignment will, however, still count towards the workers trust score, which can increase and eventually be high enough for the worker to be trusted again. This ensures, that legitimate workers do not stay untrusted forever.

2.6 Employer Client

This section will describe the design of the employer client. The design should solve the requirements seen in section 2.2. Additionally, we propose mockups for the UI design of the employer client.

The employer client should consist of two parts. A front-end interface, which should enable the user to easily manage jobs, as well as a network module for communicating with the server. The employer client is used to add jobs to the server, as well as pull results in accordance with requirements 2 and 3 from the requirements specification.

Since the employer client is a simple front-end interface for the user to interact with the server, it could be designed in more than one way. We specifically identify two distinct ways of doing this, either as a website or a desktop application. The advantages of a website would be the ease of installation and updating. One of the advantages of a desktop application is the performance. This is, however, not very important in a simple interface like this. For this project we propose a desktop application due to us having more experience in this field. The employer client could, however, just as well have been a website or an entirely different application than the one we proposed, as long as the client adheres to the server API.

In order to show, how we envision the functionality of the employer client, we created an figure that shows the interaction between the employer client and the server, see figure 2.4.

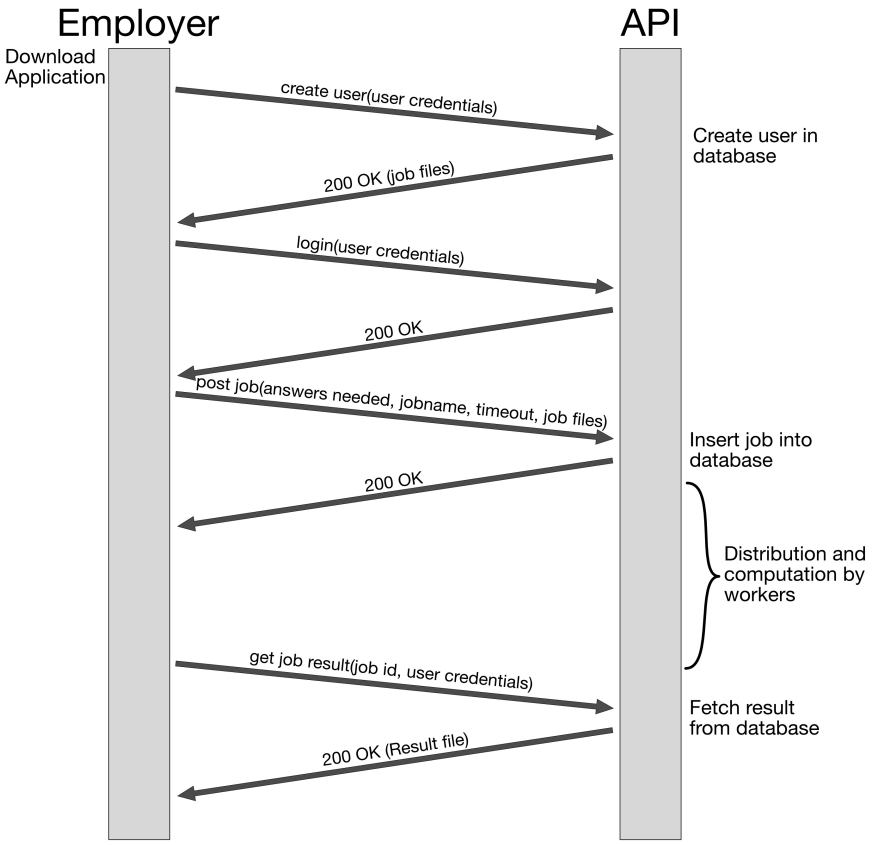


Figure 2.4: A sample interaction between the employer client and the server

First the user downloads/installs the employer client. After this the user should be prompted to either login or create an account. This screen should look like the one in figure 2.5.

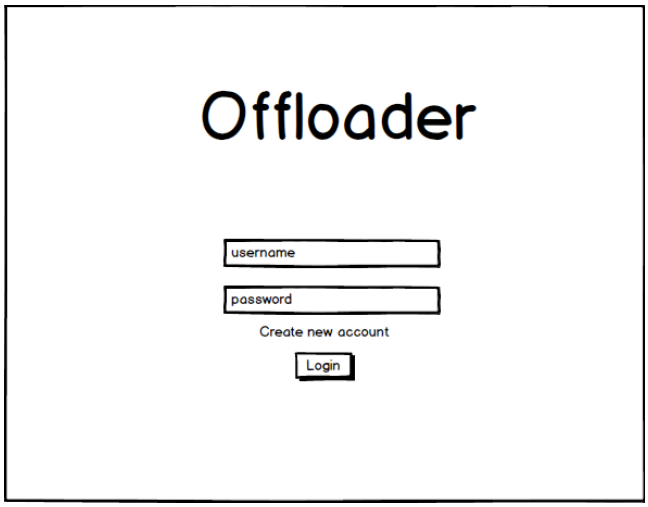


Figure 2.5: Prompting the user for login mockup

To follow the flow from figure 2.4 the user then clicks create account which brings the user to the screen seen in mockup 2.6. To ensure that the user does not accidentally input the wrong password a second password field is included. The user then has to input the password twice correctly, which should minimize the amount of password errors.

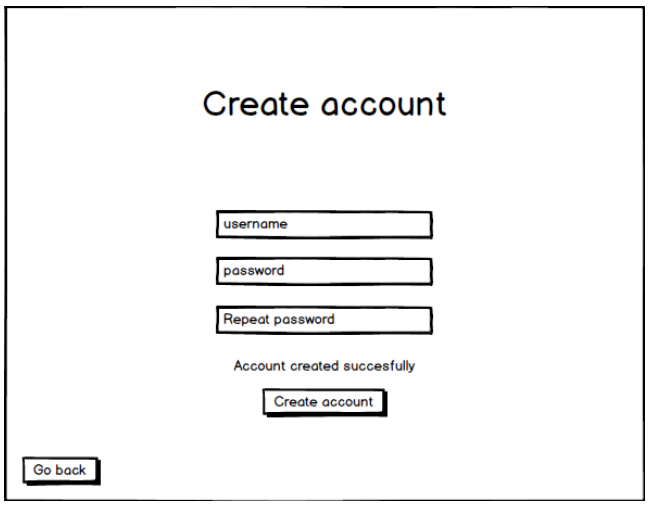


Figure 2.6: Account creation screen mockup

The user can now login using the credentials that were given in the account creation part. After the login the user is greeted with the dashboard, as seen in the mockup in figure 2.7. From here the user can see a table view of the jobs currently uploaded to the system. Initially, this is empty. For downloading the result files or

deleting the job from the system, we propose a context menu when right clicking a given job / row in the table view. The context menu contains three items. *Download job files* will redownload the job files initially uploaded when the job was created. *Download result files* will return the result files, if they are available. *Delete job* will delete the job by requesting a job deletion from the server.

The right side of the dashboard consist of a quick guide and description of how to use the system. In the bottom right corner a text field for the path to folder containing the job path to as well as a button for using the operating systems directory finder for finding the path to the folder containing the job files.

Since the user must also request a number of workers for the job, a combobox for selecting this workers was added. A combobox is a good idea for this purpose, since it restricts the user to some predefined values. The same could have been done for the timeout, but since the timeout in minutes can be any number from a big range, for example 0 to 600 minutes, a combobox would get too big. For this reason we chose to use a text field. This text field will, however, only accept integer inputs.

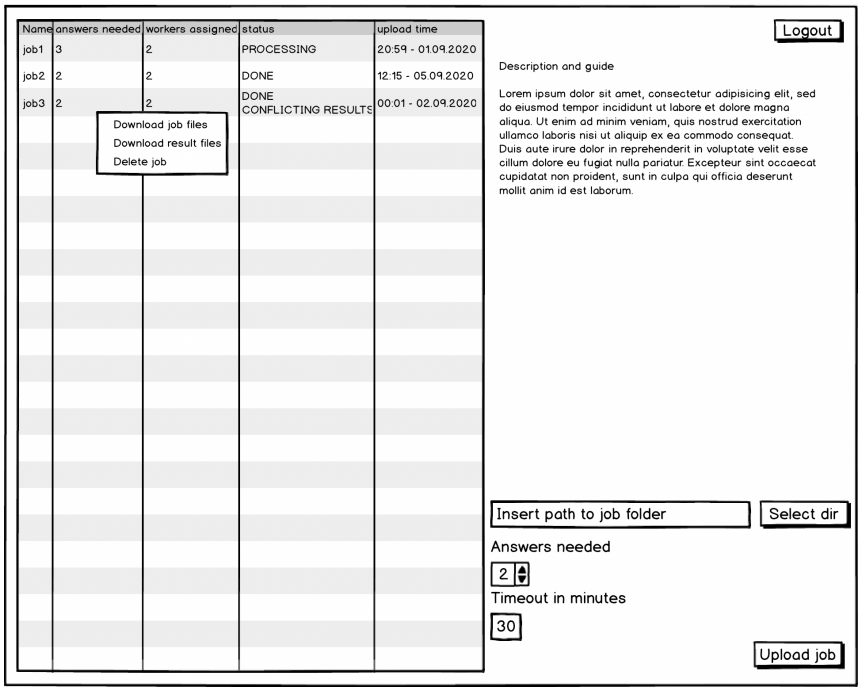


Figure 2.7: Dashboard view mockup

The network module of the employer client will contain several functions for communicating with the server in order to allow for the functionality exposed by the front-end of the employer client. These functions can be seen here:

- Get all jobs for a given user
- Upload a new job
- Download job files
- Download job results
- Delete Job
- Login
- Create account

With the network module and the front-end module the employer client is able to fulfill the requirements specifications.

2.7 Worker Client

The server must be able to offload jobs to mobile workers. To do this a mobile worker client is designed. The worker client is designed as an application for Android mobile phones as these are found to be the highest market share of mobile phones in section 1.1.1.

The flow of the application can be seen in figure 2.8. When the app starts, the user must log in using a existing username, or by making a new one. A wireframe of this can be seen in figure 2.9. After the log in the user can either choose to go to settings, log out or start processing jobs. If they choose to start processing jobs the worker is then set to *active* and the worker will process the jobs it is given. Once a job is done or canceled, the client will be ready and on stand by for another job from the server. This cycle continues until the user-specified run conditions no longer hold true, or the client is manually stopped.

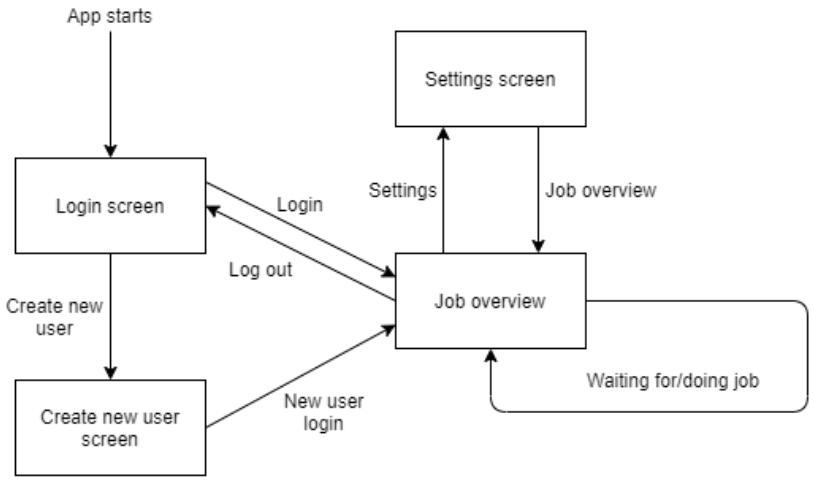


Figure 2.8: The flow of the worker client

If the worker logs in with an existing user tied to an employer, then the worker unit is joined to the employer. The ability to make a new user by the worker means that a worker can make new user that is not tied to a employer.

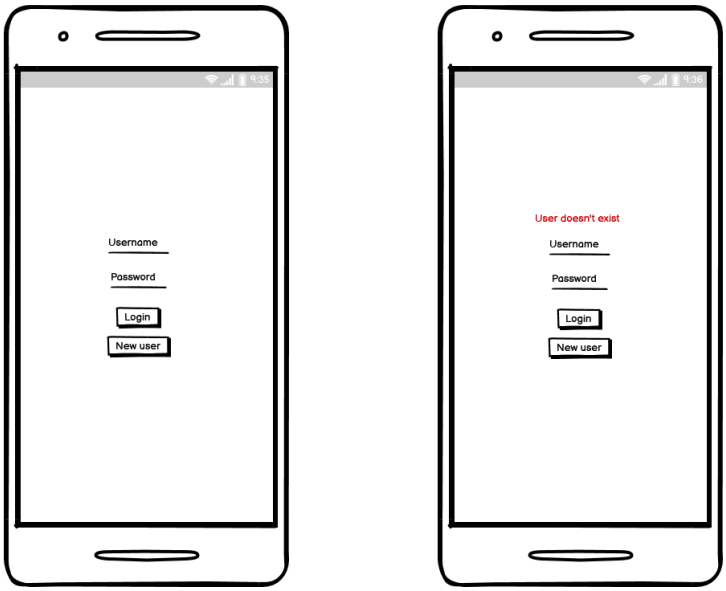


Figure 2.9: Worker login wireframe

The settings screen will be made so that a worker themselves can choose when their phone can be used to calculate a job. An example of this could be that a user chooses that their phone can only be used when it has a WiFi connection, and in

the 11pm to 7am time frame. This design can be seen on the right side of figure 2.10

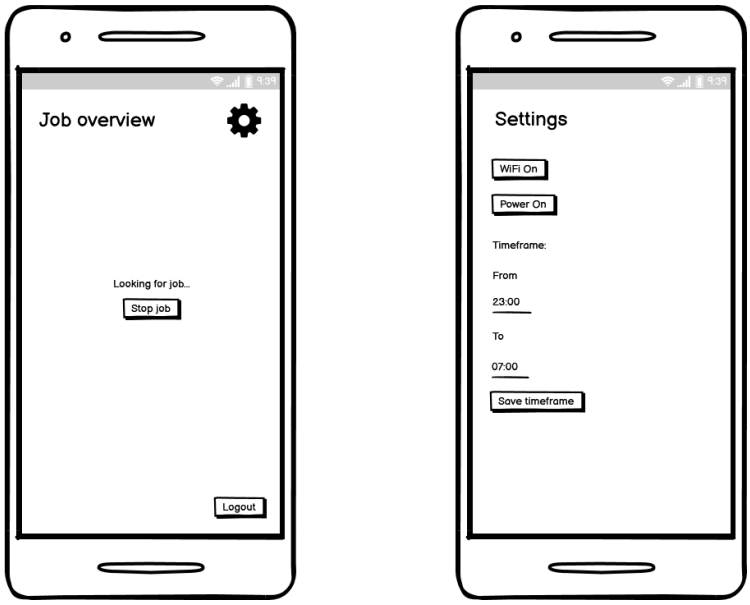


Figure 2.10: Worker job overview and settings wireframe

Furthermore, a client should be able to interrupt a running job, in order to send a timeout message to the server, if the job takes too long or loops infinitely, or stop if they do not want to contribute their phone anymore. To do this, the "Accept jobs" button turns into a "Stop job button" so that the worker can stop at anytime. This can be seen in the left side of figure 2.10.

