FACULTY
OF ELECTRICAL
ENGINEERING
CTU IN PRAGUE

Bachelor's thesis

Design and implementation of a scalable server for parallelization of optimization algorithms execution

Lukáš Forst

Department of Computer Science
Supervisor: Ing. Ondřej Vaněk, Ph.D.

May 22, 2019

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Forst** Jméno: **Lukáš** Osobní číslo: **465806**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Návrh a vývoj škálovatelného serveru pro paralelizaci běhů optimalizačních algoritmů

Název bakalářské práce anglicky:

Design and implementation of a scalable server for parallelization of optimization algorithms execution

Pokyny pro vypracování:

Large-scale optimization problems are non-trivial to solve and require a significant amount of computational resources as well as computational time to find a solution. The challenge is to solve not only a single task but a multitude of them in a parallel manner. Additionally, the tasks are non-homogeneous, often describing a different problem. This problem can be solved by designing and intelligent scheduler able to schedule such tasks on a distributed computational platform.
The goal of the thesis is:
Study the state-of-the-art approach to computational tasks scheduling. Study various types of optimization problems and approaches and understand their computational needs. Study the distributed scalable architecture and approaches to schedule tasks on such architecture. Design a scheduling and load-balancing module able to ingest various optimization tasks and schedule them with respect to several criteria. Implement the scheduler. Evaluate the scheduler on a number of scenarios.

Seznam doporučené literatury:

[1] Boyd, S., & Vandenberghe, L. (2004). Convex optimization. Cambridge university press.
[2] AWS. Load Balancing, User Guide. Amazon. Online.
https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/elb-ug.pdf.
[3] Iqbal, M. A., Saltz, J. H., & Bokhart, S. H. (1986). Performance tradeoffs in static and dynamic load balancing strategies.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Ondřej Vaněk, Ph.D., centrum umělé inteligence FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **05.02.2019** Termín odevzdání bakalářské práce: **24.05.2019**

Platnost zadání bakalářské práce: **20.09.2020**

Ing. Ondřej Vaněk, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. | Datum převzetí zadání

Podpis studenta

Acknowledgements

I would first like to thank my thesis advisor, Ondřej Vaněk, whose office's door was always open whenever I ran into a trouble spot or had a question about my research or writing.
I would like to also thank my colleagues in Blindspot Solutions and especially to Petr Eichler, whose valuable comment suggestions on my paper inspired me to improve the quality of the assignment and helped me to overcome the various challenges I faced during the development.
Finally, I must express my very profound gratitude to my parents and to my better half for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis.
This accomplishment would not have been possible without them.

Thank you.

Lukáš Forst

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 22, 2019 .....

Czech Technical University in Prague
Faculty of Electrical Engineering

Abstrakt

Velké optimalizační úlohy nejsou jednoduché a pro nalezení řešení vyžadují velké množství výpočetního výkonu, stejně jako výpočetního času. Proto je výzvou je vyřešit paralelně a v co nejkratším možném čase. Optimalizační problémy nejsou homogenní skupina, ale popisují často kompletně odlišnou doménu a tedy mohou být velice různorodé. Problém exekuce optimalizačních programů a jejich výkonové náročnosti, lze řešit za použití inteligentního plánovače, který umožňuje naplánovat samotné spuštění úlohy v distribuovaném výpočetním prostředí.
Tato práce se zabývá návrhem vyvažovače zátěže vyvinutého přímo pro optimalizační algoritmy, který minimalizuje plýtvání zdroji a zvyšuje výkon pomocí správného rozdělení využití mezi více instancí těchto algoritmů. Práce analyzuje nejmodernější řešení a technologie, které se používají k řešení takových problémů ve velkých infrastrukturách. Následně je navrženo nové řešení pro vyrovnávání zátěže specifické pro danou doménu. Práce obsahuje matematickou formalizaci problému optimalizace vyvažování zátěže a zároveň popisuje a řeší další nalezené problémy, jako je například predikce hyperbolických časových řad. Následně je v práci navržena architektura na bázi mikroslužeb pro systém vyvažování zátěže a také je daný systém kompletně realizován. Zároveň jsme navrhli a implementovali simulace a experimenty, které testují implementovaný vyvažovač zátěže.
Závěrečná kapitole práce se zabývá vymezením nezbytných kroků pro úplnou produkcionalizaci navrženého systému vyvažování zátěže a také nastiňuje budoucí vývoj nástrojů a knihoven, které byly vyvinuty vedle primárního systému.

Klíčová slova vyvažování a distribuce zátěže, optimalizační algoritmy, kombinatorická optimalizace, TASP, OptaPlanner, Kotlin, Ktor, Docker

Abstract

Large-scale optimization problems are non-trivial to solve and require a significant amount of computational resources as well as computational time to find a solution. The challenge is to solve not only a single task but a multitude of them in a parallel manner. Additionally, the tasks are non-homogeneous, often describing a different problem. This problem can be solved by designing and intelligent scheduler able to schedule such tasks on a distributed computational platform.
This work introduced the load balancer developed explicitly for the optimization algorithms, which should minimize resources wasting and increase the performance using correct utilization distribution across the multiple instances of such algorithms. The thesis analyzes the state-of-the-art solutions and technologies, that are being used to solve load balancing problems in the large infrastructures. Subsequently, the new domain-specific load balancing solution is proposed. The thesis proposes the mathematical formalization of the load balancing optimization problem and the related challenge, such as hyperbola time series prediction. Subsequently, the thesis designs the microservices architecture for the load balancer and also delivers the complete implementation of the proposed system. This thesis also proposed and implemented the simulations and experiments, which evaluates the implemented load balancer. The final chapter of the thesis addresses out of scope steps for complete productionalization of the proposed load balancing system and also outlines the future development of the tools and libraries, that was developed alongside the primary system.

Keywords load balancing, optimization algorithms, combinatorial optimization, TASP, OptaPlanner Kotlin, Ktor, Docker

Contents

Chapter 1

Introduction

The globalization of the world's economies is a major challenge to local industry and it is pushing the manufacturing sector to the transformation called *Industry 4.0* [1]. In order to become more competitive, manufacturers need to embrace emerging technologies, such as advanced analytics, artificial intelligence and mathematical optimization to improve their efficiency and productivity.

Specifically the manufacturing industry sector, which have high production costs, faces multiple problems, where employing mathematical optimization can reduce cost or improve the efficiency of the process. Taking as an example the car manufacturers, there are many processes, that can be optimized to reduce their cost or to use needed resources more efficiently, such as internal logistics, car parts transportation, parts stocking, cars manufacturing and the allocation of the various types of resources. These optimization challenges are often solved by the proprietary software systems with included optimization engine, where the one problem domain is usually handled by the single program operating specifically with such domain.

These applications typically have a user interface for the data visualization and an engine running an optimization algorithm. Although the data visualization part of the application does not require powerful hardware, the immense complexity of the mathematical optimization problems and thus the performance requirements for such optimization engine solving them, are not always satisfiable. Moreover, the computer performance is a finite resource and it costs money paid for the computer components or for the electricity used. For those reasons, the optimization software systems have often limited access to computer resources. In addition, this computer performance is not used all the time, since the typical usage of such application lays mainly in data visualization, hence the performance, required only when the optimization engine is running, is unused.

Using such software architecture seems to be highly inefficient, since the instances, that are in time t running the optimization algorithm, are over-

whelmed and at the same time t, the applications, that are not running the optimization tasks, do not use their powerful computer resources at all.

The potential solution for these problems lays in microservices architecture, where the parts of the applications are independent and able to run separately. Using this approach enables distributed computing and therefore outsourcing the demanding optimization engine to more powerful servers. This approach solves the lack of resources for the optimization engine on the less powerful servers, but it also introduces a new challenge in the load balancing of the servers, where the optimization engine is being executed.

In this thesis, we would like to present the load balancer specifically developed for the optimization algorithms, which should minimize resources wasting and increase the performance using correct utilization distribution across the multiple instances of such algorithms. In the first chapters, the thesis analyzes the state-of-the-art solutions and technologies, that are being used to solve load balancing problems in the large infrastructures. These technologies are evaluated and the new domain-specific load balancing solution is proposed. The thesis then transforms and formalizes the load balancing optimization problem into an integer linear programming problem. The related challenges arising from the problem formalization, such as hyperbola time series prediction, are defined and the solutions for such problems are proposed. Subsequently, the thesis designs and evaluates the microservices architecture for the load balancer and also delivers the complete implementation of the load balancer. This thesis also proposed and implemented the simulations and experiments, which evaluates the implemented load balancer.