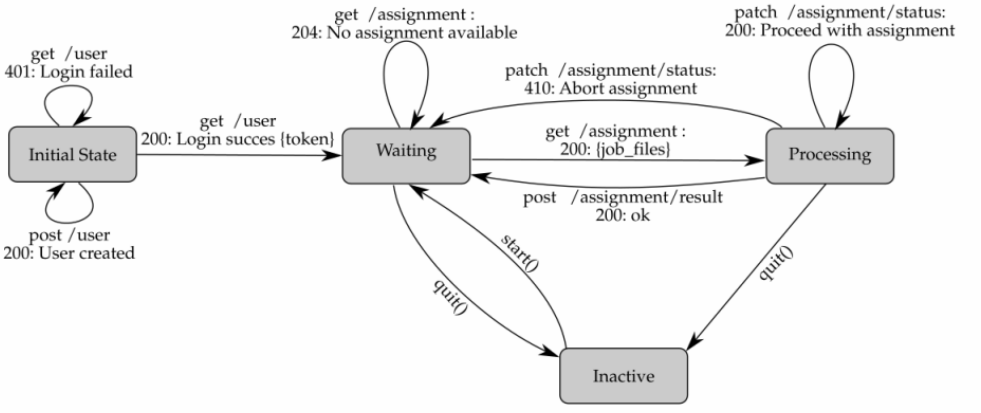


Figure 2.11: Worker network module

To connect to the server the worker client should have a network module. A flow diagram of this module can be seen in figure 2.11. After the login it can be seen that the application will be in *waiting* mode, processing incoming jobs until the user manually stops the process or until the user-specified run conditions no longer hold true. The application will then move to the *inactive* mode until the user starts it again.

Chapter 3

Implementation

This chapter will explain the general implementation of the system and of the technologies used to create our solution.

3.1 Server

This section describes the implementation of the server and is based on the design described in section 2.5.

3.1.1 API Interface

The API can be implemented in many different ways. One of the more traditional ones is an HTTP Rest API, which is compatible with most applications. Another possible HTTP interface could be a GraphQL API. For our implementation, however, we chose to stick with a traditional HTTP Rest API, as the software support for HTTP clients and servers is very mature and has excellent tooling.

Implementing a Rest API can be done either manually or by generation from a specification. Creating a manual implementation might result in easier debugging, but in turn it is much less maintainable. Using a generator allows for quickly accommodating any changes made to the specification. For this reason we chose to use a an API generator for our server.

There are several tools for generating an API, for example OpenAPI-generator[17] and Swagger[18]. When choosing an API generator, one also has to take into account the languages, that the generator can generate. For this implementation we chose to generate the server in Spring, but other languages could have worked as

well. Both Swagger and OpenAPI-generator can generate Spring code. Additionally, we had to consider, if the generator could also generate code for our client, which is written in Kotlin, see section 3.2. This is also possible with both Swagger and OpenAPI-generator. We chose to use OpenAPI-generator, but we could have just as well used Swagger.

Specification

The specification used by the OpenAPI generator is written in the .yaml format. Our specification can be seen in appendix B. In the specification we first define our schemas. These contain specification for the objects, that we wish to attach to in- and outgoing requests. One example of this could be the **JobFiles** schema, which can be seen in code snippet 3.1. This schema will then be generated as an object, and that object will be used when a request for the jobFiles for a job is received. The object contains a job id as well as the data from the jobs files formatted as bytes and written as a string.

```
1 JobFiles:
2       type: object
3       properties:
4         jobid:
5           type: integer
6           format: int64
7         data:
8           type: string
9           format: byte
```

Code snippet 3.1: JobFiles Schema from API specification

Next we describe the paths for our API. These correspond to the ones defined in the design of the API, see section 2.5.1. The difference here is, that now we also specify parameters for the requests. An example of such a path could be the */users/userCredentials*. This request can be seen in code snippet 3.2. This specific request is used for account creation. Here it can be seen, how the tag parameter is used to group the requests: This particular one belongs to the user group. It should be noted, however, that the tag is only used to group the endpoints for a developer, and does not have an impact on the generated code. In the parameters it is specified, that the user credentials are sent as a parameter in the path of the request, and that they are required. In the future we would change this to be an access token for security reasons, but for now we include the user credentials. Finally, some responses are defined. In this example the 200 OK HTTP code contains the user credentials. The parameters from the specification defines the parameters generated for the methods in the API interfaces for the controllers, which are described in further detail in section 3.1.1.

```yaml
paths:
  /users/{userCredentials}:
    post:
      tags:
        - user
      description: Creates a user
      operationId: createUser
      parameters:
        - name: userCredentials
          in: path
          description: Credentials used to create new user
          required: true
          schema:
            $ref: '#/components/schemas/UserCredentials'
      responses:
        '200':
          description: User Created
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/UserCredentials'
    ...
```

Code snippet 3.2: Specification of user creation request

Controllers

As server handles requests through HTTP endpoints, the server should be able to differentiate between requests on different paths. For that reason, routing is implemented using a controller pattern.

When OpenAPI-generator generates code from the specification, it generates API interfaces for each route used in the specification. This means, that we get three different interfaces for the controllers, that we must implement, i.e. the **AssignmentsAPI**, the **JobsAPI** and the **UsersAPI**. Code snippet 3.3 show how our implementation of the assignment controller interface handles incoming assignment requests from workers.

Here it can be seen, how the interface generated from specification is implemented and the methods overwritten. The methods showcased here is the method for handling a device requesting a job to compute. First the user credentials are checked against the user repository. The repositories are injected into their respective interfaces for interacting with the database. The repositories will be described in further detail in section 3.1.2. Next the controller checks whether the device actually belongs to the user requesting. If this is the case, we check whether the device already has an assignment, and should thus continue working on this assignment. If this is not the case, we ask the job scheduler for a new job, which is then returned in a request with status code 200.

```java
@Override
public ResponseEntity<JobFiles> getJobForDevice(UserCredentials
    userCredentials, DeviceId deviceId) {
    // Verify user authentication
    if (!userRepository.isPasswordCorrect(userCredentials))
        return ResponseEntity.badRequest().build();

    DeviceEntity device = repository.getDeviceByUUID(deviceId.getUid());

    // Check if device is already doing a job, but crashed or something
    Optional<AssignmentEntity> oldAssignment = repository.getAssignment(...);
    if (oldAssignment.isPresent()) {
        JobEntity job = jobRepository.findById(oldAssignment.get().jobId());
        File jobFile = jobFileManager.(getJobFile(job.get().jobPath));
        return ResponseEntity.ok(jobfiles);
    }

    // If device is not already doing a job find one through the scheduler
    Optional<JobEntity> job = jobScheduler.assignJob(device);
    if (job.isPresent()) {
        JobEntity jobValue = job.get();
        File jobFile = jobFileManager.getJobFile(job.get().jobPath);

        AssignmentEntity assignmentEntity = null;
        if (jobScheduler.usingTestAssignments()
            && !jobScheduler.shouldTrustDevice(device)) {
            // The worker is not trusted,
            // but gets the assignment as a test assigment
            // His answer is not counted, but his trustscore can increase,
            // if his answer is correct
            assignmentEntity = new AssignmentEntity(device, job.get(), true);
        } else {
            assignmentEntity = new AssignmentEntity(device, job.get(), false);
            jobValue.workersAssigned++;
            jobValue.jobStatus = PROCESSING;
        }
        // Increase cpu time spent by timeout for employer.
        // This decreases his priority according to the economic schedulers
        // Only if the job and device is not his own
        if (device.getOwnerId() != jobValue.getEmployer().getUserId()) {
            UserEntity employer = jobValue.getEmployer();
            employer.setCpuTimeSpentInMillis(
              employer.getCpuTimeSpentInMillis() + jobValue.timeoutMillis);
            userRepository.save(employer);
        }

        // Save the job changes in the database
        jobRepository.save(jobValue);
        assignmentRepository.save(assignmentEntity);
        return ResponseEntity.ok(jobfile);
    }
    // If not job is present, return status 202
    return ResponseEntity.status(202).build();
}
```

Code snippet 3.3: Implementation AssignmentsController

3.1.2 Data Management

The information regarding the jobs, assignments, worker devices, and users are stored in the database. For our implementation we used a PostgreSQL database. Other database management systems could, however, have worked equally well. In order to tie the server together with the database, Spring offers some functionality like repositories and entity generation using a tool called Java Persistance API or JPA, which will be described in further detail in this section.

JPA allows for generating a database with tables and columns from class definitions. An example of this, the job entity, can be seen in code snippet 3.4. JPA generates the SQL query commands from the fields and annotations used in the class. It finds the class through the *@Entity* annotation. The job entity corresponds to the *job_entity* table in the database. Here we can also generate relations such as foreign keys and constraints like not null and primary keys. In the job entity for example, we let the job id be an auto generated long and the primary key. Additionally, we define a relationship between a job and a user by annotating with the *@ManyToOne* annotation. This tells JPA, that many jobs are related to a single user, which will result in a column in the job table with a foreign key to the user table. Other relationships like one-to-many and many-to-many are also possible. The job entity also holds a field for jobPath, which holds the path to the job files on the server, an upload time a status and several others. The confidence level is used for the economic with trust scheduler, which is further described in section 3.1.3.

```
@Entity
public class JobEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long jobId;

    @ManyToOne
    public UserEntity employer;
    private String name;
    public String jobPath;
    public long uploadTime;
    public int answersNeeded;
    public int workersAssigned;
    @Enumerated(EnumType.STRING)
    public JobStatus jobStatus;
    public int timeoutInMinutes;
    public double confidenceLevel;
    ...
}
```

Code snippet 3.4: Entity definition for the database

Similar entity classes are created for assignment, device and user. This generates exactly the database designed in section 2.4 plus some columns for implementations details like the confidence level. In order to interact with the generated database, spring and JPA enables creation of repositories, which act as an interface to the database that can be used in the controller code. The repositories are then injected into the controllers using the *@Autowired* annotation. The repositories include basic functionality like *findByPrimaryKey(key)*, *findall()*. Additionally, the repositories can be extended with custom methods for data extraction. In code snippet 3.5, we have extended the user repository with queries e.g. asking whether a password is correct or querying whether the user exists in the database. These are defined with the *@Query* annotation along with some manually written SQL. In our implementation the password is simply saved in plain text, but should obviously be changed to a hash of the password at a later time for security reasons.

```
1  public interface UserRepository extends CrudRepository<UserEntity, Long> {
2      @Query(value = "SELECT EXISTS (SELECT * FROM user_entity WHERE user_name =
        ?1)", nativeQuery = true)
3      boolean userExists(String username);
4
5      @Query(value = "SELECT EXISTS (SELECT * FROM user_entity WHERE user_name =
        ?1 AND password = ?2)", nativeQuery = true)
6      boolean isPasswordCorrect(String username, String password);
7
8      @Query(value = "SELECT user_id FROM user_entity WHERE user_name = ?1",
       nativeQuery = true)
9      long getUserID(String username);
10
11     @Query(value = "SELECT * FROM user_entity WHERE user_name = ?1",
       nativeQuery = true)
12     UserEntity getUserByUsername(String username);
13 }
```

Code snippet 3.5: JobRepository interface to database

For storing the jobs files we make use of the jobpath field in the job entity. This path contains the path to the file stored on the server locally. For saving, deleting and fetching the data files, we introduce the **JobFileManager** class, which is able to do this using the path from the job entity. This class also handles incoming result files, when a worker uploads it. Here, however, some more logic is needed, since we also compare the results uploaded by the workers to ensure they are correct.

Whenever a worker uploads a result, the result file is saved in a folder containing the intermediate results. When enough answers have been submitted we check for equality. The idea is, that if many results agree, we can say with a higher confidence level, that the result is correct. This is done using the *getConfidenceLevel()* function created for the purpose and can be seen in code snippet 3.6. First the function creates a map from hashes to lists of paths (strings). We call this map the

HashToFilesMap. The idea is, that the hash which has the longest list of files which content hashes to this value is the best answer in terms of confidence level.

Next we iterate through all the zipped result files and generate a hashmap for each intermediate result. This hashmap maps from a path inside each the zip file to the content of a given file, while disregarding meta data such as creation date etc. We call this the FileToContentMap. Examples of entries in this map can be seen below. The value is hashed using the SHA-256 algorithm.

folder1/resultfile2 : e45a68ae385e458512784f9cfb873b2215c61b664e45a41e40dbf2
resultfile1 : 8c7c63ab67501043bf4a09d8d7360df54bd3bed188edac16cd2430

If two result zip files are identical the two sets of key-value pairs in the FileToContentMaps must be identical. This is due to the fact, that if some file inside one of the results had a different subpath compared to the other result zip file, the key would change. If the contents differed, the value would differ. This means, that if we hash the string representation of FileToContentMaps, the hash should only be identical if all files and paths to the files are identical. We then insert into the HashToFilesMap an entry with a key being the hash of the FileToContentMaps and a value being the path to the result file.