In the last chapter, the future work is addressed and the steps, for moving the load balancer into the production environment, are proposed. Apart from the planned steps for the load balancer system, the last chapter outlines the future of the developed tools and libraries.

1.1 Thesis goals

To summarize the previous introduction, the main goals of the thesis were set as follows.

Study the state-of-the-art approach to computational tasks scheduling

This thesis extensively studies and describes the state of the art algorithms used for computational task scheduling and load balancing in section 2.1. It also presents technologies used to solve such problems. The difference between the state-of-the-art load balancing strategies and why the thesis proposes and implements the new technology instead of using the existing one is presented in section 2.1.3.

Study various types of optimization problems and approaches and understand their computational needs

The thesis brings an overview of and describes various optimization problems, techniques, and algorithms that are being used in real-life situations to solve diverse industries problems in section 2.2.

Study the distributed scalable architecture and approaches to schedule tasks on such architecture

The distributed architecture study is presented in section **??**. This section also introduces the overall distributed architecture of the load balancer including the simulation module.

Design a scheduling and load-balancing module able to ingest various optimization tasks and schedule them with respect to several criteria

The complex problem of the load balancing of the optimization tasks is formalized in chapter 3. Based on the problem formalization, solution design, including the final load balancing algorithm, is introduced in chapter 4.

Implement the scheduler

The load balancing module design from chapter 4 is used for load balancing module implementation in chapter 5. The very same chapter contains an overview of the module architecture in section **??**. The architecture was designed to keep future infrastructure development in mind and thus, few out of scope future steps were outlined in section 7.1.

Evaluate the scheduler on a number of scenarios

The chapter 6 describes the way, how the scheduler and the load balancing system was tested. It also describes in section 6.2 how the runtime data of optimization algorithms were collected. The next section 6.3 presents the scheduler evaluation.

Apart from the original goals of the thesis, which were set in the assignment, many out of scope and future goals arose during the formalization, research, and implementation. These goals and overviews how to achieve them are outlined in section 7.1.

Chapter 2

State of the art

2.1 Load Balancing

Load balancing is a technique for a division of processing work in the distributed environment of execution units [1] to deliver faster service with higher efficiency. It improves the distribution of workloads across the whole environment and thus balances resources usage while maximizing throughput and minimizing response time. Load balancer is typically either dedicated *hardware device* or *software program*.

A **hardware** load balancer is a dedicated hardware device which distributes network traffic across a cluster of servers [2]. These devices are used mainly in the data centers to ensure equal distribution of traffic between the application servers. The main benefit of using hardware load balancer is zero balancing overhead on the host machines, because all decisions are made on dedicated hardware specially developed for such tasks.

A **software** load balancer is a program operating on the application server with the same aim as a hardware load balancer. The main advantage of the software load balancing is that it can be heavily customized and deployed to its server. This paper will discuss only the software load balancing approach.

In general, software load balancing algorithms can be classified as either *static* or *dynamic*.

2.1.1 Static Load Balancing

Static load balancing is an approach where system information is provided a priori and load balancer does not use performance information about exe-

[1] In general, the execution unit can be CPU, network links, storage devices or other devices, in this paper *execution unit* or also referred as *execution node* or as *host* is a computer executing assigned job

cution node [2], to make distribution decisions. The performance possibilities and the load of the execution point (or node) are not taken into account when decision - where to execute the current task - is being made, because load-balancing decisions are made at compile time. When a decision is made, no other interaction with executing node, regarding the current task, is being made. In other words, once the load is allocated to the execution node, it cannot be transferred to another node. The static load balancing method is to reduce the overall execution time of a concurrent program while minimizing the communication delays [3]. The main advantage of static load balancing methods is mainly the fact that there is minimal communication delay between system nodes and therefore execution overhead is minimized to almost zero. For that reason is static load balancing mainly used in the fields, where server response is crucial such as serving a web page.Also, the implementation of some static load balancing algorithm is straightforward, since the used methods are elementary.

The main disadvantage of static load balancing is that it does not take into account the current state of the system when making a decision. This could potentially lead to performance issues in the whole system because some nodes can be overloaded, although others are not working at all.

Another drawback of this approach is that hardware resources are allocated only once in the execution time. Since optimization jobs are very heterogeneous, they sometimes have different power requirements during the execution. For example *TASP*[3] uses only one thread when creating a feasible plan in the first algorithm iteration - this task relays only on single core performance. However, when the first iteration is completed, all following can be done by multiple threads, therefore it could be useful to execute the first iteration on a machine with better single core performance and then transfer the algorithm into a device focused on multithreading execution. This is something that can not be done while using static load balancing.
Following static load balancing algorithms are commonly used.

First Alive

First alive or also called *Central Manager* algorithm uses the concept of a primary server and backup servers [4]. All tasks are scheduled to be executed on the primary server unless the central server is down. Then the load will be forwarded to the first backup server. This algorithm has almost zero levels of internal process communication, which leads to better performance when there are lots of smaller tasks.

[2]Execution node - Server executing task which is being scheduled by the load balancer. In our case, this task is solving an optimization problem by the solver.

[3]**Task and Asset Scheduling Platform** - proprietary optimization software developed by Blindspot Solutions, described in section 2.2.2

Round Robin

Round Robin algorithm which distributes workload evenly to all nodes. It is being done in round-robin order, where the load is distributed to each node in circular order without any priority. Round Robin is easy to implement and as well as *First alive* algorithm has almost none inner communication overhead. This algorithm performs best when tasks have equal, or at least similar, processing time.

Weighted Round Robin

Weighted round robin algorithm maintains a weighted list of servers and forwards new connections in proportion to the weight, or preference, of each server. This algorithm uses more computation times than the round robin algorithm. However, the additional computation results in distributing the traffic more efficiently to the server that is most capable of handling the request [4].

Threshold Algorithm

Threshold algorithm - execution nodes keep a private copy of the system's load when the load state of a node exceeds a load level limit, the node sends message to all remote nodes that it is overloaded. If the local state is not overloaded, then the load is allocated locally. Otherwise a remote node, that is not overloaded, is selected and if no such node exists it is also allocated locally. This algorithm has low interprocess communication and a large number of local process allocations. The later reduces the overhead of remote process allocation and the overhead of remote memory access, which leads to performance improvements [5].

Least Connections

Least connections algorithm maintains a record of active server connections and forward a new connection to the server with the least number of active connections [4]. This can be generally useful while having many concurrent requests, that can be dispatched quickly.

Randomized Algorithm

The randomized algorithm uses a random selection of the execution node without having any information about it.

2.1.2 Dynamic Load Balancing

Unlike static load balancing algorithms, dynamic algorithms use runtime state information to more informative decisions while distributing the jobs.

They monitor changes on the system workload and take it into account when the decision, where to execute a job, is being made. The process of monitoring the system is not stopped after the execution job started and if circumstances change, job execution can be transferred to another system node, which then proceeds with execution.

While many different load balancing algorithms have been proposed, there are four basic steps that nearly all algorithms have in common [6].

1. Monitoring workstation performance (load monitoring)

2. Exchanging this information between workstations (synchronization)

3. Calculating new distributions and making the work movement decision (rebalancing criteria)

4. Actual data movement (job migration)

Dynamic load balancing algorithms can be divided into two groups based on their control form, or in other words, where load balancing decisions are made [6].

- Centralized - a single node in the network is responsible for all load distribution

- Distributed - all nodes ale equal

While in the centralized scheme are decisions made in one master workstation, in a distributed scheme, the load balancing algorithm runs on all nodes and each node balances itself. Each of this approach has its ups and downs, the centralized scheme can be the potential performance bottleneck since it relies on one system node, on the other hand distributed scheme has communication overhead, because it requires broadcast communication between all algorithm instances.