After creating the HashToFilesMap we can iterate through it and find the key, that holds the longest lists of files as value, i.e. the most files agreeing on the result. A data class called **ConfidenceResult**, which holds a list of ids of the assignments that had the correct results, a list of the ids of the test assignments with a correct result, the path to the best result file, i.e. a path to any file containing the correct result, and finally the confidence level, a floating point value between 0 and 1.0. Finding the assignment ids is done by parsing the file name of the result file for a given assignment, since it was appended the assignment by the server, when the worker uploaded it. The confidence data is then used to update the status of a job in the database through the controller.

```
fun getConfidenceLevel(fileList: List<File>) : ConfidenceResult {
    var confidenceResultData: ConfidenceResult = ConfidenceResult();
    // Map from hash of contents to files having this hash
    var hashToFilesMap: MutableMap<String, MutableList<String>> = mutableMapOf
    ();


    for(file: File in fileList){
        var fileToContentMap = getFileContentHashMap(file)
        if(hasher.hash(fileToContentMap.toString()) !in hashToFilesMap.keys){
            hashToFilesMap[hasher.hash(fileToContentMap.toString())] =
    mutableListOf()
        }

        hashToFilesMap[hasher.hash(fileToContentMap.toString())]!!.add(file.
    absolutePath);
    }

    // Find biggest number of entries agreeing
    var highestNumberOfResultsAgreeing = 0;
    var highestNumberHash = "";
    var nonTestAssignmentsFound: Int = 0;
    for (key in hashToFilesMap.keys){
        // Filter to remove test assignments, which have the name _testAssig.
    zip at the end
        var list = hashToFilesMap[key]!!.filter{ Regex(".*/result_file_
    ([0-9]*).zip").containsMatchIn(it) };
        nonTestAssignmentsFound += list.size;

        if(list!!.size > highestNumberOfResultsAgreeing){
            highestNumberOfResultsAgreeing = list.size
            highestNumberHash = key
        }
    }

    // Add assignment ids, that were in the majority result group
    // Paths look like this: /Users/philipholler/IdeaProjects/Offloading/
    test_data/result_test/results/result_file_15.zip
    // Where 15 is the assignment id of the result file
    // testAssig.zip are results from test assignments
    for(f in hashToFilesMap[highestNumberHash]!!){
        if(Regex(".*/result_file_([0-9]*).zip").containsMatchIn(f)){
            var matches = Regex(".*/result_file_([0-9]*).zip").find(f)!!
            var workerId: Long = matches.groups.get(1)!!.value.toLong();
            confidenceResultData.correctAssignmentIds.add(workerId);
        }
        // Still give credit to test workers
        else if(Regex(".*/result_file_([0-9]*)_testAssig.zip").containsMatchIn
    (f)){
            var matches = Regex(".*/result_file_([0-9]*)_testAssig.zip").find(
    f)!!
            var workerId: Long = matches.groups.get(1)!!.value.toLong();
            confidenceResultData.correctTestAssignmentIds.add(workerId)
        }
    }

    confidenceResultData.bestFilePath = hashToFilesMap[highestNumberHash]!!.
    get(0)
    confidenceResultData.confidenceLevel = highestNumberOfResultsAgreeing.
    toDouble() / nonTestAssignmentsFound.toDouble()

    return confidenceResultData;
}
```

Code snippet 3.6: Function for checking zip file equality

3.1.3 Scheduler

The scheduler is implemented using dependency injection by defining how a scheduler is instantiated in a Spring configuration file and then injecting it into the relevant controllers. An example of how this is done can be seen in listing 3.7, where a scheduler implementation is declared, and then in listing 3.8 where it is subsequently injected into a class and used.

```
@Bean
JobScheduler getJobScheduler(){
    return new FIFOJobScheduler(jobRepository);
}
```

Code snippet 3.7: Scheduler injection declaration

```
@Autowired
JobScheduler jobScheduler;
...
@Override
public ResponseEntity<JobFiles> getJobForDevice(UserCredentials
    userCredentials, DeviceId deviceId) {
    ...
    Optional<JobEntity> job = jobScheduler.assignJob(device);
    ...
}
```

Code snippet 3.8: Scheduler injection use

Using dependency injection to implement the scheduler, makes it easier to quickly swap scheduler implementations for the whole program, as well as run the same tests twice, but with another scheduler implementation. It should be noted that Spring handles the actual injection, which is the reason the declaration and injection are annotated with @Bean and @Autowired respectively. In order to make the schedulers interchangeable they all implement the same **Scheduler** interface which can be seen in code snippet 3.9. A scheduler should be able to find a job assignment given a device and possibly some filter for which jobs can be chosen. Additionally, the scheduler is able to determine, if a worker should continue a computation. The worker pings the server with some interval while computing to see, if it should stop. A reason for stopping could for example be, that the employer has deleted the job due to finding some error in the code. Additionally, the scheduler should have methods for determining whether test assignments should be used or whether the scheduler should trust a given worker or not. These two last methods are implementation details for the economic with trust scheduler which is described further in section 3.1.3.

```java
public interface JobScheduler {
    Optional<JobEntity> assignJob(DeviceEntity worker);
    Optional<JobEntity> assignJob(DeviceEntity worker, JobFilter jobFilter);
    boolean shouldContinue(long assignmentID);
    boolean usingTestAssignments();
    boolean shouldTrustDevice(DeviceEntity worker);
}
```

Code snippet 3.9: Scheduler interface

FIFO

The FIFO scheduler is implemented as seen in 3.10. the assignJob method uses a SQL query to sort the jobs in the database by upload date and returns the oldest job. Since the FIFO scheduler does not use the trust feature, it simply returns true on the *shouldTrustDevice()* method. The FIFO scheduler trusts all workers always. The *usingTestAssignments()* method also always returns false, since test assignments do not make sense without the trust features.

```java
public class FIFOJobScheduler implements JobScheduler{
    JobRepository jobRepository;
    @Override
    public synchronized Optional<JobEntity> assignJob(DeviceEntity device) {
        return jobRepository.getOldestAvailableJob();
    }
    @Override
    public boolean shouldContinue(long assignmentID) {
        boolean shouldContinue = true;
        Optional<AssignmentEntity> assignmentOpt = assignmentRepository.
    findById(assignmentID);
        // If the assignment is not present, it should not continue
        if(assignment.isPresent()){
            JobEntity job = assignment.get().job;
            // If the job is done, also quit
            if(job.getJobStatus() == JobEntity.JobStatus.DONE
                || job.getJobStatus() == DONE_CONFLICTING_RESULTS){
                shouldContinue = false;
            }
            // If should not continue, just mark as done
            if(!shouldContinue){
                assignment.get().setStatus(AssignmentEntity.Status.DONE);
                assignmentRepository.save(assignment.get());
            }
        } else {
            shouldContinue = false;
        }
        return shouldContinue;
    }
    public boolean usingTestAssignments(){return false;}
    public boolean shouldTrustDevice(DeviceEntity device) { return true;}
}
```

Code snippet 3.10: Scheduler injection use

Economic Scheduler

The economic scheduler and economic with trust schedulers described in section 2.5.2 are the same scheduler in the implementation, with the option to enable or disable the trust heuristic depending on the value of an argument in the constructor. This was done to avoid duplicate code, since the trust feature is an extension to the economic scheduler. In the code snippets 3.11 and 3.12 the implementation of the economic scheduler can be seen. Here we left out the trust part, which will be described in the next section. When the *assignJob()* method is called, the scheduler first tries to find a job in the database that belongs to the user the device is assigned to, i.e. a private job. This is done, since we assume that users trust their own workers. In case there is a private job the method will return it as a newJob object. In case no private job is found the scheduler moves on to assigning public jobs by using querying the database for the highest prioritized job, that the user has not yet contributed to. This is done by sorting the users by difference between CPU time contributed and CPU time spent as described in section 2.5.2.

```
1  public class EconomicScheduler implements JobScheduler {
2      JobRepository jobRepository;
3      AssignmentRepository assignmentRepository;
4      DeviceRepository deviceRepository;
5      boolean usingTrust;
6      ...
7      @Override
8      public Optional<JobEntity> assignJob(DeviceEntity device) {
9          Optional<JobEntity> jobFromSameUser = jobRepository.
       getOldestAvailableJobFromSameUser(device.getOwner().getUserName());
10         if(jobFromSameUser.isPresent()){
11             return jobFromSameUser;
12         }
13         // Look for other users jobs, that have the highest possible priority
14         Optional<JobEntity> jobFromOtherUsers = jobRepository.
       getJobWithHighestUserPriorityFromOtherUser(device.getOwner().getUserId());
15
16         return jobFromOtherUsers;
17     }
18 }
```

Code snippet 3.11: Scheduler Economic Scheduler pt1

The *ShouldContinue()* method seen in code snippet 3.12 returns a boolean that decides if a worker shall continue to work on the active job it has been assigned to, whenever it pings the server. First we take a precaution and check if the assignment exists in the database and in the case that it does not exist, the method will then change the shouldContinue variable to false. If it is still true, the scheduler then looks at the state of the job and in the case that the job status is *DONE*, the method returns false. A job can be set to done prematurely, if the employers worker returns a result, which is always considered correct. Finally, if the shouldContinue returns false it changes the state of the assignment in the database to *DONE*.

The *usingTestAssignments* method can be seen in 3.12. This method is used to determine if the scheduler should use the trust aspect of the scheduler. The *shouldTrustDevice* is used to decide if the worker given as an argument is trusted based on the method described in 2.5.2. If the scheduler deems the worker trustworthy the worker will receive a real assignment. If not, it will receive a test assignment as described in section 2.5.2.

```
@Override
public boolean shouldContinue(long assignmentID) {
    boolean shouldContinue = true;

    Optional<AssignmentEntity> assignmentOpt = assignmentRepository.findById(
    assignmentID);
    // If the assignment is not present, it should not continue
    // This should not possibly happen, but this is a precaution
    if(assignmentOpt.isPresent()){
        AssignmentEntity assignment = assignmentOpt.get();
        JobEntity job = assignment.job;

        // If the job is done, also quit
        if(job.getJobStatus() == JobEntity.JobStatus.DONE
        || job.getJobStatus() == JobEntity.JobStatus.DONE_CONFLICTING_RESULTS)
    {
            shouldContinue = false;
        }
        // If should not continue, just mark as done
        if(!shouldContinue){
            assignment.setStatus(AssignmentEntity.Status.DONE);
            assignmentRepository.save(assignment);
        }
    } else {
        shouldContinue = false;
    }
    return shouldContinue;
}

public boolean usingTestAssignments() { return this.usingTrust; }

public boolean shouldTrustDevice(DeviceEntity worker) {
    if(!usingTrust)  // If not using trust, trust anyone
        return true;

    double allDevicesAvgTrustScore = deviceRepository.getAvgTrustScore();
    return !(device.trustScore < allDevicesAvgTrustScore / 2);
}
```

Code snippet 3.12: Scheduler injection use

3.2 Employer Client

In this section we describe how the design of the employer client, described in section 2.6, is realized.

First we had to choose a framework for implementing the client. In order to achieve a high level of compatibility with different operating systems, we looked into frameworks for the Java Virtual Machine. This naturally leads to the native Java UI framework called JavaFX. JavaFX is, however, an older framework and comprehensive to work with compared to newer UI frameworks. Nevertheless, some newer frameworks build on top of JavaFX are available, e.g. TornadoFX[19]. TornadoFX maintains the high compatibility of JavaFX, since it compiles directly to Java and JavaFX. The compiled version can simply be wrapped in a .jar file, which can run on all system compatible with the Java Virtual Machine. Additionally, TornadoFX applications are written in the Kotlin language, which brings several language benefits, such as null safety. Furthermore, TornadoFX uses a higher level syntax for defining the GUI, which, in our opinions, allows for more readable and maintainable code as well as better separation of view and controller. For these reasons we chose to write the employer client using TornadoFX.

As mentioned in the design of the employer client in section 2.6, the employer client consists of two modules, i.e. the front-end module as well as the network module communicating with the server. These modules will be described in the following sections.

3.2.1 Front-end

A UI in TornadoFX is written in Kotlin using declarative UI. This makes the addition of new interface items fairly simple. We will here make a quick description of a part of the dashboard view containing the table view of all jobs. The code can be seen in code snippet 3.13. In TornadoFX the controllers and views are connected using injection, which uses the controller as a singleton and avoids problems with circular dependencies. For the **DashboardView** class two controllers as well as the UserModel are added using injection. In our implementation the controller is responsible for holding and mutating the model. This is done using the *Observable* classes in Kotlin. Variables declared with an observable class can take view items as observers, which allows for automatically updating the view, when the model changes. This can for example be seen in the tableview, where the tableview observes the observable list of jobs from the dashboard controller. In the bottom of the code, it can be seen which code runs when the view is loaded. This includes e.g. fetching and updating the list of jobs.

```
class DashboardView : View("Offloading Dashboard"){
    val user: UserModel by inject()
    val loginController: LoginController by inject()
    val dbController: DashboardController by inject()

    override val root = hbox(){
        setPrefSize(1000.0, 700.0)

        // Left side containing the tableview of jobs assigned
        hbox{
            alignment = Pos.TOP_LEFT
            paddingAll = 10.0

            tableview(dbController.jobs){
                minWidth = 500.0
                columnResizePolicy = CONSTRAINED_RESIZE_POLICY
                vboxConstraints {
                    vGrow = Priority.ALWAYS
                }
                readonlyColumn("Job", Job::name)
                readonlyColumn("Workers requested", Job::answersNeeded)
                readonlyColumn("Workers Assigned", Job::workersAssigned)
                readonlyColumn("Status", Job::status)
                readonlyColumn("Confidence level", Job::confidenceLevel)
                readonlyColumn("Upload time", Job::timestamp)

                contextMenu = ContextMenu().apply {
                    item("Delete"){
                        action {
                            selectedItem?.apply {
                                dbController.deleteJob(this)
                            }
                        }
                    }
                    item("Download Result"){
                        action {
                            selectedItem?.apply {
                                dbController.downloadResults(this)
                            }
                        }
                    }
                    item("Download Job File"){
                        action {
                            selectedItem?.apply {
                                dbController.downloadJobFiles(this)
                            }
                        }
                    }
                }
            }

        }
    ....
    override fun onDock() {
        dbController.fetchJobsForUser()
        super.onDock()
    }
}
```

Code snippet 3.13: Dashboard view code

The final design of the employer client can be seen in figure 3.1. The designs were based on the mockups created in the design phase in section 2.6.

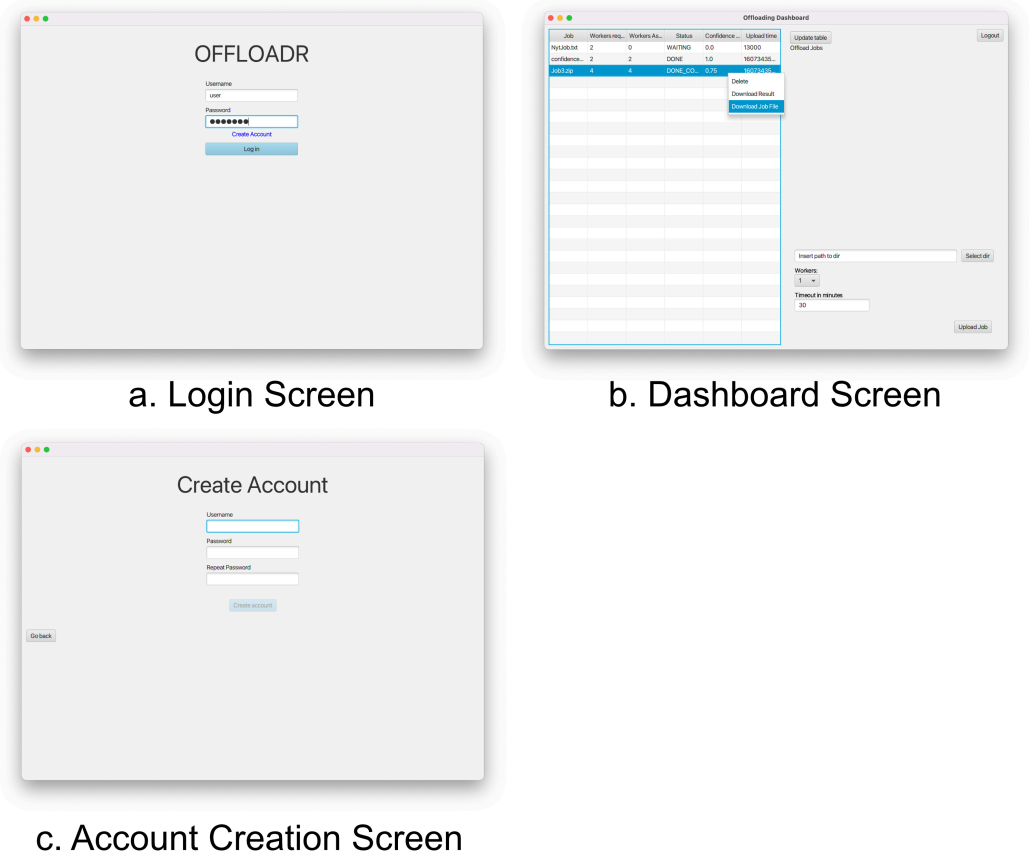


Figure 3.1: Final design of the employer client

The code from the **DashboardController** class that can be seen in code snippet 3.14 shows how we implemented the controllers for the views. Here it can be seen how the controller holds a list of observable jobs, and how this list is populated using the API. The API is described in detail in section 3.2.2. The view can call the function *fetchJobsForUser()*, which uses the API to get the jobs for the given user. After this the jobs are all parsed from JSON to the **Job** class. The implementation of this class is described further in section 3.2.2. Since the functions mutate an observable list, i.e. the job list, the view is automatically updated with the results from the API call. An MVVM pattern could have been applied here, but since all user information come from the same source, it would simply result in a wrapper class exposing the same methods as the API. If more resources from different sources were added, a **ModelView** class should be implemented. For now, however, we chose to simply use the JobAPI directly.

```
class DashboardController: Controller(){
    ...
    var jobs = observableListOf<Job>(listOf<Job>(
    ))
    var jobAPI: JobApi = JobApi();
    ...
    val adapter: JsonAdapter<Job> = Serializer.moshi.adapter(Job::class.java)

    fun fetchJobsForUser(){
        try{
            var jobNewList = getJobsForUserParsed(user.getCredentials())

            runLater{
                jobs.clear()
                jobs.addAll(jobNewList)
            }
        }
        catch (e: Exception){
            e.printStackTrace()
        }
    }

    private fun getJobsForUserParsed(userCredentials: UserCredentials):
    MutableList<Job>{
        try {
            var jsonResult = jobAPI.getJobsForUser(userCredentials)

            var newJobList: MutableList<Job> = mutableListOf()

            val length: Int = jsonResult.size

            for (i in 0 until length){
                var j: Job? = adapter.fromJsonValue(jsonResult[i])
                if(j != null){
                    newJobList.add(j)
                }
            }

            return newJobList
        }
        catch (e: Exception){
            println("Could not get jobs for user: $userCredentials")
            return mutableListOf()
        }
    }

    ...
}
```

Code snippet 3.14: Dashboard controller code

3.2.2 Network Module

Since the server exposes a Rest HTTP API, the employer client must be able to communicate with it using only HTTP requests. For implementing this we could have manually written HTTP requests using the build-in HTTP class in Kotlin. This would, however, necessitate manual code changes after every server update. This would make the employer client less maintainable. For this reason we looked into generating the client. This was also a natural choice, since we already generate much of the server using an OpenAPI specification, as seen in section 3.1. This means, that we already have the specification of the communication protocol in a .yaml formatted file (see appendix B).

OpenAPI-generator allows us to do client generation using the same specification. The problem was, that in the current version (4.3.1) of the Open API generator, the generated Kotlin version has several issues, like the inability to accept binary files sent in the body of an HTTP request. Luckily, on the 20th of November 2020 a new beta version (5.0.3-Beta) was released, which amended these issues. Unfortunately, the new beta version was not easily directly integrated in the project. For this reason the code generation was simply done using the command-line-interface version and the .yaml specification file.

Using the OpenAPI we can quickly generate a new and compatible HTTP Client and network module for the employer client upon changes in the server. The generator automatically generates an API class for each tags in our OpenAPI specification, in our case one for jobs, users and assignments. The classes then contain methods for wrapping the arguments in an HTTP request compatible with the format of the server. For example the *JobsAPI.getJobsForUser()* method simply takes the user credentials, and then handles everything regarding the HTTP request itself automatically.

In addition to generating the APIs, the OpenAPI generator can also generate the model classes, in our case classes like Job, jobresult, user credentials etc. It is also one of these classes, the job class, that is directly used in the table view in the DashBoardView and DashBoardController classes, as seen in the previous subsection.

3.3 Worker Client

In section 1.1.1 we found that Termux could emulate a bash terminal for the worker, with a Python interpreter included. With this we can make the worker run jobs as python programs. The design of the application in the following sections are based on the mockups seen in section 2.7.

3.3.1 Termux

The Termux android app, as described in section 1.1.1, allows us to interpret program written in python2 and python3. The Termux project is open source and the code is available on their GitHub repository[20]. To serve our purpose, we forked the Termux app and modified the terminal to run as a headless background task.

We then created an interface for bridging Termux to our android app. This bridge component defines a CommandBundle and a function for enqueuing command bundles to be executed in the terminal, see code snippet 3.15. A **CommandBundle** consist of a command string that is typed in to the terminal, and a function that consumes the terminal output when the command has finished executing. The TermuxBridge is responsible for sequentially executing commands from the command queue and calling the associated consumer function when the execution finishes.

When starting the application, a sequence of setup commands are enqueued. These commands include initializing necessary file directories and installing Python interpreters.

```
1
2  public class TermuxBridge {
3
4      public static class CommandBundle {
5          private final String commandText;
6          private final Consumer<String> onExecuted;
7
8          public CommandBundle(String commandText, Consumer<String>
       outputConsumer) { ... }
9      }
10
11     private final ConcurrentLinkedQueue<CommandBundle> commandQueue = ...;
12
13     public void enqueueCommand(CommandBundle commandBundle) { ... }
14 }
```

Code snippet 3.15: TermuxBridge

3.3.2 Front-end

As described in the requirements in section 2.2, "The user must be able to enlist a mobile phone as a worker". To connect the client and the worker phones, the same username must be used. To do this on the mobile phone, a login screen is made.

In figure 3.2 the job overview can be seen. This is the first screen a worker meets after logging in. Here the worker can see if it is actively looking for a job or already running one, or inactive. it is also possible for the worker to stop looking for jobs, or running a job by pressing the *DEACTIVATE* button. They can log out using the *logout* button or going to the settings by pressing the button with the gears. This meets the requirement from section 2.2: "The user should be able to stop a job running on a mobile phone at any time".

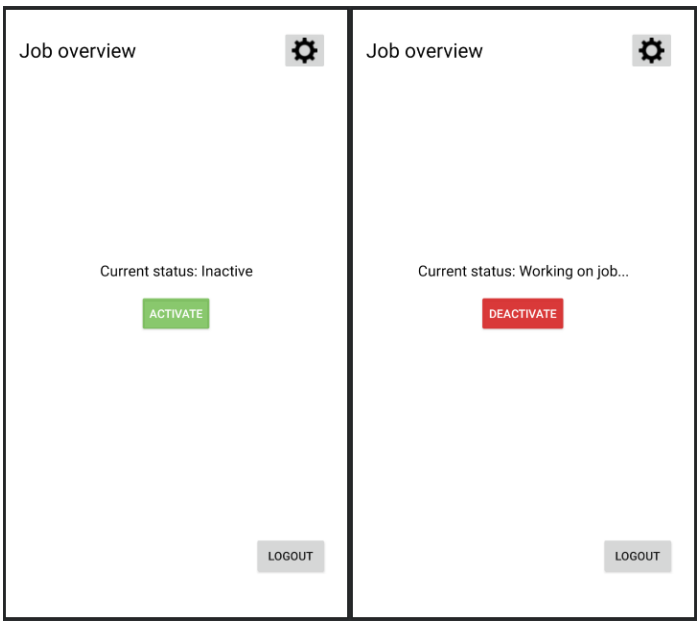


Figure 3.2: Job overview for mobile phones

A requirement also from section 2.2 says: "A worker phone should be usable while a job is running on the mobile phone." Because of this, the Termux app needs a front-end with a settings menu. This menu can be seen in figure 3.3. These settings can be accessed through the login.

With the settings the user can decide when the phone can be used for jobs. The user can then modify the settings for the WiFi, power and time of day. The WiFi and Power buttons can be cycled by clicking to either *ON*, *OFF* or *ALWAYS* which respectively means that the phone can be used either when the WiFi/power is on

and connected, off, or whenever. On the left side of figure 3.3 it can be seen that it possible, but not necessary, to add a timeframe. If no timeframe is specified the phone is marked as available at all times, if the other settings allow it.

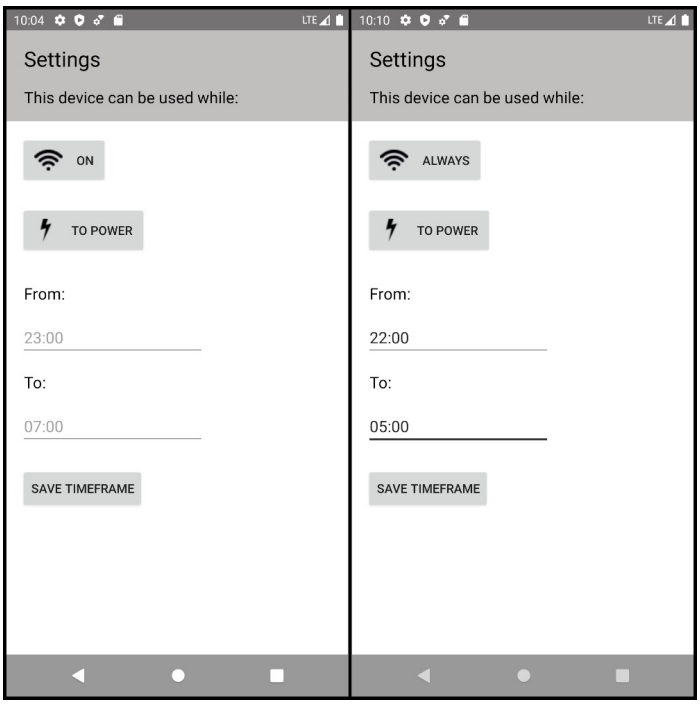


Figure 3.3: Settings screen for mobile phones

SharedPreferences is used to save these preferences even when the app is closed. This makes it possible for Termux to check if the job should still be running, when it is running as a background job.

A code snippet of this can be seen in code 3.16. In this code snippet the prefEditor and savedPrefs methods can be seen. These methods return objects of the editor and saved values of the type SharedPreferences in the system. With these objects preferences can be saved and acquired, even when the app has been closed.

```
1
2  public class Preferences {
3
4      {...}
5
6      public static SharedPreferences.Editor prefEditor(Context appCon){
7          SharedPreferences saved_values = PreferenceManager.
       getDefaultSharedPreferences(appCon);
8          SharedPreferences.Editor editor = saved_values.edit();
9          return editor;
10     }
11
12     public static SharedPreferences savedPrefs(Context appCon){
13         return PreferenceManager.getDefaultSharedPreferences(appCon);
14     }
15
16     {...}
17 }
```

Code snippet 3.16: Preferences

3.3.3 Network module

The mobile app uses HTTP requests for communication medium with the server. The design of this medium can be seen in section 2.7.

The API interface to the server is generated using the same OpenAPI specification as used for the server. This provides us with a API client in the Kotlin language, which contains methods for getting jobs etc. This code and approach is identical to the one used for the employer client network module described in section 3.2.2. For this reason we do not go into further detail, but instead refer the reader to section 3.2.2.

3.4 Testing

In this section we will describe how the implementation was tested. We started out by planning what parts of the application to focus on in testing. We wanted to test the schedulers, controllers, the file manager and JPA repositories. The schedulers, JPA repositories and the file manager consist of fairly small and simple functions, and for this reason we chose to unit test these. The controllers, however, interact with so much of the code from the entire system, that a bigger integration test was needed for this. The bigger integration test was done in conjunction with the performance tests, since the performance test already depend on the controllers to work properly. The performance tests produces a lot of data regarding each run, which we then analyzed, see chapter 4, to see if the controllers appeared to

work correctly. While running the performance tests, we also measured code coverage. For the performance test with the economic scheduler we managed above 88% code coverage for the scheduler, about 60% for all controllers and 100% for the getConfidenceLevel from the file manager. For the employer client and the worker client we tested in through the experiment, where we used the employer client to upload a job, which is then managed by the server and processed by a worker. This experiment was done to test the performance of the system and can be seen in section 4.2. In the rest of this section, we describe the unit tests conducted for the JPA repositories, file manager and schedulers.

3.4.1 JPA Tests

In order to test that our custom queries for our JPA repositories work, we created some tests. An example of this can be seen in code snippet 3.17. The query is supposed to get the highest priority job with a different user than the one requesting a job, and that does not have enough workers processing it already. Priority is in this query measured in terms of banked CPU time, i.e. CPU time contributed subtracted the CPU time spent. First we create three users, with the user asking for a job being the one with the highest banked CPU time. Next we have a user with slightly lower banked CPU time and then one with negative banked CPU time. Now we save an unfinished job for each of the three users. Finally, we assert, that when the query is called, we do in fact get a result, that this result is a job from the highest priority user, and that the job was not uploaded by the same user that made the request.

```
@Test
void jobRepositoryTest01(){
    // Test for getJobWithHighestUserPriorityFromOtherUser query

    // Create users
    UserEntity thisUser = new UserEntity("this", "secret");
    thisUser.setCpuTimeContributedInMillis(1000); thisUser.
    setCpuTimeSpentInMillis(0);
    UserEntity otherUserGood = new UserEntity("otherGood", "secret");
    otherUserGood.setCpuTimeContributedInMillis(500); otherUserGood.
    setCpuTimeSpentInMillis(0);
    UserEntity otherUserBad = new UserEntity("otherBad", "secret");
    otherUserBad.setCpuTimeContributedInMillis(0); otherUserGood.
    setCpuTimeSpentInMillis(100);
    userRepository.saveAll(Arrays.asList(thisUser, otherUserBad, otherUserGood
    ));

    // Create jobs
    JobEntity thisUserJob = new JobEntity(thisUser, "thisPath", "thisJob", 3,
    60);
    JobEntity otherUserGoodJob = new JobEntity(otherUserGood, "otherGoodPath",
     "otherGoodJob", 3, 60);
    JobEntity otherUserBadJob = new JobEntity(otherUserBad, "otherBadPath", "
    otherBadJob",3, 60);
    jobRepository.saveAll(Arrays.asList(thisUserJob, otherUserGoodJob,
    otherUserBadJob));

    // Use methods, that's being tested
    Optional<JobEntity> jobOpt = jobRepository.
    getJobWithHighestUserPriorityFromOtherUser(thisUser.getUserId());

    // Assert that the answer is correct
    assertTrue(jobOpt.isPresent());

    JobEntity job = jobOpt.get();
    assertEquals(otherUserGoodJob.getJobId(), job.getJobId());
}
```