The main advantage of dynamic load balancing is that it allows changing execution node in runtime. For that reason, it is possible to change hardware characteristics according to the job execution phase. For example, execute initial phase of optimization algorithm on the machine with powerful single core performance and then move the job to the computer with multiple, less powerful, cores to let it run in parallel. Also as a result of runtime scheduling, dynamic load balancing algorithms tend to provide significant improvements in performance over static algorithms. However, this comes at the additional cost of collecting and maintaining load information [6]. For that reason, dynamic load balancing suites better for long running tasks, which can be managed and distributed better than for fast queries.

Dynamic load balancing strategies

There are three major parameters which usually define the strategy a specific load balancing algorithm will employ. These three parameters answer three important questions [6]:

1. Who makes the load balancing decision?
2. What information is used to make the load balancing decision?
3. Where the load balancing decision is made?

Question number 1 is answered based on whether a **sender-initiated** or **receiver-initiated** policy is employed. In *sender-initiated* policies, congested nodes attempt to move work to lightly-loaded nodes. In *receiver-initiated* policies, lightly-loaded nodes look for heavily-loaded nodes from which work may be received [6].

The question 'What information is used to make the load balancing decision¿ is answered by following policies - **global** and **local**. When the algorithm uses *global* policy, the load balancer uses the performance profiles of all execution nodes connected to the network. When using *local* policy, only local [4] nodes are taken into account while creating a performance profile of the system.

The last parameter - 'where the load balancing decision is made' - is answered by used control form, as mentioned previously, dynamic load balancing algorithms are divided into two groups based on their control form - **centralized** and **distributed**.

I would like to present two general dynamic load balancing algorithms - *Central Queue Algorithm* and *Local Queue Algorithm.*

Central Queue Algorithm

Central queue algorithm is based on centralized receiver-initiated load balancing strategy. It uses a cyclic FIFO queue on the main host to store new activities[5] and unfulfilled requests. New activity request is inserted into the queue, and there it is stored until some execution node picks it up.

Whenever a request for an activity (which is sent by executing node in the case when its load has fallen below a specified threshold) is received by the queue manager[6], it removes the first activity from the queue and sends it to the requester. If the queue is empty, the request is buffered, until a new activity is available. If a new activity arrives at the queue manager while there

[4] Workstations are usually divided into groups, in this context *local* means in the same group of workstations

[5] Activities - jobs to be executed, in our case optimization job

[6] Queue manager - central server which manages queue

are unanswered requests in the queue, the first such request is removed from the queue, and the new activity is assigned to it.

When an execution node load falls under the threshold, the local load manager sends a request for a new activity to the central load manager (which manages the central system queue). The central load manager answers the request immediately if a ready activity is found in the queue, or queues the request until a new activity arrives [7].

Local Queue Algorithm

Local queue algorithms use distributed receiver-initiated strategy.

Its main feature is that it supports dynamic process migration.This algorithm in the first step uses the static allocation of all new processes - all processes are allocated to under loaded hosts. In the second step, the process migration is initiated by a host when its load falls under predefined threshold[7]. In such case, the execution node attempts to get several processes from remote hosts. It randomly sends requests with the number of local ready processes to remote load managers. When a load manager receives such a request, it compares the local number of ready processes with the received number. If the former is greater than the latter, then some of the running processes are transferred to the requester and an affirmative confirmation with the number of processes transferred is returned. [7]

Local queue algorithm is a distributed load balancing algorithm where each execution node requests a new activity when it is underloaded. The main advantage of using such an algorithm is the fact that there is no central point, where all requests are managed and distributed to another segment of the system. For that reason is this particular algorithm copes and performs well under an increased or expanding workload.

2.1.3 Load Balancing for Optimization Algorithms

In general, load balancing algorithms don't use information about what exactly is being executed on the execution nodes. This is because they are working mainly on the network layer and thus don't need that information. Also, they are mainly designed to be generic - to be used with any system and to be suitable for every environment. From the load balancer point of view, everything behind load balancing layer of the system is a black box.

Because there is no knowledge about the algorithms operating on the execution nodes, load balancing algorithm can not make fully informed decision about the job execution. However, this paper focus on the load balancing and execution scheduling of optimization algorithms, therefore, unlike generic load balancing solutions, proposed load balancer **have** the information about execution algorithms on the host machine and thus, the load balancing decision

[7] This threshold can be defined by the user, and it is an input for the algorithm

is more informed. More informed load balancing decisions could potentially lead to better performance and costs reduction as well as higher capacity of the whole system.

Since load balancer is aware of algorithms running on the hosts, it can take into account execution criteria which can be specified (such as execution time) or at least estimated (how much memory will be needed according to the domain size) in advance to make even more informed balancing decision when scheduling the job execution. This is also the main difference between the generally used and existing load balancing software and a solution proposed in this paper.

2.2 Optimization Algorithms

In this section, the thesis presents various optimization techniques and existing solvers implementations, that can be used to solve such optimization problems.

2.2.1 Linear Optimization

Linear optimization (or linear programming) is a method to achieve the best outcome in a mathematical modelwhose requirements are represented by linear relationships. The algorithms are widely utilized in company management, such as planning, production, transportation, technology, and other issues.

The main benefit of linear optimization is that it provides the best possible solution, because optimization algorithms are guaranteed to provide an optimal solution. Although almost everything can be represented as a linear problem, linear programming solvers could be unable to provide a solution since, in most cases, computation time grows exponentially. Even though there are solvers that can provide ϵ (partial) solution,this solution can be (and in most cases is) unusable because it is not optimal at all.

There are plenty of linear programming solvers available. We want to highlight the following two optimization kits.

GLPK

GNU - *GNU Linear Programming Kit* is a software package intended for solving large-scale linear programming (LP), mixed integer programming (MIP), and other related problems. It is a set of routines written in ANSI *C* and organized in the form of a callable library [8]. Although originally is GLPK written in *C* programming language, there is an independent project,

which provides Java-based interface for execution of GLPK via Java Native Interface. [8]

Google OR-Tools

Google OR-Tools - OR-Tools is an open source software suite for optimization, tuned for tackling the world's toughest problems in vehicle routing, flows, integer and linear programming, and constraint programming [9]. Tools contain *Glop*, which is Google's custom linear solver. One of the most significant advantages of Google OR-Tools is an API supporting multiple programming languages - *C++, Python, C#*, and *Java*.

2.2.2 Heuristic algorithms

Heuristics algorithms (or HA) are designed to solve optimization problems faster and more efficient fashion than Linear Optimization methods by using different kinds of heuristics and metaheuristics. In exchange for that, algorithms sacrifice optimality, accuracy, precision, and completeness. Thus the solution provided by HA is not guaranteed to be optimal. HA are often used to solve various types of NP-complete problems such as Vehicle Routing, Task Assignment, Job Scheduling, or Traveling Salesmen Problem. Heuristic algorithms are most often employed when approximate solutions are sufficient and exact solutions are necessarily computationally expensive [10].

The main advantage of heuristic algorithms is that they provide a quick feasible solution. Because the implementation of HA is easier than LP and they provide at least a feasible solution for optimization problems, they are solving; they are widely used in organizations that face such optimization problems. The main downside of HA is the fact that they can't guarantee that the found solution is the optimal one.

, We want to mention two implementations of heuristics algorithms - OptaPlanner and TASP.

OptaPlanner

OptaPlanner is an open source generic heuristics based constraint solver. It is designed to solve optimization problems such as Vehicle Routing, Agenda Scheduling, etc. While solving an optimization task, it combines and uses various optimization heuristics and metaheuristics such as Tabu Search or Simulated Annealing.

OptaPlanner is written in pure Java and runs on JVM. Therefore it can be used as a Java library.

[8] Java Native Interface - Interface provided by the Java platform to run and integrate non-Java language libraries

TASP

Task and Asset Scheduling Platform is a lightweight framework developed by Blindspot Solutions [11] designed to solve a large variety of optimization and scheduling problems from the area of logistics, workforce management, manufacturing, planning, and others. It contains a modular, efficient planning engine utilizing the latest optimization algorithms. TASP is delivered as a software library to be used through its API in applications which require powerful scheduling capabilities.

It is written in Kotlin which runs on JVM. Therefore it can be easily used as a library to any JVM based project.