Code snippet 3.17: A test for the job repository's custom queries

3.4.2 Scheduler Test

Several scheduler tests were made to check, if the schedulers do in fact return the correct jobs given some predefined entries in the database. An example of this can be seen in the code snippet 3.18. Similar tests are written for the economic with trust scheduler as well as the FIFO scheduler. This is a test for the economic scheduler, that verifies, that it returns job with the highest priority, i.e. a job from the set of users with the most banked CPU time. To verify this, two users were made, one with 0 banked CPU time, the badUser, and one with 100 banked CPU time, the goodUser. Both users then upload a job. A worker from a third user then requests a job, and should get the job uploaded by the goodUser.

```java
@Test
void economicSchedulerTest01() {
    // Create users
    UserEntity badUser = new UserEntity("user1", "secret");
    badUser.setCpuTimeContributedInMillis(0);
    badUser.setCpuTimeSpentInMillis(0);
    UserEntity goodUser = new UserEntity("user2", "secret");
    goodUser.setCpuTimeSpentInMillis(0);
    goodUser.setCpuTimeContributedInMillis(100);
    UserEntity workerUser = new UserEntity("user3", "secret");
    userRepository.saveAll(Arrays.asList(workerUser, badUser, goodUser));

    // Create jobs in system
    JobEntity goodUserJob = new JobEntity(goodUser, "somepath", "goodJob",
    3,60);
    JobEntity badUserJob = new JobEntity(badUser, "someOtherPath", "badJob",
    3, 60);
    jobRepository.saveAll(Arrays.asList(goodUserJob, badUserJob));

    // Create worker asking for job
    DeviceEntity pollingWorker = new DeviceEntity(workerUser, "007");
    deviceRepository.save(pollingWorker);

    // Get job from scheduler
    Optional<JobEntity> jobOpt = scheduler.assignJob(pollingWorker);

    // Assert that a job is present, and that the employer is the goodUser
    with most banked CPU time
    assertTrue(jobOpt.isPresent());
    JobEntity job = jobOpt.get();
    assertEquals(goodUser.getUserId(), job.employer.getUserId());
}
```

Code snippet 3.18: Test for asserting that the economic scheduler chooses the correct user

3.4.3 Result Files Integrity Tests

In section 3.1 we describe the implementation of an algorithm to calculate our confidence level in a result calculated for a job. This is very important in the effort of ensuring the correctness of results computed by the workers and should thus be tested. After all tests, the test data files are deleted and reset to the starting point to ensure that the tests do not influence one another.

In code snippet 3.19 a test for confidence level can be seen. In this case two different folders containing files with the same name and meta data. One text file, however, contains some additional characters. This should cause the algorithm to differentiate between them. Since only two result files are present, this should result in a confidence level of 0.5, which is indeed the case.

```
@Test
public void getConfidenceLevelTest01(){
    // Result dir to put the zipped files
    String resultDir = pathResolver.generateNewResultFolder(pathToWorkingDir);

    // Get File handle for test files
    File folderToZip = new File(pathToStartingData + File.separator + "
    identical");
    File folderToZip2 = new File(pathToStartingData + File.separator + "
    notidentical");

    File zipFile1 = new File(resultDir + File.separator + "zipfile1.zip");
    File zipFile2 = new File(resultDir + File.separator + "zipfile2.zip");

    // Zip both files
    FileUtilsKt.zipDir(folderToZip.getAbsolutePath(), zipFile1.getAbsolutePath
    ());
    FileUtilsKt.zipDir(folderToZip2.getAbsolutePath(), zipFile2.
    getAbsolutePath());

    ArrayList<File> zipFiles = new ArrayList<>();
    zipFiles.add(zipFile1);
    zipFiles.add(zipFile2);

    Pair<File, Double> result = FileUtilsKt.getConfidenceLevel(zipFiles);

    double delta = 0.001;

    assertTrue(Math.abs(result.getValue() - 0.5) < delta);
}
```

Code snippet 3.19: Test for calculating confidence level with two different results

The second case is tested in code snippet 3.20. In this test, the contents of the files inside the result zip file are actually identical. This means, that the algorithm should return a confidence level of 1.0, since both results should be considered equal. This test also passes.

```
@Test
public void getConfidenceLevelTest02(){
    // Result dir to put the zipped files
    String resultDir = pathResolver.generateNewResultFolder(pathToWorkingDir);

    // Get File handle for test files
    File folderToZip = new File(pathToStartingData + File.separator + "
    identical");
    File folderToZip2 = new File(pathToStartingData + File.separator + "
    identical");

    File zipFile1 = new File(resultDir + File.separator + "zipfile1.zip");
    File zipFile2 = new File(resultDir + File.separator + "zipfile2.zip");

    // Zip both files
    FileUtilsKt.zipDir(folderToZip.getAbsolutePath(), zipFile1.getAbsolutePath
    ());
    FileUtilsKt.zipDir(folderToZip2.getAbsolutePath(), zipFile2.
    getAbsolutePath());

    ArrayList<File> zipFiles = new ArrayList<>();
    zipFiles.add(zipFile1);
    zipFiles.add(zipFile2);

    Pair<File, Double> result = FileUtilsKt.getConfidenceLevel(zipFiles);

    double delta = 0.001;

    assertTrue(Math.abs(result.getValue() - 1.0) < delta);
}
```

Code snippet 3.20: Test for confidence level with two identical results

Chapter 4

Experiments and Results

In this chapter we will present the experiments conducted to examine the behavior of the schedulers. We also describe the performance test done on the system, to evaluate the effectiveness of uploading a series of job to the system versus computing them locally.

4.1 Scheduler Experiments

In this section we will describe the experiments conducted regarding the scheduling of jobs for the workers. These include experiments regarding confidence level, correctness and economic incentives.

4.1.1 Simulated Experiment Setup

Testing the performance of the schedulers requires a large amount of users to be using the system simultaneously. Since we do not have access to a large number of physical devices, we instead set up a number of simulated clients. These clients are created as a part of a Spring Boot test in the server program, meaning they can interact with the server controllers directly instead of going through an http request. This allows us to run a large number of clients, with relatively small overhead. This does, however, also present a worst case scenario, as the delays incurred by network communication are eliminated, and the server is immediately flooded by a large number of requests.

When generating fake users, we mark a fraction of them to be malicious, meaning that all workers owned by that user will act as malicious workers, which return incorrect results. The simulated employer client has a JobSpawner property which

determines how they upload jobs over time. The JobSpawner also determines the amount of answers requested and the expected compute time for each job. This expected compute time determines how long it will take a simulated worker client to 'compute' the job.

To realistically imitate worker clients without using too much CPU power, all simulated workers simply wait a fixed amount of time after receiving an assignment, and then return a single text file as the result. Malicious workers will write a random number between 1-9 in the text file, while non-malicious workers will write 0 in the text file. This allows us to differentiate between correct and incorrect results when collecting statistics. Additionally, it allows for the malicious user to randomly coordinate results in the sense, that they send the same wrong result, which further increases the difficulty in finding the correct result. Each simulated worker also has a CPU factor between 0.5 and 2.0 that determines the speed of which they 'compute' a job. This factor is multiplied on to the expected job time provided by the employer, meaning a worker with CPU factor 2.0 will wait 4 times longer before returning a result, than a worker with CPU factor 0.5. This simulates the difference in computational power among clients.

We use a UserBase to represent a collection of simulated users, workers and employers used for a statistical test. A UserBase can be configured to match a certain testing scenario, for example a UserBase might have a large amount of employers relative to employers, and thereby represent a scenario where the system is flooded by jobs faster than they can be computed. Before each of the following tests, we briefly describe the UserBase created for that test.

4.1.2 Confidence test with 40% malicious workers

For the first test, the UserBase has 140 users, where 60 of them has employer clients and the remaining 80 users have one worker client each. Out of the 80 worker clients, 32 (40%) are malicious. Each employer spawns a job approximately every four seconds, and each job has takes three seconds to compute on average. All jobs uploaded by the employers request four answers.

The result can be seen on figure 4.1 where the x-axis shows intervals of job completions, i.e. 1-20 referring to the first 20 jobs to be completed. The y-axis shows the confidence level from 0.0 to 1.0. We can see that the first 20 jobs have the same confidence level with both of the schedulers. This is because the system has not yet received results from any workers and the trust scheduler can therefore not determine a trust level for the workers yet. After the system has received 20 results it starts to be able to determine the trust level of some workers.

If we look at the graph we can see that the FIFO scheduler is generally steady in its confidence level. This is due to the fact that it does not differentiate workers on their trust score, but only between the age of the jobs. The small variations depends on how many malicious workers happened to be assigned together. Since there is a surplus of honest workers they are most likely to be a part of the majority for each assignments. For the Economic with trust scheduler we see a bigger change per 20 job. This is due to the economic scheduler not using workers with a trust score lower than the threshold for real assignments, and these untrusted workers simply get a test assignment, as described in section 2.5.2.

Based on the graph in figure 4.1 we can see that the economic scheduler with trust scheduler does impact the average confidence level, i.e. the validity of the results, compared to the FIFO scheduler. Having a high confidence is, however, not worth much, if the actual results getting through our system are still wrong. In this metric the economic with trust scheduler also fared better with 37 out of the 200 job results being wrong. The FIFO scheduler allowed 72 wrong results through out of the 200 job results. This means, that our confidence level is actually lower, than the actual ratio of correct results out of total results, which indicates that our confidence level might be a conservative measure of result validity. In this case the confidence level was on average about 76% for the economic with trust scheduler, where the 82% of the results were correct. Using the FIFO scheduler it was 61% and 64%, respectively. The trust scheduler manages to reduce the wrong results by almost 50% in this worst case scenario.

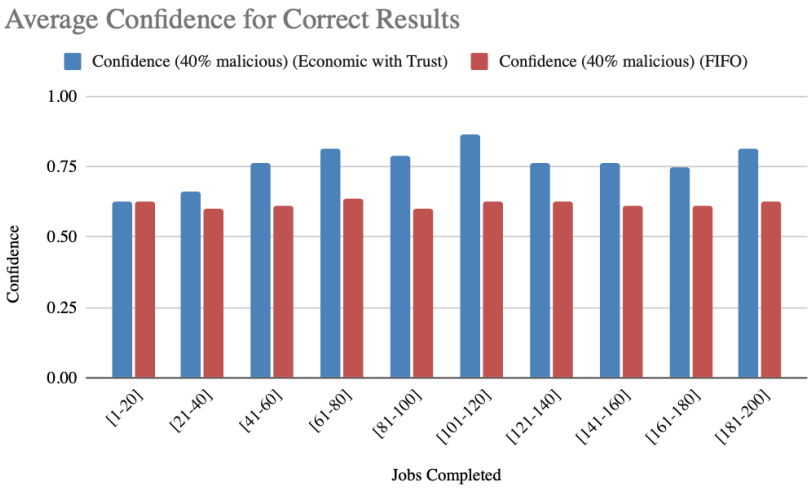


Figure 4.1: Confidence test with 40% malicious workers

The results get even more interesting when looking at the jobs from job number 200 and onward to the final job finished within the 1 minute test. The results for this can be seen in table 4.1. The economic with trust scheduler only manages 245 jobs in total. But out of the last 45 jobs, only 3 are wrong i.e. 93% correctness. The economic with trust scheduler increases its correctness ratio due to choking the malicious users. The FIFO scheduler, on the other hand, keeps approximately the same correctness as before. The choking of the malicious users does, however, reduce the number of completed jobs significantly, from 539 to 245. The increasing correctness ratio behavior of the economic with trust scheduler will be further explored in section 4.1.4.

	Economic Trust Scheduler	**FIFO scheduler**
Total jobs finished	245	539
Total incorrect results	40	212
Total Correctness ratio	83.6%	60.6%
Correctness ratio after 200th job	93%	58.7%

Table 4.1: Worst case comparison of Economic with Trust Scheduler and FIFO Scheduler

4.1.3 Confidence test with 5% malicious users

In order to test a more realistic scenario compared to the 40% malicious users test, we conduct a test with only 5% malicious users. We still consider this number to be quite high, but not a worst case. The test conditions are exactly the same as in the test with 40% malicious workers, see section 4.1.2, except that the amount of malicious users is 5% instead of 40%.

The results can be seen in figure 4.1 with an x-axis showing intervals of job completions, i.e. 1-20 referring to the first 20 jobs to be completed. The y-axis shows the confidence level from 0.0 to 1.0. The difference between the FIFO scheduler and the economic with trust scheduler is smaller than the test with 40% malicious users. There is, however, still a difference. They start out with the same confidence level of about 95%. The economic with trust scheduler, however, manages to have a higher confidence level for all other intervals. The economic with trust scheduler even manages to have an average confidence level of 1.0, i.e. completely confident, in four out of the ten intervals.

When comparing the actual correctness of the results with the confidence level, we see a similar picture as the one for the 40% malicious users experiment, see section 4.1.2. The confidence level is again lower or equal to the actual correctness ratio. The economic with trust scheduler fared better with 0 out of 200 job results

being wrong, i.e. no errors at all. The FIFO scheduler allowed 2 out of 200 wrong results through in this setup.

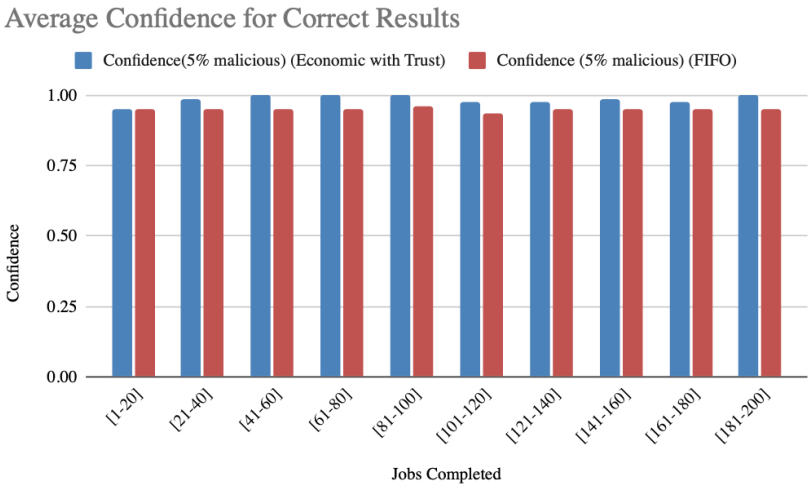


Figure 4.2: Confidence test with 5% malicious workers

4.1.4 Correctness Ratio Experiments

In this section we want to test if the choice of scheduler influences the correctness of the result. We also want to verify if there is a correlation between our calculated confidence level and the actual correctness of the result.