Chapter 3

Problem formalization

The problem with the implementation of optimization algorithms in applications is that their performance requirements are quite high and are fully utilized only while working. The optimization algorithm is not running all the time and for that reason hardware resources are mainly unused. These unused resources could be potentially used by another instance of the algorithm or can be shut down completely to reduce hosting costs. Also adding more time to the job execution does not always bring a better solution but it certainly costs more.

The figure 3.1 displays standard case, when the application is deployed on the server. Because the application uses an optimization algorithm, the server must be powerful enough to deliver suitable results. This also means, that the server resources must be allocated to the application and to nothing else, which leads to resource wasting.

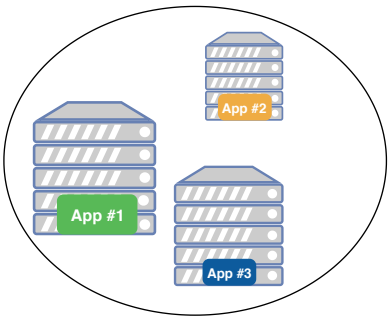

Figure 3.1: Problem visualization

The next figure 3.2 visualizes scenario when the load balancer is used. Instead of using powerful servers for each of the applications, it is enough to use a small server or even deploy multiple applications on the one machine. This can be done thanks to the outsourcing of the optimization jobs to the external servers with more computational power. The circles are optimization jobs

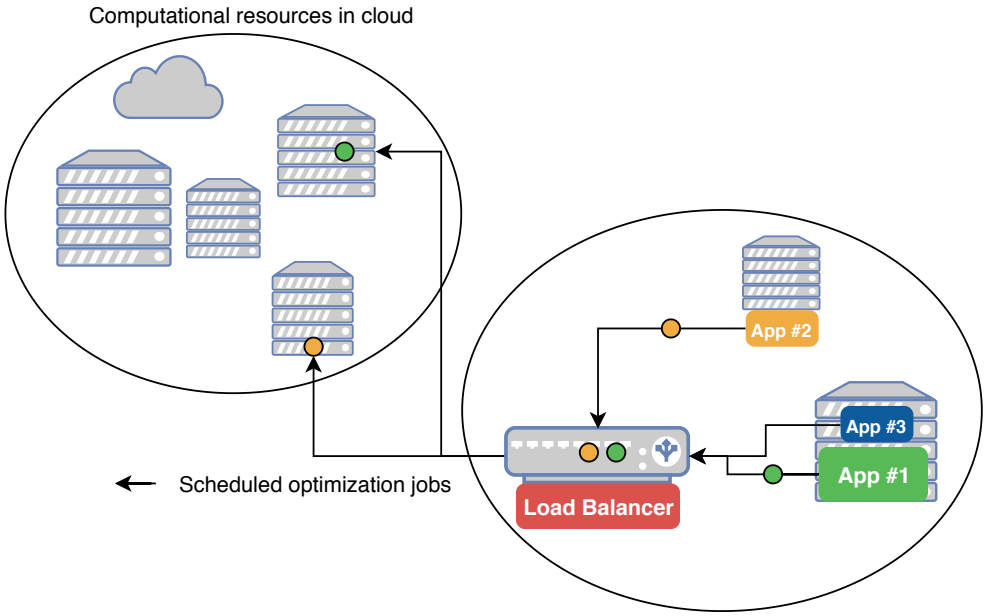

Figure 3.2: Problem visualization with the load balancer

triggered by the corresponding colored applications and are being executed on the different servers in the cloud.

Moreover, as can be seen in the figure 3.2, these servers do not need to be allocated all the time. Most resources providers, such as AWS [12] or Azure [13] allow to reallocate hardware on the fly. This leads to potentially significant cost savings and ultimately to the more effective infrastructure.

3.1 Formal definition

Following definition implies, that it can be represented as an Integer Linear Programming (ILP) problem. Although, it is not entirely formalized, it is only mechanical work to transcript all implications into equations. For that reason, and better readability, we left the implications.

3.1.1 Variables Definition

Following indexes, inputs and variables are used in the optimization criteria.

Indexes

- j - index used to identify something related to the execution job, in real world this is most likely job id, located in the right upper corner - x^j, set of all jobs in the system is represented by J
- r - index used to identify resources, written in the left upper corner - ${}^r x$, set of resources is represented by R
- t - right bottom index represents time - x_t

Input

Input which is specified before executing optimization job by the user outside of the system. When the new execution job is requested - this is done by the user, or client application, the following data should be provided.

- D^j - maximal duration of the job execution which cannot be exceeded
- P^j - maximally used resources cost per job, or in other words highest possible price paid for the job execution which cannot be exceeded

There are also constants defined before load balancer startup.

- ${}^r c$ - the cost of the using particular resources per one time unit

Each of the previously mentioned variables must be non-negative.

Program Output

Apart from the result of the underlying optimization algorithm, following data are returned to the user after successful job execution.

- T^j - time taken, duration of the actual job execution
- C^j - resource costs, how much money was paid for the job execution

Variables

- v_t^j - value (i.e. cost of the scheduled plan) of the job j at the time t Value is greater than zero, and it is non-increasing during the time.

$$\forall t, j : v_t^j \geq v_{t+1}^j > 0 \tag{3.1}$$

  It is non-increasing because optimization, algorithms return always best found solution, so when worse solution, than currently best one, is found, returned is still the best solution found.

- ${}^r x_t^j$ - represents assignment of the resources r at time t to job j

$${}^r x_t^j = \{0, 1\} \tag{3.2}$$

  Each x is either 1 = indexed resources are assigned to the job at given time or 0 = given combination does not have assignment. We assume that each job has only one such assignment at one time, which effectively means that this job is executed on the single computation node. This is defined by the following constraint:

$$\forall j, t : \sum_{r \in R} {}^r x_t^j \leq 1 \tag{3.3}$$

- ${}^r \Delta_t^j$ - enhancement of the value v with resources r on the job j per time t.

$${}^r \Delta_t^j = {}^r |v_t^j - v_{t-1}^j| \cdot {}^r x_t^j \tag{3.4}$$

  It is improvement of the solution value v which can be achieved by using resources r at time t. This value is always non-negative since optimization algorithm always stores best found solution, and therefore $\forall j, t, r : {}^r \Delta_t^j \geq 0$. ${}^r x_t^j$ ensures that only resources, that are actually used, are taken in account.

- S_t^j - reward for improving solution value until time t per job j. Accumulation of enhancements ${}^r \Delta_t^j$ through all resources r and time units t.

$$S_t^j = \sum_t \sum_{r \in R} {}^r \Delta_t^j \tag{3.5}$$

- C_t^j - defines how much execution of job j cost from the beginning of the execution until time t. Sum of all allocated resources for their time for a particular job.

$$C_t^j = \sum_t \sum_{r \in R} {}^r c \cdot {}^r x_t^j \tag{3.6}$$

  Because C_t^j is defined as sum and ${}^r c$ is non-negative, it is true that $\forall j, t : C_{t+1}^j \geq C_t^j$. The input of the program specifies maximal cost paid

for job execution as a P^j, therefore it must be enforced by the system that this cost will not be exceeded. This constraint can be defined as follows.

$$\forall t, j : \; P^j \leq C_t^j \implies \sum_{t+1}^{\infty} \sum_{r \in R} {}^r x_t^j = 0 \tag{3.7}$$

Which effectively means that when cost of job execution C_t^j has reached maximal defined cost P^j, no resources can be assigned to this job.

- t - time, it is not only index but also variable, there are also constraints regarding time - since client application can specify deadline to job D^j, there must be an additional constraint for job execution in a matter of resources assignment.

$$\forall t, j : \; D^j \leq t \implies \sum_{t+1}^{\infty} \sum_{r \in R} {}^r x_t^j = 0 \tag{3.8}$$

When maximal time is exceeded, no additional resources can be assigned to the job execution, which could be defined by following constraint.

3.1.2 Resources reconfiguration

The system should be capable of changing resources assignment per job in the runtime. This will help to distribute performance according to the load of the current node across the whole network. Unfortunately, it is not always possible to reconfigure resources assignment while scheduling is being performed. Therefore there must be at least one-time unit, between different resources assignment, where no resources are assigned to the job. In other words, if resources reconfiguration is triggered in time t then $\sum_{r \in R} {}^r x_t^j = 0$. This can be formalized as:

$$\sum_{r \in R} {}^r x_t^j = 1 \implies \forall r \in R : \; {}^r x_t^j - {}^r x_{t+1}^j \geq 0 \tag{3.9}$$

3.1.3 Optimization criteria

The main goal of the system is to minimize the outcome value of the underlining optimization algorithm and at the same time to minimize the cost of used resources. We can optimize a single job or sum of outcomes from all jobs in the system at once. The first approach provides a possibility to control and optimize the outcome of a particular job, which is an advantage for a single client (job owner), but it does not necessarily mean that it is optimal for the whole system and vice versa. Optimization criteria for the single job at particular time t is then maximization of the weighted difference between the value enhancement reward and cost paid for the enhancement, which can be described by the following equation.

$$\max crit_t^j = \alpha S_t^j - (1 - \alpha) C_t^j \qquad 0 \leq \alpha \leq 1 \tag{3.10}$$

For optimization of system-wide resources and costs, all jobs execution optimization is then defined like a weighted sum of all rewards per jobs lowered by sum of all resources costs across the set of all jobs.

$$\max crit_t = \alpha \sum_{j \in J} S_t^j - (1 - \alpha) \sum_{j \in J} C_t^j \qquad 0 \leq \alpha \leq 1 \tag{3.11}$$

Based on the previous equation, it is possible to define time independent optimization criterion.

$$\max crit = \sum_{t}^{\infty} crit_t \tag{3.12}$$

3.2 Resulting problem

Proposed formal definition (3.1) implies that problem is integer linear problem with job execution cost as its main optimization criteria. These problems are solvable by various types of mixed integer linear programming solvers. Linear optimization and its characteristics are outlined in previous section 2.2.1.

However, apart from load balancing decisions, the problem brings another challenge in time series prediction during the job executions. This problem arises because algorithm, by definition presented in section 3.1, estimates enhancement of the job's solution value when the job uses particular resources in the next period. In other words, the algorithm needs to estimate the impact of using particular resources in the next period on solution value function development. Therefore in time t it needs to estimate value of the solution value function v_{t+1}^j. The definition in section 3.1 describes it as ${}^r\Delta_t^j$ variable. Based on the ${}^r\Delta_t^j$ value, the reward for improving solution value S_t^j is computed. This reward is then used when the load balancing decisions are being made to reflect the suitability of the current execution plan solution.

Chapter 4

Solution design

The presented problem can be solved using many possible approaches, We have decided to use mathematic optimization for running algorithms values predictions and heuristic algorithm for load balancing decisions.

4.1 Solution value prediction

To have the most informed decisions while creating load balancing decisions, algorithm creating these decisions needs to be able to estimate the impact of assigned resources to the development of the solution value of the job. Therefore in time t it needs to estimate value of the solution value function v_{t+1}^{j}. The definition in section 3.1 describes this estimated impact as ${}^{r}\Delta_{t}^{j}$ variable. The reward for improving solution value S_{t}^{j} is then computed based on this ${}^{r}\Delta_{t}^{j}$ value. This reward is then used when the load balancing decisions are being made to reflect the suitability of the current execution plan solution.

Unfortunately, it is practically impossible to predict the values exactly. For instance, heuristics and metaheuristics optimization algorithms are usually stochastic and therefore there is **no guarantee** that they eventually find a better solution than the current one. That said, the only thing that is guaranteed is that the solution value function of the job overtime period is monotonically non-increasing. Empirically, the convergence of the solution value towards an optimal solution has hyperbolic shape. This assumption was made after the analysis of the runtime solution value data from the optimization algorithms described in section 6.2. In addition, the script, which is able to visualize, the solution value function during the time was developed. This Python script is part of the implementation and can be found in the `scripts` folder inside the root of the project.

Moreover, it is not possible to use the time as x axis, because it would not be possible to extract information about how adding more available performance will modify the predicted solution value function of the job. For

that reason, it is better to use the number of iterations, that optimization algorithm performed, instead of the time unit.

4.1.1 Hyperbola time series fitting

The generic equation for expressing the hyperbola function on the two-dimensional graph is the following.

$$a + \frac{b}{x + c} = y \tag{4.1}$$

Where a, b, c are parameters and x, y are axis values. For usage in a computer algorithm, it is better to transform the equation into the form, where there is no division. Apart from the better performance in favor of multiplication [14], it is possible that the state when $x = -c$ occurs. If there is the division, the resulting value will be infinity, which breaks the next algorithm iterations.

Instead, it is better to use the following form, where there is no division and 0, as a result, is not a problem for the following algorithm iterations.

$$ax + ac + b - yc = yx \tag{4.2}$$

Since the solution value prediction should be computed as fast as possible, it was decided to use the mathematics optimization approach transforming the problem into non-linear least squares problem.

Non-linear least squares problems are often solved by the algorithms based on iterative estimate improvement. The example of these algorithms is the Gauss-Newton algorithm [15] and the Levenberg–Marquardt algorithm [16]. The second mentioned non-linear algorithm is more robust than the Gauss-Newton, because of the Marquardt parameter [16], which means that in many cases it finds a solution even if it starts very far off the final minimum.

Using the Levenberg–Marquardt algorithm means, that it is necessary to know the derivation of the function, the algorithm is trying to fit in. Derivation of the used function 4.2 is then following.

$$f'(x) = (c + x, 1, a - y) \tag{4.3}$$

As the target values, are used $x \cdot y$. As the parameters, that Levenberg–Marquardt algorithm is trying to fit in are used a, b, c.

4.2 Load balancing decisions

To make the most informed load balancing decisions while scheduling multiple optimization jobs, the application uses dynamic scheduling with a centralized node running the load balancing algorithm.

The application also takes advantage of knowing the exact maximal time of an execution thanks to the input parameter D^j defined in section 3.1.1. Also, because of this parameter, it is possible to create scheduling decisions for a much larger time horizon, because the algorithm is aware of the future workload.

The definition formalized in section 3.1 implies that it is possible to use integer mixed integer linear programming solver, because the problem itself is defined as an integer linear programming problem. That is indeed possible, but after careful consideration, we rather decided to use a heuristic approach. The main reason for selecting this type of optimization algorithm is, that it provides suitable results during the whole runtime. This could be very handy when the time for load balancing decisions is tight and in such case, the mixed integer linear programming solver would not have enough time to provide a suitable solution, because the one, it provided would not be optimal at all.

The optimization solver, which was chosen for the application, was previously mentioned (2.2.2) heuristics optimization engine OptaPlanner. Unlike TASP (2.2.2), the OptaPlanner is open sourced. The implementation based on OptaPlanner is described in section 5.3.

4.3 Complete application algorithm

The load balancing algorithm phases are divided into scheduling rounds. Each scheduling round produces a new execution plan. The execution plan is created for a specified scheduling horizon, this horizon is the period, which is limited by the time $t+1$ and the t_{max}, which defines the end of the execution plan and where the t is the time of execution. In other words, each scheduling round, executed in time t, produces the scheduling plan with scheduling horizon from $t+1$ to t_{max}. The scheduling horizon can be configured in the scheduling properties described in chapter 5. The length of the scheduling window can be configured as well.

The formalized algorithm is then presented in the algorithm 1. The job related properties D, T, P and C are the very same variables defined in the section 3.1. The functions mentioned in the algorithm are described in the algorithm detailed description listed below the algorithm itself. The function *predict* is related to the step described in the step 4 and the function *schedule* to the step 5.

Input: Q - queue with jobs to schedule
1 Q: jobs queue;
2 J: set of jobs;
3 P: predictions;
4 E: execution plan;
5 $J \leftarrow Q$.poll();
6 $J' \leftarrow J$.filter($job \rightarrow job.D > job.T \wedge job.P > job.C$);
7 $J'' \leftarrow$ convert(J');
8 $P \leftarrow$ predict(J'');
9 $E \leftarrow$ schedule(J'', P);
10 $E' \leftarrow$ convert(E);
Result: Q is empty
Output: E' - execution plan

Algorithm 1: Load balancing algorithm

1. Poll jobs from the jobs queue.
   - The incoming jobs from API are stored in the queue, until the new scheduling round is engaged.
2. Analyze and filter jobs, that are not relevant for the current scheduling round.
   - Filtered out are the jobs, whose execution time exceeded the maximal possible execution time D^j or whose cost has exceeded the maximal cost P^j.
3. Convert received jobs into the inner data representation.
   - The data must be converted from the data transfer objects structure to the OptaPlanner domain representation.
4. Create predictions based on the solution value of the jobs and on the history of the load balancing decisions.
   - The process of making the predictions is described in section 4.1.
5. Produce execution plan using OptaPlanner scheduling.
   - The scheduling is limited to the amount of time defined as scheduling window. When the scheduling reaches the scheduling window (for example 60 seconds), it is stopped.
6. Convert created plan into time schedule and send them back to the client.
7. Engage the next scheduling round and go to the step 1.