For this experiment we ran the test three times for each scheduler to average out variations in individual runs. We setup the environment with 80 workers and 60 employers. In order to ensure, that no jobs will be finished using the users own workers, no worker belongs to the same user as an employer. This makes the test a worst case scenario, since a user using its own worker induces perfect confidence levels for the specific job. For this test we again use 40% malicious users, i.e. about 32 of the workers always send false results back. The jobs are set to take on average 3 seconds and 3 answers from workers are needed for each job. The employers spawn a new job every 4 seconds. The test is run for 3 minutes for every run. The results can be seen in figure 4.3. In the x-axis we see the finished job intervals. The first being the correctness of the first 75 jobs. On the y-axis we see the correctness ratio and confidence level, which is a number between 0 and 1.0. The economic with trust scheduler manages significantly higher correctness ratio even within the first 75 finished jobs. The FIFO and regular economic scheduler manage about 60% correct answers. After the first 75 finished jobs, the economic with trust reaches a correctness ratio of 94%, while the others stay about 60%. This

is due to the economic with trust scheduler identifying most malicious users and ignoring their results. For the rest of the test all schedulers stay about the same. This is also expected, since the trust scheduler does not improve after it has already identified the malicious users. The other schedulers does not take trust into account and should therefor not improve outside of random variations. Since the economic with trust scheduler does not reach 100%, it has not identified all malicious users. The confidence ratio of all the schedulers also stay about the same. For the schedulers without trust the confidence level is a fairly precise estimate of the correctness, where as for the economic with trust it is a conservative estimate of the correctness. The economic with trust scheduler lines ends earlier than the others due to it not finishing as many jobs in the 3 minute timeframe. This is because the economic with trust scheduler uses test assignments for untrusted workers, which in a sense wastes compute power. This wasted compute power would, however, likely have returned wrong results anyway. In the graph the schedulers without trust vary quite a bit from about 50% to 80% correctness ratio for the different intervals. This is even the case with these three runs. The results varied even more in the individual runs, where the result for individual intervals vary from 32% to 89% for the FIFO scheduler. This is not visible in the graph, however. The trust scheduler did not show this behavior and was significantly more stable in its result. For the economic with trust scheduler the worst case for any interval was 83% with the best being 100%.

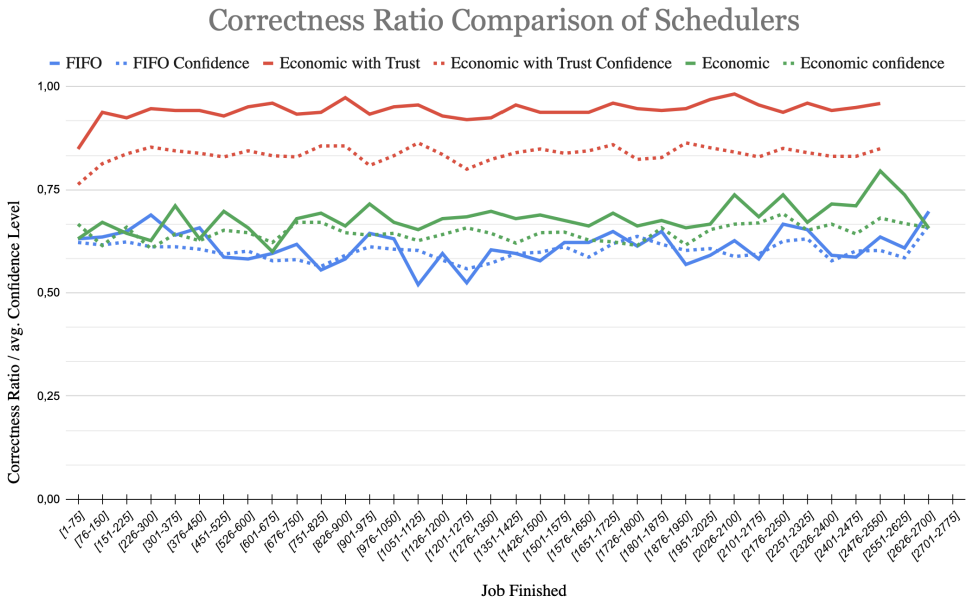


Figure 4.3: Comparison of schedulers for correctness and confidence

4.1.5 Economic Incentive Experiments

The Economic scheduler, described in section 2.5.2, prioritizes users with more banked processing time than other users. The following experiments showcase how the current economic scheduler implementation rewards users for participating as workers.

The UserBase used for this experiment consists of 50 employers and 70 workers. Like previously, the users with employers do not have any workers of their own in the system. This represent a scenario where the employers must entirely rely on other users, and thus clearly highlights the prioritization policy used by the scheduler. Each employer is given an amount of banked processing time before starting the experiment. The first employer is given zero seconds of banked time, the second is given ten seconds, the third twenty, and so on.

The jobs spawned by the employers have an average compute time of four seconds, and the employer requests 3 answers. On average the employer uploads a new job every four seconds. The low amount of workers relative to employers, and high spawn rate of jobs, results in the system being flooded by a large amount of jobs. The data was collected over a duration of 240 seconds.

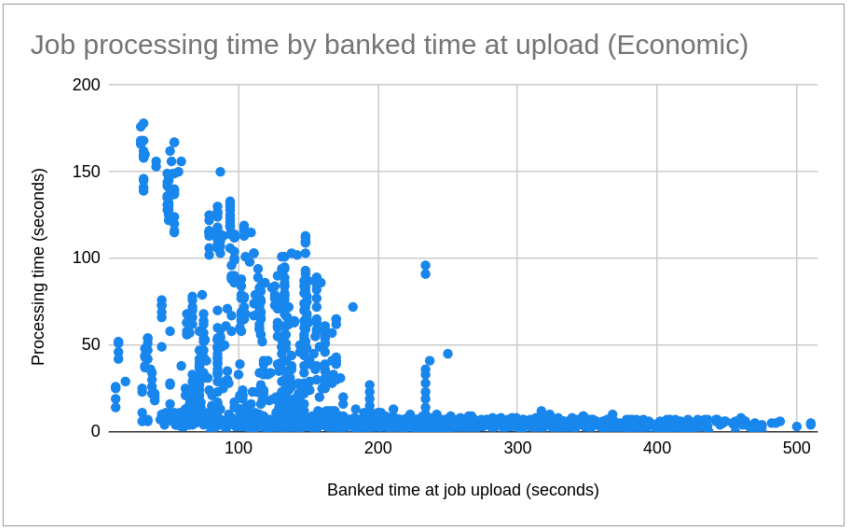


Figure 4.4: Processing time by banked time at the time of upload for the Economic scheduler

Figure 4.4 shows the amount of seconds from a job was uploaded until the result was ready on the y-axis, and on the x-axis we see the amount of banked time held by the user at the time of uploading the job. We can see that at a certain threshold, around 200-230 seconds of banked time, all uploaded jobs are completed

within about 15 seconds of uploading the job. Before this point, it is generally the tendency that a lower amount of banked time equates to potentially slower job completion time.

As previously mentioned, each job takes 3 workers to complete. Assuming one job per employer, and 70 employers, then the top $(70/3) \approx 23$ employers could potentially seize all of the available workers. The employer that has 23rd most banked time starts with *(50 - 23 - 1) * 10 seconds = 260 seconds* of banked time. This can explain the flat line after the 260 mark. This pattern of low job time extends all the way down to around 200 seconds, which could be explained by the users losing banked as time progresses, and so the threshold at which a user is in top 23 employers is also lowered.

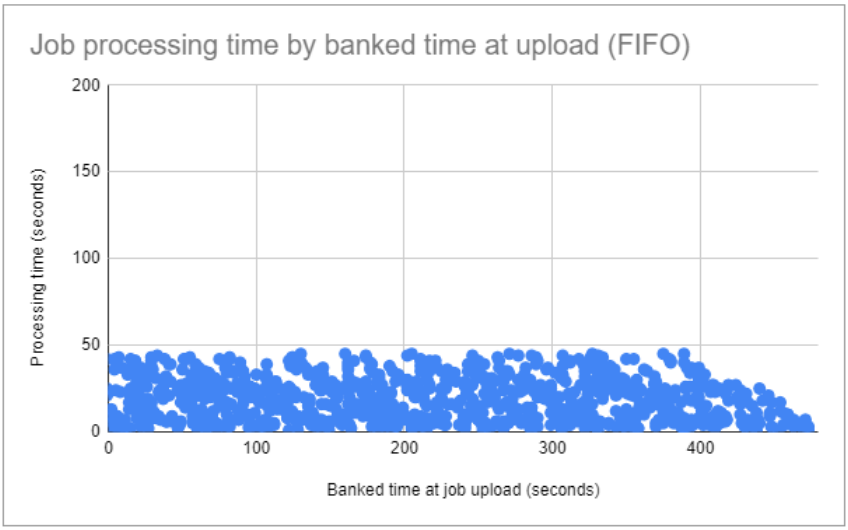


Figure 4.5: Processing time by banked time at the time of upload for the FIFO scheduler

Figure 4.5 shows data for the same scenario, but using the FIFO scheduler instead of the Economic scheduler. Here we see that the wait time is independent of banked time, as expected. The slope in the data from 400 to 500 seconds of banked time can be explained by the fact that the banked time of the users decrease over time, and so most jobs in this range are posted early on in the experiment, where the amount of jobs in the system is low. It important to note that these two graphs only show points for jobs that were completed, and therefore a lot of uploaded jobs are not shown. In figure 4.6 we see the percentage of incomplete jobs, grouped by the users banked time when the jobs were uploaded. Here we very clearly see that the Economic scheduler heavily favors users with a lot of banked time, while the FIFO scheduler ignores this measure.

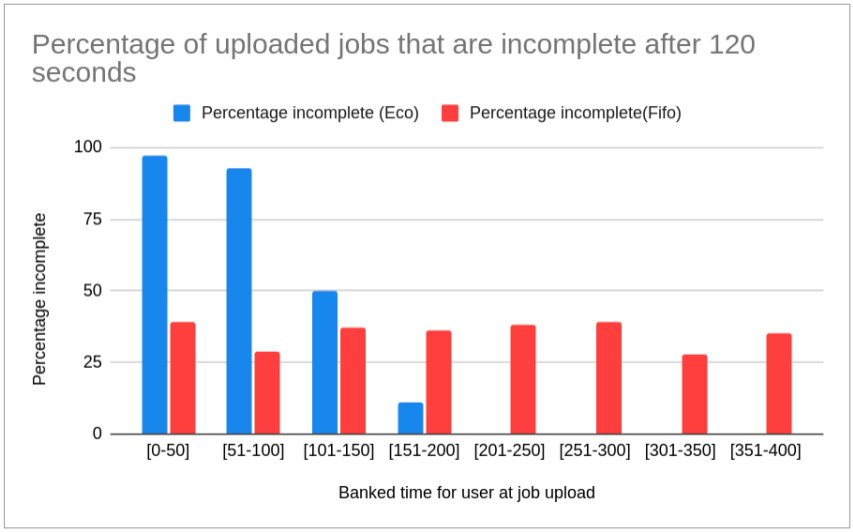


Figure 4.6: Percentage of jobs that has not been completed after upload grouped by range of banked time at time of upload

If we wanted to have a more equal distribution for the Economic scheduler, we could consider factoring the wait time into the prioritization, so that employers who have waited a long time will eventually be prioritized higher than somebody who has more banked processing time, but has waited less time.

4.2 System Performance Test

These tests are made to show the difference in running a job on a limited employer device, several times in parallel, and offloading the same job to several worker devices. The employer and worker devices used for this test can be seen in table 4.2, where the Raspberry Pi is the employer and the OnePlus 6 is the worker. These tests were conducted with a test job that takes approximately 30 seconds to run on the Raspberry Pi 4. The test job code can be seen in appendix C.

This test was conducted with the Raspberry Pi seen in table 4.2. Ten instances of the same job was run at parallel 8 times. The average of these runs was 108.24 seconds, with a maximum deviation of 41.64 seconds. The fastest run was 86.6 seconds and the slowest was 126.68 seconds. This deviation can be explained by the Raspberry Pi getting slower for each time the jobs are run, as the temperature rises and its CPU down clocks.

Raspberry Pi 4	**OnePlus 6**
Model B	Qualcomm Snapdragon 845
2GB RAM	8GB

Table 4.2: Hardware Specifications of the test devices

As the next part of the test, the same job was offloaded to run five times consecutively on a OnePlus mobile phone, the specs can be seen in table 4.2. The average time from a job requested to completed for the mobile phone was 17.89 seconds with a maximum deviation of 0.5 seconds.

This test shows that even if the Raspberry Pi is cooled before running the program it still takes 86.6 seconds, as opposed to sending the job through our system where the fastest time could be under 20 seconds if 10 workers get the job instantly. Some overhead would, however, be induced for distributing the job to workers and retrieving the result, but assuming that this takes less than 65 seconds, it would still be at least equally fast. The devices used for this test may, however, not represent the devices being used in a real world scenario of our system. The devices do, however, showcase that some desktop computers, here the Raspberry Pi 4, are slow enough that it can be faster to offload to a mobile phone. Another thing to notice here is, that the Raspberry Pi heated up and became very unresponsive while running the tests, which yielded it unusable for the duration of the test. This is another reason why it might be advantageous to offload the computation some other device, in this case our system, especially if the computation a long time. Ideally, we would have liked to actually offload to more phones, but we were unable to obtain ten modern Android phones. Even the Oneplus 6 is several years old by now. This test does, however, still give an indication of the possibilities of our approach.

Chapter 5

Reflections

This chapter will contain considerations about the project, where some of the design and implementation choices we made are considered. Furthermore, this chapter also features the conclusion to the project, as well as any future work that could not be included in the project due to time and/or scope constraints.

5.1 Discussion

This section is composed of a more critical view of the decisions we made. It is put here, at the end of the report, rather than one of the earlier sections, as it is a reflection on our choices, rather than a justification.

OpenAPI Tools

The only OpenAPI tool used in our project is the OpenAPI-generator for generating our controller interfaces and objects. Considering that the OpenAPI-specification is just that, a specification, we could also have set up a tool that checks whether our implementation lives up to our specification, and added it to our build pipeline through Maven. This includes mock servers and clients, so we could have tested both our client and server implementations in this manner. The benefit of doing this is making testing automated, which is never a bad safety property to have. The downside is, that this would take extra time to actually implement.

There's also the issue of these tools not necessarily being available for Maven, so we would also have to implement these as a Maven plugin. The time investment needed to implement one or more of these tools into our pipeline would not be worth the advantage of having them on a project this small in size, as any

testing we need could be done manually instead. If this project were to grow continuously, however, having this sort of automated test would be invaluable, as it allows us, the developers, to rapidly test whether the changes we've made live up to our specification, which itself can be changed rapidly.

Execution Language

In our implementation, uploaded jobs must be specified as python programs, as it was quite easy to find an interpreter for the mobile workers. For a proof-of-concept project, this is fine, but using Python has some implications.

First and foremost, we have no way to ensure that the programs provided by the employers will actually terminate. This could potentially be a very big flaw, as it would allow a malicious employer to make a job with an infinite loop, and then request a large amount of workers to just deny the processing of any other job. A potential solution to this problem could be a job timeout, and this will be considered more in detail in the future work section, see section 5.3.2. Alternatively, a language that is not Turing complete could be considered as a domain specific language, specifically for making remote executions. This, however, comes with the downside of not being able to model all problems that is possible to model in a Turing complete language.