Chapter 5

Implementation

In this chapter, we would like to present the technologies that were used while implementing the previously described load balancer. During the development, base package of the application was named `OLB`, which is an acronym for **O**ptimization **L**oad **B**alancer. In the following pages, the developed application is called this way.

Although this paper goals aim solely on the scheduler and introducing the new load balancing strategy, the architecture keeps in mind the future work outlined in section 7.1. Therefore the system's architecture is based on the idea of cooperating microservices, where each service has control over a specific part of the infrastructure.

Microservice architecture is a software design architectural style that structures an application as a collection of loosely coupled services that are organized around the system's business capabilities [17] and are independently deployable with enabled continuous delivery [18]. Microservice architectural design also helps the system's better horizontal scalability by using multiple instances of one microservice and the orchestration module.

5.0.1 Architecture scheme

Scheme 5.1 visualizes only system's core architecture. However, the whole design keeps in mind future infrastructure development proposed in section 7.1. The implementation itself was developed accordingly and used technologies and techniques are described in the following sections.

Core component, which is responsible for scheduling and load balancing, is composed of API, converter, predictions, and scheduler module.

- *API module* implements common core interface `OlbCoreApi` and is responsible for scheduling requests handling. Actual implementation of API interface is class `OlbCoreApiImpl`.

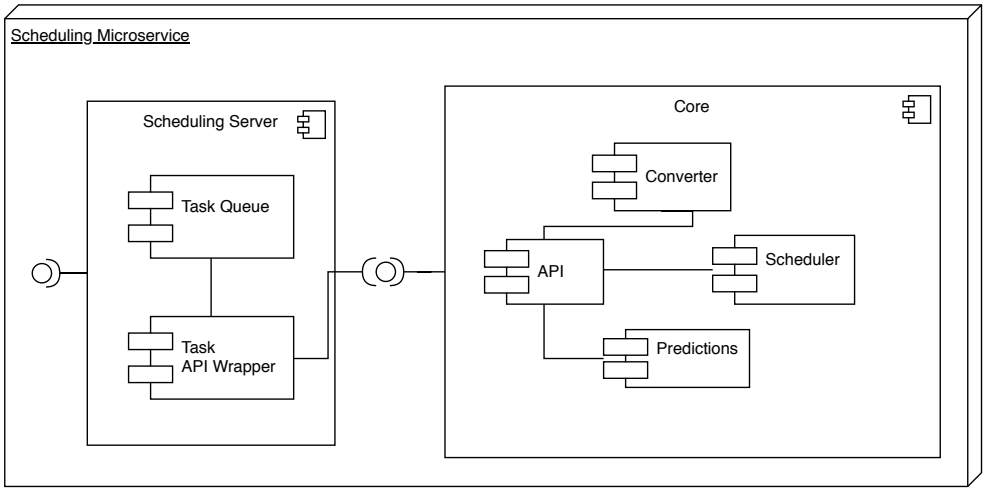

Figure 5.1: Microservice architecture with scheduling core

- *Converter module* is used to convert received input data into scheduler's inner data representation and then back to common data transfer objects. This converter is implemented as a class `InputToDomainConverter`.

- *Scheduler module* contains constraints, evaluator and scheduling system based on OptaPlanner solution (described in section 5.3). Scheduler module consist of multiple packages, `constraints` - containing all constrains for scheduling algorithm, `domain` with defined scheduling domain for OptaPlanner, `evaluation` which includes evaluator calculating planning score and `solver` package with factory initializing OptaPlanner scheduling core.

Scheduling server provides HTTP API access to the core and serves as the microservice base. The server module is based on Ktor framework (described in section 5.1.4) that runs under the hood and provides HTTP functionality., The server API uses binary serialization and accepts data transfer objects used in the solutions. This serialization approach was chosen because of the number of interfaces and loosely coupled data objects, that are being used in the application.

Simulations architecture

Simulations module (project module `simulations`) is designed as another microservice to simulate future load balancing system's behavior. Following figure 5.2 shows the architecture of the module.

Input data are generated by the input generator module, which can be found, for example in `DomainBuilder` class. Algorithm values are parsed

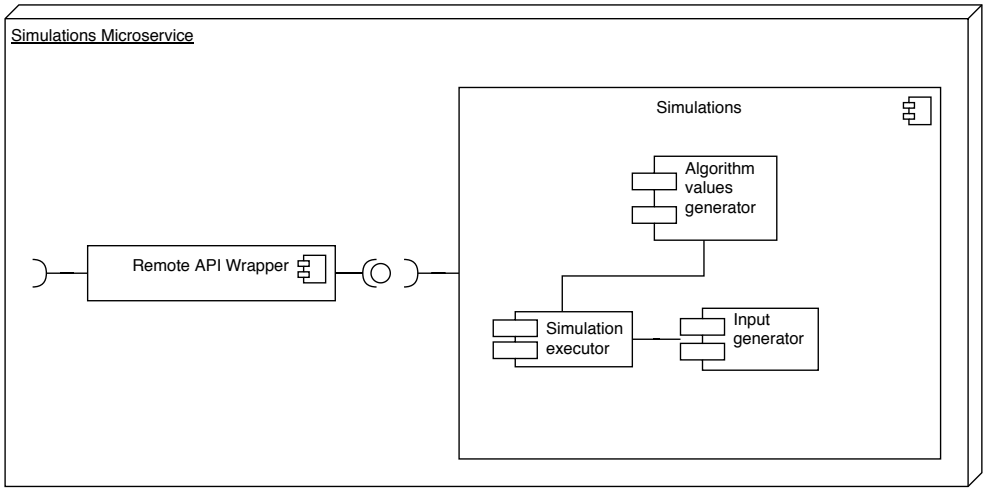

Figure 5.2: Simulations module scheme

from the file in class `DataParser` and the corresponding model is created on top of them in `JobWithHistoryFactory`.

Remote API wrapper is used to create a connection between remote microservice, with scheduling API, and simulation. For the simulation module, the connection seems to be synchronous. It is because there is a blocking queue used to store and answer the calls between these two microservices. Thanks to this solution, simulations can be run in a microservices mode as well as locally without any additional effort needed. Remote scheduling solution is implemented in the project module `remote−scheduler`. Local simulations can be found for example in `OnePlanningRoundMain` or `ExecutionConfiguration` classes.

5.1 Development stack

The development stack is based on the Java platform, targeting primarily JVM 11[9] but including backwards compatibility with JVM 8. However, the traditional Java platform programming language Java was not used.

5.1.1 Programming Language

The OLB[10] is not bound to the single technology, which could limit the development stack and for that reason, we had a free choice while choosing the programming language used for the OLB implementation.

[9] Java Virtual Machine - runtime environment for Java byte code
[10] Optimization Load Balancer, name of the application

OLB is programmed in the programming language **Kotlin**. This cross-platform, statically typed, the general-purpose programming language is developed by JetBrains [19]. Kotlin is 100% interoperable with Java because it uses JVM as its runtime and it is compiled to the Java Bytecode. Apart from the Java Bytecode, it can also be compiled to JavaScript or native code. [19] The main advantage of Kotlin is its aggressive and robust type inference, meaning that for the most of the time, it is not necessary to specify used data type since Kotlin compiler can infer it from the context. [19] It results in concise language syntax and therefore to the faster development in general.

Another great advantage is Kotlin's *null safety*. Kotlin compiler distinguishes between non-nullable types and nullable types and enforces *null checks*[11] when the object has a nullable data type. This feature effectively leads to fewer problems in the code and drastically reduces *Null Pointer Exceptions*[12] during the runtime.

5.1.2 Build environment

The application uses Gradle as its build automation system. It was chosen mainly because its incrementally build system, that works by tracking input and output of tasks, including files changes tracking, and only running tasks, that are necessary and thus reducing the required time to build the project. Also, it processes only these files that were changed between tasks execution. Another reason we choose Gradle was that it is preferred build system for Kotlin.

The preferred approach to assemble the application is to use Docker and build it as the Docker image, which can be then run inside the Docker container.

Docker build environment

To keep the build clean and reusable on almost every operating system and machine setup we decided to use **multistage Docker build**[13] which uses different base docker images for the build and the run phase. Since OLB targets the JVM 11 environment and uses Gradle as its build system, *gradle:5.4.0-jdk11-slim* is used as base image for build stage. This image contains all necessary Gradle build tools while having a smaller size than the common Gradle Docker image. Even smaller (in terms of size) are *alpine* based docker images. Alpine is the smallest possible Linux core, which is widely used in the full range of Docker base images. Alpine is focused on the smallest possible size of the image, while having all the necessary tools built

[11] Check whether the object being used has not null value

[12] Exception raised when code access reference that has null value

[13] Docker is a technology that performs operating-system-level virtualization, meaning that it uses hosts operating system kernel

in. Unfortunately, there were (at the time of development) no official JVM 11 alpine images since there is no official stable OpenJDK[14] 11 build for Alpine Linux.

5.1.3 Runtime environment

The preferred runtime environment is a Docker system, where the application image runs inside the created docker container.

Docker runtime environment

The build application files are copied from the Docker build stage to the Docker runtime stage. As the runtime base image in the multistage build was used *openjdk:11-jre-slim* image, because it is an official OpenJDK 11 Docker image and therefore it is declared as stable.

Because there was used *gradle application plugin* while building the application, startup scripts were generated by the Gradle. These scripts are then used to start the application itself inside the Docker container.

When starting the whole application, multiple services must be started up. Therefore, because of the containerized environment, where containers can not access each other, multiple containers must be started, and the virtual network connecting them must be created. This process can be automated using Docker Compose.

Docker Compose

Docker Compose [20] is an application for defining, running and managing multi-container Docker applications. It automatically creates Docker networks as well as Docker volumes. With writing down the definition of multiple Docker applications to the one Docker Compose configuration file, it is possible to create robust microservices architecture, which can be built or started using a single command.

Thanks to the created Docker networks, containers can communicate with each other using Docker Compose service names, therefore they do not need to know specific IP address they have.

Docker Compose is used in the implementation of OLB since it is designed with microservices architecture in mind. There are two services - Scheduling server and Scheduling client. Scheduling server provides the ability to schedule process execution on the various computers and contains all core algorithms. Scheduling client is an example application which uses the ability of scheduling server. There are implemented various simulations, which are being executed by scheduling client.

[14] Open-source implementation of the Java Platform, Standard Edition

5.1.4 Framework

Because of the overall microservices architecture of the project, a web framework was needed. There are many Java-based web frameworks that could be used. We would like to present two of them - *Spring Boot*, which is a traditional and widely used web framework for all kind of Java web applications and *Ktor* - relatively new, lightweight Kotlin framework build upon the Kotlin Coroutines[15].

Spring Boot

Spring Boot [21] is an open source Java Spring-powered web framework. It takes an opinionated view of the Spring platform, meaning that Spring Boot automatically configure Spring and 3rd party libraries whenever possible, and therefore enables usage of it to broader audience. It is highly dependent on the starter templates feature which provides pre-configured templates for various types of web applications. This, for example, allows the user to start with already working web server and thus simplify the start of the application development [22]. Spring Boot contains comprehensive infrastructure support for developing enterprise monolith web applications as well as micro services [22].

Ktor

Ktor [23] is an open source web framework for building asynchronous servers and clients in connected systems such as web applications and HTTP services. It designed for quickly building web applications with minimal effort and it doesn't impose a lot of technical constraints such as logging, persistent, serializing, dependency injection etc. [24] It is developed by the same company as Kotlin is, JetBrains.

The final decision was to use **Ktor** as the web framework, mainly because of its very light implementation and native Kotlin support. Also, for such a project, the features of Spring would not be fully utilized and therefore, the complexity of Spring could potentially slow down the entire application.

Because Ktor by default does not contain any dependency an injection framework, We decided to use lightweight DI[16] framework *Koin* [25]. This framework is written in Kotlin and have its own DSL[17] for the dependency specification, which is very handy for the medium-sized project.

[15] Way of asynchronous or non-blocking programming that generalize subroutines for non-preemptive multitasking with using suspended/resumed task execution

[16] Dependency Injection

[17] Domain Specific Language

Route discovery library

The default way, how to create a HTTP endpoint (Ktor calls them *routes*), which can handle HTTP requests is registering it within the `Application` context. The `Application` context is accessible by its instance that is given to the user when the Ktor is being started. This means that no *route* can be registered without using an instance of `Application`.

During the development of the application and using Ktor framework, we decided that the proprietary way of registering routes was not something we would like to be using, mainly because it did not allow to have pure class serving only as a route without having to inject the *Application* instance. Also, since the routes must be registered during the application startup, using the new class for each route would mean to create an instance of the class and executing the method to register the routes manually. Another reason we did not like the Ktor default approach was that we prefer to inject class dependencies using the construct injection instead of using the setters injection.

- Constructor dependency injection - using the constructor of the class to set all instances of the objects, that class uses. The main advantage is that the instance of the class is always in a valid state because it has all dependencies resolved during the instance creation.
- Setter dependency injection - the dependent objects are provided by the setter methods. This gives the freedom to manipulate the state of the dependency references at any time. However, it is possible to use the instance without setting the dependencies which could lead to the undefined behavior or the *Null Pointer Exceptions*

To solve this `Application` instance dependency and to enable constructor dependency injection approach, we decided to implement a simple library which would solve this issue for us. We came up with a different way how to register various types of application's routes using the annotations, reflection and dependency injection strategy.

Preconditions for successful usage of the library are the following:

- *Koin* [25] - dependency injection framework which is used for resolving dependencies needed in the routes
- *Reflection library* [26] - library used for runtime lookup for classes with specific annotation
- Using the `@Route` annotation on the class that is meant to be route, the class has to also inherit from `RouteBase`
- Registering all necessary routes dependencies in *Koin* modules during the application startup

- Provide base package name where routes are placed.

Following algorithm is used to find and register all routes used in the project.

Input: Package name, where all routes are stored
1 **routes** ← find all classes annotated as `@Route` in provided package name;
2 **for** ***route*** *in* ***routes*** **do**
3 | **dependencies** ← obtain dependencies needed for creating instance of **route**;
4 | **routeInstance** ← create instance of **route** using **dependencies**;
5 | register **routeInstance** in instance of `Application` ;
6 **end**
Output: All routes are registered and ready to use

The implementation of the simple route is then following:

```
@Route
class HelloRoute(sr: Service) : RouteBase("hello") {
    init {
        route {
            get {
                call.respond(sr.sayHello())
            }
        }
    }
}
```

The route is automatically instantiated and registered by the Routes discovery library. The programmer does not need to handle it by himself.

For the library startup, we designed a builder class using fluent builder pattern `ApplicationDependencyBuilder`. Usage of this class can be found in the `ServerStarter.kt`.

5.2 Algorithms values prediction

The implementation of the Levenberg–Marquardt algorithm is not the aim of this paper, for that reason, the application uses Java compatible library, which contains implementation Levenberg–Marquardt algorithm and provides the way, how to use it.

OLB uses two different libraries Apache Commons Mathematics Library [27] and Mathematical Finance Library [28]. Both libraries have custom wrapper which implements `HyperbolicRegression`, abstract class and they can be exchanged in the application when decided. The reason, there are two different implementations and two different libraries, is that their performance

can differ in distinguish scenarios. By default, the Mathematical Finance Library is used, because it provides better results in runtime predictions[18]. The implementations can be found as `ApacheHyperbolicRegression` class for Apache Commons Mathematics Library and `FinMathRegression` class for Mathematical Finance Library.