Secondly, dependency handling is going to be somewhat complicated to do. Either everything has to be packaged statically in the job folder, or a job should ship a list of dependencies and their sources to be downloaded independently of the server.

Neither case is particularly favorable, though packaging dependencies statically is likely the more dependable solution, as it does not rely on any third parties.

5.2 Reflections on the process

As a whole this project went well. We did, however, have have some problems for some sprints, where we did not reach our goals. Our version of the scrum work process, including sprints, can be seen in chapter 6. One example of us having difficulties reaching a goal was for example, when we were researching how to hardware accelerate programs on mobile phones. This ended up being unfeasible within the scope of this project, due to the tools being very early in their development. Just getting the hardware acceleration tools to run was very difficult, and

we also did not have the hardware in terms of mobile phones, to test it. Additionally, the sprint goals regarding authentication ended up taking two weekly sprints without amounting to any progress. For this reason we chose to continue without it. The reason for this was difficulties with combining the authentication in Spring with the code generated by OpenAPI. In general most of our problems were in the middle of the process, i.e. from October to the middle of November. We started out by fairly quickly defining our problem and then got to work. The last sprints from November to December went well and we progressed quickly.

Another limitation on our process was the lack of hardware. For a real test for a system like this, we would ideally have had many mobile phones to do a proper system test. We did, however, somewhat remedy this by creating the entire mock system for the performance tests, described in section 4.2.

We also had difficulties finding good working hours for the project. One of the members of the group did not follow the same courses as the rest, which resulted in us having only a few hours a week with all members present. Additionally, the courses on this semester had many mini projects and lectures. This further reduced the time available for the project. Sometimes we would make a sprint goal a week ahead, but within that week we would only have about eight hours work time, which was not enough to realistically reach the goal. And sometimes these eight hours would be spread over several days and would not be the same hours for all group members. We tried to remedy this by extending the sprints to two weeks in the middle of the semester, which helped a bit. From the end of November, however, we returned to one week sprints.

5.3 Future Works

In this section we will discuss parts of the program that we would like to implement into the system in the future.

5.3.1 Job management

In our implementation it is possible to specify a *timeout*, which is meant to be the limit that a worker can work on a job before it is forcibly stopped by the server. This is a limit the employer sets when making jobs. The employer uses more banked CPU time when the timeout is higher. This is so that if the employer is malicious that they cannot set a time limit so high that steals the assigned workers time. Also this is set as a protection for the employer. If the program has some bug, so that it

does not ever halt, the worker will eventually stop, and the employer must at most pay the timeout in CPU time.

Currently in the system the job continues to run even though it has surpassed the time limit. In the future we would like to implement a way to stop the job when the timeout is reached. This way the employer can survey whether the program is correctly written or if there is a problem with the individual worker, by seeing how many timed out. This feature also allows the system to investigate a worker in case they never finish a project inside the given time limit.

In addition to this feature we would like to include a way for the employer to ask for the result whenever they want it, even when not every assigned worker has returned their results. This means that the employer will get results that are not yet tested for correctness. This feature is meant to help the employer quickly determine if there is a problem with the written program.

5.3.2 Scheduling with timeout

In the section above we explained how we would like to implement the timeout feature, but for this to work, scheduling needs to be altered as well. A worker can decide in what time frame it can calculate a job. This means that in the current system a job that takes up to two hours can be assigned to a worker who can only work for 30 minutes. To avoid that, we would like to implement a feature to the schedulers. This feature should not allow for assigning jobs to workers where the job timeout is higher than the workers time frame, e.g. if a workers time frame is less than 30 minutes it cannot get assigned a job with a timeout time larger than 30 minutes. We propose implementing this feature by adding a time frame to the request, that the worker sends to the server when querying for a new assignment.

5.3.3 User authentication security

We would like to introduce proper user credential authentication when logging in, in the future. Currently we send user credentials with every request in the path in plain text. This makes it very easy to intercept the information and in case of interception, no decryption is needed to use the information. One way to make sure that data is secure is to encrypt it before sending, by using a one way hashing algorithm to hash all passwords. In addition to encryption we would like to add token authentication which will decrease the amount of times user credentials will be sent through the system. This option also allows us to better manage users, e.g. determining how long they can stay logged in and what level of authority they have.

5.3.4 Catching malicious users

We would like to introduce ways to determine whether a user is malicious since both employer and worker have the option to be malicious.

Malicious employer

One problem a malicious employer could create is if the employer sends a job into the system that randomly returns different results. E.g. if an employer asks for 10 answers and create a 70 percent chance of getting one answer and 30 percent of getting another there will most likely be a majority of one answer. In this case the system would punish the innocent workers that are not part of the majority of "correct" answers by lowering their trust value. This case can be solved by the solution proposed in the following section.

Malicious worker

A worker has the possibility of being malicious by intentionally sending bad results back instead of calculating the jobs correctly. Currently our system uses a trust aspect to determine if a worker has been sending bad results. A way to counter our trust heuristic could be sending a fake result for every nth job done finished.

Another problem could be that the system could accidentally mistake an error in the employers program as a malicious worker. E.g. if the employer has created a badly written program with race conditions then the worker will be punished since each worker most likely will return different results. One way to make sure that the program is not at fault is to send jobs with known solutions to workers the server is suspicious about. This way the server can compare the results it receives from the workers with the results it already knows is the correct one. Unfortunately this can also be countered if the malicious worker knows when these jobs are being sent. For example, if every time a worker sends a bad result it then receives a test job, then the worker can counter this by calculating the test job correctly, making it seem like the employer is at fault. To solve this problem a test job could be sent in a random fashion, e.g. sending a test job several test jobs in a row, or sending one at a random time in the future, after the worker is deemed suspicious. This idea would further increase validity of our trust system, but at the cost of performance. Making workers calculate dummy programs is wasted potential.

5.3.5 Calculating non-deterministic jobs

So far in the system we have assumed each program written by the employer is deterministic. We would therefore like to include a solution that does not punish workers for non-deterministic programs. An example of a non-deterministic program could be one that solves Sudokus, because a Sudoku can have multiple answers that are all correct. In this case each worker might return a different set of answers, with them all being correct.

For this case we an idea could be to create a sandbox option where employers can use non-deterministic programs. This could e.g. be done by only using trusted devices to calculate the jobs. This way only devices that have given mostly correct results can be used, and it is more likely to be true answers even if it is the only one with that answer. The problem with this method is however that an employer could wait a long time to get trusted workers. Another solution could be to outright disable result checking for a particular job and simply let the employer deal with the correctness of all results from all workers. This is the cheapest and simplest solutions. It does, however, also lack any form of guarantees or heuristics.

Another proposal could be to make the employers make small result checking programs, that the server can use to verify results. This is particularly interesting since many problems are difficult to calculate, but easy to verify, which allows the server to do it without too much overhead. An example of this includes all decision problems with non-deterministic time complexity, where all solutions can be checked in deterministic polynomial time[21].

5.3.6 Gamification

In order to further incentivize the users to contribute with worker time, we would also like to introduce gamification aspects into the system. This could for example include visible statistics regarding job completion in the employer client. Another idea could be to introduce ranking systems, where the users could see their current score, possibly measured in terms of banked CPU time, compared to other users. The rankings could also be regional, for example the ranking between Danish users. Implementing gamification could be done by expanding the statistics object in the HTTP protocol to allow the workers to send more information regarding the computation of a specific job. Additionally, the database would have to be expanded with more statistics in the user table. The employer client and/or worker client should also receive some updates to visualize the user statistics in order to show the user the current rankings.

5.4 Conclusion

In this report we presented a problem description in section 1.1.2 for a system that could utilize the computational power of phones in time frames that they are not being used, e.g. when the phones are being charged while the owner is sleeping. We will now describe how we solved these problems.

We presented a framework that distributes computation of Python programs, by utilizing the processing power of mobile phones. The system allows uploading programs from an employer client to the server. The server then automatically distributes the jobs to the worker clients that are connected to the system. To determine a strategy for distributing jobs to the workers in the system, we introduced three schedulers. The first scheduler was a simple FIFO scheduler, which primarily served as a baseline for comparison and evaluation of the other schedulers. The Economic scheduler incentivizes users to contribute processing power to the network, by prioritizing jobs based on how much processing time a user has contributed. To counteract possibly malicious users we introduced a scheduler policy that assigns a trust measure to each worker in the system. Our experiments show that this scheduler was able to identify and exclude these malicious workers after they had processed a few jobs. These experiment were, however, conducted under the assumption that malicious workers behave in a predictable way where they always upload incorrect results.

To enable computation of programs on the phones, we incorporated the open source terminal emulator Termux into the worker client. The emulated terminal allowed us to interpret the Python scripts that the worker downloads from the server.

Evaluating the solution based on the requirements specification shown in section 2.2, we see that requirement 7. calling for security using authentication tokens, was not satisfied. Requirement number 12. calling for statistics to be shown on worker clients was also not fulfilled. The solution does, however, satisfy all of the remaining eleven requirements.

We tested the performance of the system, by uploading a program to the server, and comparing the processing time to how long it would take to execute the program locally. This test showed that when the user wishes to run multiple programs at the same time, or is executing the program on a slow machine, there can be a performance gain by uploading the job to the system. Ultimately, however, the processing time for a job in the system is bounded by the slowest worker that the job is assigned to, and so the performance of the system is dependent on the quality of the devices that are connected as workers.

Chapter 6

Work Process

In this chapter we describe our work process through the project. This project is a semester project, which means that it has been made over the course of 4 months side by side with semester courses. We decided to use scrum since it is our preferred method for project management. We tried to create a traditional scrum team with a product owner, scrum master and developer team. We had small deviations from traditional scrum like no daily scrum. This is because we had a limited time to work on the project in the start of the semester due to the courses. Even though we officially had a product owner, it was sometimes more of a shared role between all members. Officially, Hannah Lockey was the scrum master and Philip Holler was the product owner.

6.1 Scrum board

Miro is an online visual collaboration platform for teamwork. They have several templates for brainstorming, workflow, and more.[22]

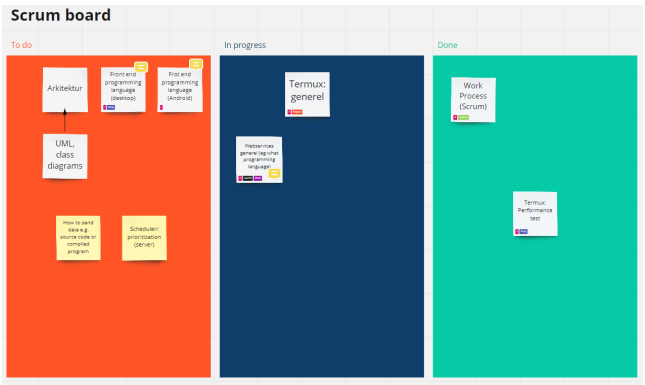


Figure 6.1: Screenshot of the Scrum board

We use Miro as a Scrum board, and with this the Scrum master can make an overview, and administer the project tasks to the group members. An example of a Scrum board can be seen in figure 6.1

We also used Miro as a brainstorming board for the MoSCoW analysis, which our requirement specification is based on. They already had a template for this, so we did not have to make our own. A screenshot of this can be seen in figure 6.2. As seen in the figure the notes are divided into three colors. We did this to quickly get an overview over which part of the system the requirement is linked to.

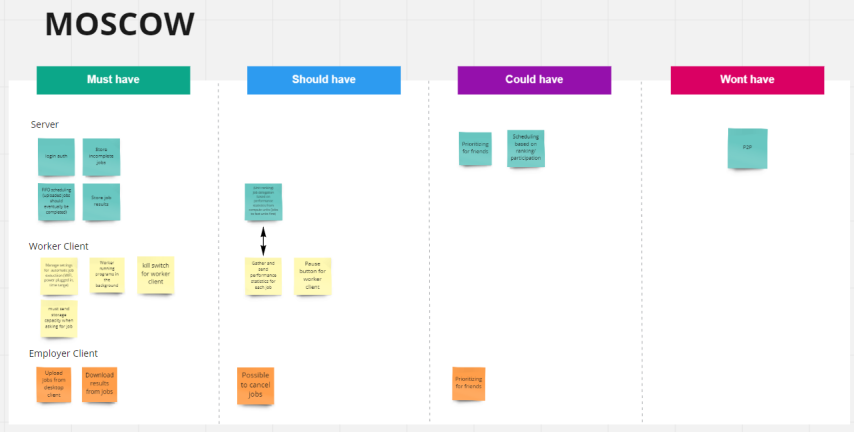


Figure 6.2: Screenshot of the MoSCoW brainstorm

6.2 Sprints

Our sprints always start off with a sprint review of the last sprint. Our sprint reviews consists off five items:

- Next meeting date
- Review of last sprint
- New sprint goal, and how to reach it
- Code review
- Product requirements review from Product Owner

The Scrum Master then makes a new Scrum Board in Miro based on the sprint review, and group members are assigned tasks. Examples of Scrum boards to the sprints can be seen in figure 6.3

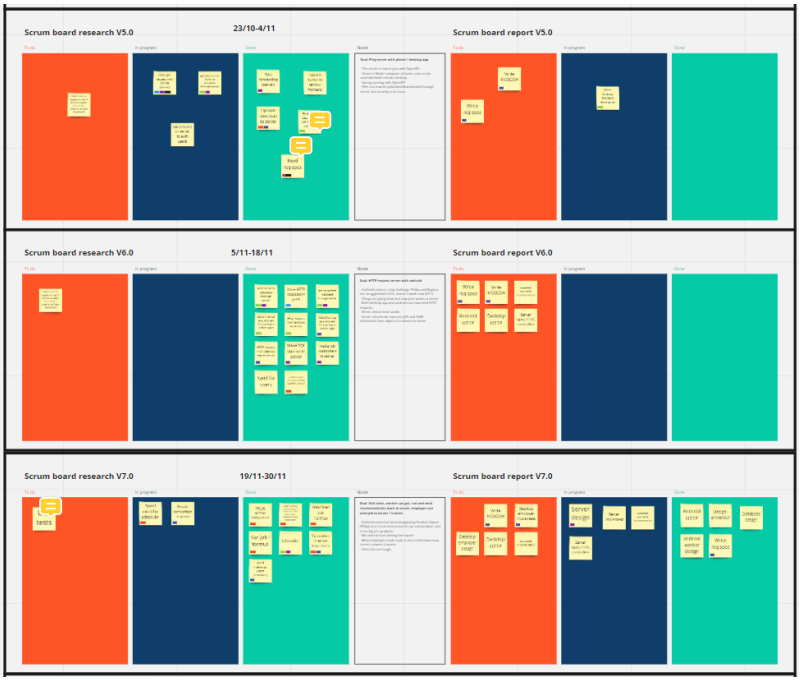


Figure 6.3: Screenshot of the Scrum sprints

We meet up as much as we can in between the sprints in our group room. This makes it possible for us to have fast meetings if anyone has questions. We use Discord to meet-up if physical meetings are not possible. [23]

We started with one week sprints, but they were too short as we have many lectures in the first half of the semester. The sprints were then remade to be two weeks long. After most of the courses stopped in late November, the sprints were converted to one week again.

Bibliography

[1] pewresearch.org. *Smartphone Ownership Is Growing Rapidly Around the World, but Not Always Equally*. https://www.pewresearch.org/global/2019/02/05/smartphone-ownership-is-growing-rapidly-around-the-world-but-not-always-equally/. 2019.

[2] Cisco. *Global Cloud Index Projects Cloud Traffic to Represent 95 Percent of Total Data Center Traffic by 2021*. https://newsroom.cisco.com/press-release-content?articleId=1908858. 2018.

[3] Primate Labs. *Cross Platform Benchmark*. https://www.geekbench.com. 2020.

[4] Qualcomm. *Snapdragon 865+ 5G Mobile Platform*. https://www.qualcomm.com/products/snapdragon-865-plus-5g-mobile-platform. 2020.

[5] Intel. *Intel Xeon W-2155 Processor*. https://ark.intel.com/content/www/us/en/ark/products/125042/intel-xeon-w-2155-processor-13-75m-cache-3-30-ghz.html. 2020.

[6] Ian Cutress and Joe Shields. *The Intel Xeon W Review: W-2195, W-2155, W-2123, W-2104 and W-2102 Tested*. https://www.anandtech.com/show/13116/the-intel-xeon-w-review-w-2195-w-2155-w-2123-w-2104-and-w-2102-tested/2. 2018.

[7] PrimateLabs. *Cross Platform Benchmark*. https://browser.geekbench.com/android_devices/oneplus-8. 2020.

[8] PrimateLabs. *Cross Platform Benchmark*. https://browser.geekbench.com/v5/cpu/search?utf8=&q=W-2155. 2020.

[9] Qualcomm. *Snapdragon 865 5G Mobile Platform*. https://www.qualcomm.com/products/snapdragon-865-5g-mobile-platform. 2020.

[10] Qualcomm. *Qualcomm Neural Processing SDK for AI*. https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk. 2020.

[11] Apple. *Integrate machine learning models into your app*. https://developer.apple.com/documentation/coreml. 2020.

[12] Huawei. *The most powerful and intelligent, ever*. https://consumer.huawei.com/en/campaign/kirin980/. 2020.

[13] Samsung. *Smarter future powered by AI*. https://www.samsung.com/semiconductor/minisite/exynos/technology/ai/. 2020.

[14] Statista. *Mobile OS market share 2019*. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/. 2019.

[15] termux. *Termux is an Android terminal emulator and Linux environment app*. https://termux.com. 2020.

[16] Bram Cohen. *Incentives Build Robustness in BitTorrent*. http://www.bittorrent.org/bittorrentecon.pdf. 2003.

[17] Open API Github Contributors. *OpenAPI Generator allows generation of API client libraries*. https://github.com/OpenAPITools/openapi-generator. 2020.

[18] Smartbear Software. *API development for everyone*. https://swagger.io. 2020.

[19] Tornadofx. *JavaFX Framework for Kotlin*. https://tornadofx.io. 2020.

[20] Termux. *Termux*. https://github.com/termux/termux-app. 2020.

[21] Michael Sipser. *Introduction to the Theory of Computation*. Vol. 3rd edition. Cengage Learning, 2013, 294, definition 7.19. ISBN: 9781133187790.

[22] miro.com.

[23] discord.com.

Appendix A

C program for initial Termux test

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

// Iterative function to find factorial of a number using for loop
unsigned long factorial(int n){
    unsigned long long int fact = 1;

    for (int i = 1; i <= n; i++) {
        fact = fact * i;
    }

    return fact;
}

long test_factorial_performance(){
    struct timespec spec;
    long before, after;

    printf("Started factorial performance test...\n");

    // Get current time
    clock_gettime(CLOCK_REALTIME, &spec);
    before = round(spec.tv_nsec / 1.0e6);

    // Run calculations
    unsigned long calcResult;
    for(int i = 0; i < 200000; i++){
        calcResult = factorial(65);
    }

    // Get current time
    clock_gettime(CLOCK_REALTIME, &spec);
    after = round(spec.tv_nsec / 1.0e6);
    printf("Finished factorial performance test... Calc result: %lu\n",
    calcResult);
    printf("Milliseconds elapsed: %lu\n", after - before);
```

```
38
39      return after - before;
40  }
41
42  // A utility function to swap to integers
43  void swap (int *a, int *b)
44  {
45      int temp = *a;
46      *a = *b;
47      *b = temp;
48  }
49
50  // A function to generate a random permutation of arr[]
51  void randomize ( int arr[], int n )
52  {
53      // Use a different seed value so that we don't get same
54      // result each time we run this program
55      srand ( time(NULL) );
56
57      // Start from the last element and swap one by one. We don't
58      // need to run for the first element that's why i > 0
59      for (int i = n-1; i > 0; i--)
60      {
61          // Pick a random index from 0 to i
62          int j = rand() % (i+1);
63
64          // Swap arr[i] with the element at random index
65          swap(&arr[i], &arr[j]);
66      }
67  }
68
69  long test_array_shuffle(){
70      struct timespec spec;
71      long before, after;
72
73      printf("Starting array shuffle test...\n");
74
75      int arr[10000];
76      for(int i = 0; i < 10000; i++){
77          arr[i] = i;
78      }
79
80      // Get current time before
81      clock_gettime(CLOCK_REALTIME, &spec);
82      before = round(spec.tv_nsec / 1.0e6);
83
84      // Computations
85      for(int i = 0; i < 1000; i++){
86          int n = sizeof(arr)/ sizeof(arr[0]);
87          randomize (arr, n);
88      }
89
90      // Get current time
91      clock_gettime(CLOCK_REALTIME, &spec);
92      after = round(spec.tv_nsec / 1.0e6);
93
```

```
94      printf("Finished array shuffle performance test\n");
95
96      printf("Milliseconds elapsed: %lu\n", after - before);
97
98      return after - before;
99  }
100
101
102
103 int main(int argc, const char* argv[]){
104     long result_fac = test_factorial_performance();
105
106     long result_arr = test_array_shuffle();
107
108     printf("Total time elapsed: %ld", result_fac + result_arr);
109
110     return 0;
111 }
```

Code snippet A.1: C code used for initial performance test on android

Appendix B

OpenAPI Specification

```yaml
openapi: "3.0.3"
info:
  description: "Semester project for 7th semester at Aalborg University"
  version: "1.0.0"
  title: "Offloading"
servers:
  - url: http://localhost:8080

tags:
  - name : user
  - name : job
  - name : assignment

components:
  schemas:
    Job:
      type: object
      properties:
        id:
          type: integer
          format: int64
          default: 0
        name:
          type: string
        jobpath:
          type: string
        timestamp:
          type: integer
          format: int64
          default: 0
        status:
          type: string
        employer:
          type: string
        answersNeeded:
          type: integer
        workersAssigned:
          type: integer
```

```
      timeoutInMinutes:
        type: integer
      confidenceLevel:
        type: number
        format: double

  Statistics:
    type: object
    properties:
      didFinish:
        type: boolean
      cpuTime:
        type: integer
        format: int64

  Result:
    type: object
    properties:
      jobid:
        type: integer
        format: int64
      resultfile:
        type: string
        format: byte
      statistics:
        $ref: '#/components/schemas/Statistics'


  JobFiles:
    type: object
    properties:
      jobid:
        type: integer
        format: int64
      data:
        type: string
        format: byte

  UserCredentials:
    type: object
    properties:
      username:
        type: string
      password:
        type: string

  DeviceId:
    type: object
    properties:
      uuid:
        type: string

  JobId:
    type: object
    properties:
      jobID:
```

```
          type: integer
          format: int64

paths:
  # Users
  /users/{userCredentials}:
    post:
      tags:
        - user
      description: Creates a user
      operationId: createUser
      parameters:
        - name: userCredentials
          in: path
          description: Credentials used to create new user
          required: true
          schema:
            $ref: '#/components/schemas/UserCredentials'
      responses:
        '200':
          description: User Created
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/UserCredentials'

    get:
      tags:
        - user
      description: Fakes a login
      operationId: login
      parameters:
        - name: userCredentials
          in: path
          description: Credentials used to login user
          required: true
          schema:
            $ref: '#/components/schemas/UserCredentials'
        - name: deviceId
          in: query
          description: If logged in from a worker
          required: false
          schema:
            $ref: '#/components/schemas/DeviceId'
      responses:
        '200':
          description: User logged in
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/UserCredentials'
        '404':
          description: User credentials invalid

    delete:
      tags:
```

```
151         - user
152       description: Deletes a user
153       operationId: deleteUser
154       parameters:
155         - name: userCredentials
156           in: path
157           description: Credentials for user that should be deleted
158           required: true
159           schema:
160             $ref: '#/components/schemas/UserCredentials'
161       responses:
162         '200':
163           description: User Deleted
164           content:
165             application/json:
166               schema:
167                 $ref: '#/components/schemas/UserCredentials'
168
169
170
171   /jobs/{userCredentials}:
172     post:
173       tags:
174         - job
175       description: Uploads a job to be computed
176       operationId: postJob
177
178       parameters:
179         - name: userCredentials
180           required: true
181           in: path
182           schema:
183             $ref: '#/components/schemas/UserCredentials'
184         - name: answersNeeded
185           required: true
186           in: query
187           schema:
188             type: integer
189             format: int32
190         - name: jobname
191           required: true
192           in: query
193           schema:
194             type: string
195         - name: timeout
196           required: true
197           in: query
198           schema:
199             type: integer
200
201       requestBody:
202         required: true
203         content:
204           application/json:
205             schema:
206               type: string
```

```
              format: byte
      responses:
        '200':
          description: Computed job
          content:
            application/json:
              schema:
                type: integer
                format: int64


    get:
      tags:
        - job
      description: Returns all jobs uploaded by given user
      operationId: getJobsForUser
      parameters:
        - name: userCredentials
          in: path
          description: Username of user to filter jobs by
          required: true
          schema:
            $ref: '#/components/schemas/UserCredentials'
      responses:
        '200':
          description: List of jobs for selected user
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Job'

  /jobs/{userCredentials}/{jobId}:
    delete:
      tags:
        - job
      description: Deletes a job
      operationId: deleteJob
      responses:
        '200':
          description: Job deleted
      parameters:
        - name: jobId
          in: path
          description: Job to be deleted
          required: true
          schema:
            type: integer
            format: int64
        - name: userCredentials
          required: true
          in: path
          schema:
            $ref: '#/components/schemas/UserCredentials'

```

```
263   /jobs/{userCredentials}/{jobId}/result:
264     get:
265       tags:
266         - job
267       description: Gets the job result
268       operationId: getJobResult
269
270       parameters:
271         - name: jobId
272           in: path
273           description: Job to query result for
274           required: true
275           schema:
276             type: integer
277             format: int64
278         - name: userCredentials
279           required: true
280           in: path
281           schema:
282             $ref: '#/components/schemas/UserCredentials'
283
284       responses:
285         '200':
286           description: Found the job result
287           content:
288             application/json:
289               schema:
290                 $ref: '#/components/schemas/JobFiles'
291
292         '202':
293           description: Result file not ready yet
294
295
296   /jobs/{userCredentials}/{jobId}/files:
297     get:
298       tags:
299         - job
300       description: Gets the job files
301       operationId: getJobFiles
302
303       parameters:
304         - name: jobId
305           in: path
306           description: Job to get files from
307           required: true
308           schema:
309             type: integer
310             format: int64
311         - name: userCredentials
312           required: true
313           in: path
314           schema:
315             $ref: '#/components/schemas/UserCredentials'
316
317       responses:
318         '200':
```

```
          description: Found the job files
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/JobFiles'

        '202':
          description: Could not find job


  # Jobs assigned to workers
  /assignments/{userCredentials}/{deviceId}:
    get:
      tags:
        - assignment
      description: Returns a job for the device to process
      operationId: getJobForDevice

      parameters:
        - name: userCredentials
          in: path
          description: User authentication
          required: true
          schema:
            $ref: '#/components/schemas/UserCredentials'
        - name: deviceId
          in: path
          description: Identification for device
          required: true
          schema:
            $ref: '#/components/schemas/DeviceId'

      responses:
        '200':
          description: Job id and files for job assigned to device
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/JobFiles'
        '202':
          description: No jobs available for device

  /assignments/{userCredentials}/{deviceId}/{jobId}:
    post:
      tags:
        - assignment
      description: Uploads the result
      operationId: uploadJobResult

      requestBody:
        content:
          application/json:
            schema:
              title: jobresult
              type: object
              properties:
```

```
                result:
                  $ref: '#/components/schemas/JobFiles'

    parameters:
      - name: userCredentials
        in: path
        description: User authentication from worker
        required: true
        schema:
          $ref: '#/components/schemas/UserCredentials'
      - name: deviceId
        in: path
        description: Identification for device
        required: true
        schema:
          $ref: '#/components/schemas/DeviceId'
      - name: jobId
        in: path
        description: Job upload result for
        required: true
        schema:
          type: integer
          format: int64

    responses:
      '200':
        description: Upload successful

  patch:
    tags:
      - assignment
    description: Notify the server that the worker is still working on the job
    operationId: pingAssignment
    parameters:
      - name: userCredentials
        in: path
        description: User authentication from worker
        required: true
        schema:
          $ref: '#/components/schemas/UserCredentials'
      - name: deviceId
        in: path
        description: Identification for device
        required: true
        schema:
          $ref: '#/components/schemas/DeviceId'
      - name: jobId
        in: path
        description: Job that the device is currently processing
        required: true
        schema:
          type: integer
          format: int64
    responses:
      '200':
```

```
          description: Continue
        '410':
          description: Assignment aborted

    delete:
      tags:
        - assignment
      description: Quits the current assignment
      operationId: quitAssignment
      responses:
        '200':
          description: Assignment quit
      parameters:
        - name: userCredentials
          in: path
          description: User authentication from worker
          required: true
          schema:
            $ref: '#/components/schemas/UserCredentials'
        - name: deviceId
          in: path
          description: Identification for device
          required: true
          schema:
            $ref: '#/components/schemas/DeviceId'
        - name: jobId
          in: path
          description: Job to quit
          required: true
          schema:
            type: integer
            format: int64
```

Code snippet B.1: API specification for OpenAPI in .yaml format

Appendix C

Python Test Program for System Test

```
import random
import time
import math

start_time = time.time()
l = list(range(3700))

for i in l:
    random.shuffle(l)
    math.factorial(i)

print("--- %s seconds ---" % (time.time() - start_time))
```

Code snippet C.1: 30 second test program