5.3 Load balancing decisions with OptaPlanner

Scheduling system implementation uses OptaPlanner library core in version `7.19.0.Final` [29] and it is crucial part of the `core` module and application itself.

5.3.1 Formalized definition representation

In this section, the thesis describes mapping between formalized problem definition proposed in section 3.1 and actual implementation in the code.

Each job, which is indexed in definition as j, implements `Job` interface and the main implementation, used in the core while scheduling, is the `PlanningJob` class. Input variables D^j and P^j are then specified as a `JobParameters` property of the `Job` interface.

Resources r are implemented as a sealed class `Resources` composing of `CpuResources` and `MemoryResources`. Resources belong to resources pools, which can be imagined as physical computers or a virtual machines. Resource pools then carry information about the cost ${}^{r}c$ of the underlying resources. The pools are represented as a `ResourcesPool` interface.

Class `JobValue` represents a solution value of the job during time v_t^j and can be found as a property in the interface `JobWithHistory`. A job, that has some historical information (like a solution value of the job during the time and the scheduling data). Solution value of the job related information uses `AssignmentsEvaluation` during scheduling to compute reward for scheduler with value S_t^j. Resources cost C_t^j value is used in the cost constraint `CostEvaluation` for comparison with P^j. In the similar constraint `TimeEvaluation`, t value is compared with D^j to check, whether all specified constraints are satisfied.

Scheduling output is always `AllocationPlan`. It contains job domain, resources domain, the overall cost and created time schedule. From the latest, values T^j and C^j can be computed very easily, therefore are not present directly in the interface as properties.

[18] There is a Python code, which can provide graphs from values predictions, please see `plot_predictions.py` and `pw.forst.olb.simulation.prediction` package.

5.3.2 Scheduling Algorithm

The OptaPlanner scheduling itself has two main phases. *Construction heuristics* that tries to build an initial solution in a finite length of time. This partial solution is not always feasible, but it is found in a relatively short time, and then it is passed to the next scheduling phase. *Local search* with metaheuristics that can enhance the partial solution found in the previous phase.

OptaPlanner contains various types of construction heuristics (i.e. *first fit*, *weakest fit* and *strongest fit*) as well as local search metaheuristics such as *hill climbing*, *tabu search* and *simulated annealing*. As the best combination of construction heuristics and local search proved to be *first fit* with *tabu search*.

The First Fit algorithm cycles through all the planning entities, initializing one planning entity at a time. It assigns the planning entity to the best available planning value, taking the already initialized planning entities into account. It terminates when all entities have been initialized [30].

The Tabu Search metaheuristics search is based on the local search optimization method and enhances it by worse step strategy, when at each step worsening moves can be accepted if no improving move is available. Besides, prohibitions are introduced to discourage the search from coming back to previously-visited solutions [31].

5.3.3 Implementation

The scheduling implementation can be found in module `core`. The constraints are placed in the package `pw.forst.olb.core.constraints`. OptaPlanner related implementation is placed inside the `pw.forst.olb.core.domain` package and the plan solution evaluator in `pw.forst.olb.core.evaluation` package.

All custom constraints should implement `CompletePlanEvaluation` or `PlanEvaluation` interface. That way, they can be used in custom score evaluator, which is then used to generate the score of the given plan. The constraints stored in collections of previously mentioned interfaces and evaluated at once, when the plan is submitted. Thanks to this solution, additional constraints can be added or removed very easily.

As an input for the scheduling algorithm interface `SchedulingProperties` is used. This interface defines properties necessary for the scheduling infrastructure such as `maxTimePlanningSpend` which explains how long can the application run the scheduling or `cores`, describing how many cores in the computer can algorithm utilize by spawning scheduling threads.

The OptaPlanner scheduler instance is created by custom factory implementation `OptaPlannerSolverFactory` and uses base configuration defined in `solverConfiguration.xml` file.

Chapter 6

Experiments

In this chapter, we would like to present, how we tested the load balancer. The simulations were designed to reflect the real-life situations and to simulate the most common usage of the load balancer.

6.1 Simulations implementation

To test the load balancer, there are two modules, that contain simulation code used for testing - `simulation` and `remote−scheduler` module. The simulation engine and all simulation use cases are located in the simulation module. The second mentioned module contains the server, which runs simulations on the remote API. This is especially useful when testing the whole scheduling environment and it is as closest as possible to the real-life environment, which should be based on the microservices architecture.

Both simulation can be started up locally using `SimulationExecutor` or using `docker−compose` inside the microservices runtime environment. Input data for the tests are created randomly, based on the number of scheduled jobs and given planning horizon (how much steps ahead should scheduler count with). The number of scheduled jobs can be easily edited directly in code. For the simulation data and the input configuration please refer to `OnePlanningRoundSimulation` and `RuntimeSimulation` classes.

6.2 Optimization algorithms data

For the proper testing environment, the real runtime data of the optimization algorithm were needed. We decided to use TASP (mentioned in subsection 2.2.2) as heuristic algorithm, which execution could be potentially scheduled by the instance of the optimization load balancer. We did not implement a new TASP instance, instead, we used examples from Stochastic Dynamic Vehicle Routing Problem master's thesis by Petr Eichler [32].

These instances solve the real-life vehicle routing problem and mainly for that reason are ideal for the testing purposes. We slightly modified the code from the thesis for observation purposes and added the time measuring functionality, which tracked the time between the algorithm's iterations and the current solution value of the job in each iteration.

In overall, we executed and measured 56 different instances of TASP. The observations can be found in `jobs-data/input` folder inside the implementation project and they are being used in the simulations, where the simulation module randomly selects one file with runtime data for each job and assigns it to the job, that is being scheduled. The data are then effectively used as input data for the load balancer.

6.3 Simulations

There are two main randomized simulation scenarios inside the simulation module (module described in 5.0.1). The first simulation is designed to be static. It creates only one execution plan and then it shut downs. The second simulation reflects the presumed production environment and is described in figure 6.1.

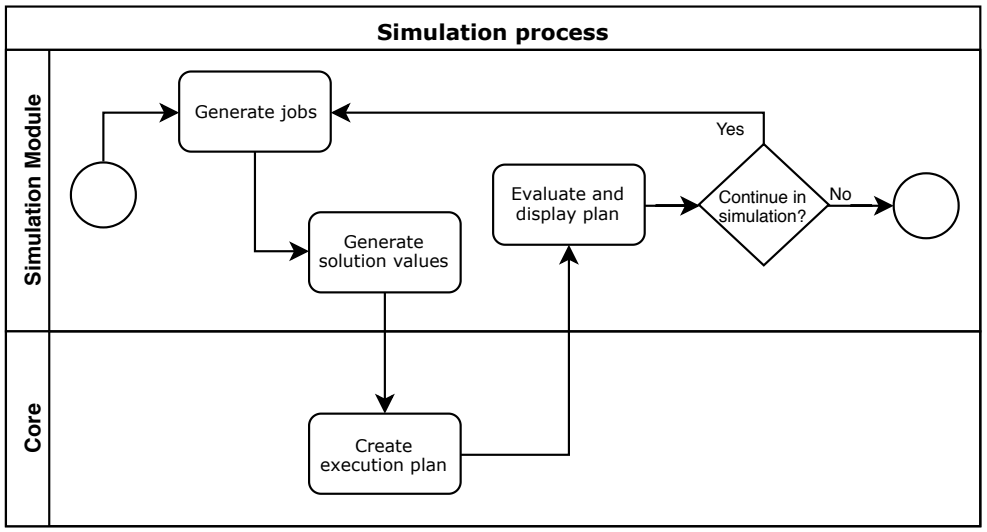

Figure 6.1: Simulation process

- **Generate jobs** - simulation randomly create new optimization jobs, which should be scheduled by the core. It also generates their scheduling parameters.
- **Generate solution values** - simulation uses the solution values functions, generated by the TASP (2.2.2) algorithm as described in section

6.2. The data files are randomly assigned to the particular jobs, therefore each job represents unique TASP instance and has unique solution value during time.

- **Create execution plan** - simulation executes plan creation by calling the core API. The process of plan creation is described in section 4.3.
- **Evaluate and display plan** - simulation engine uses evaluator to check for the constraint violations and then prints the result into the log. It also prints the text representation of the plan into the standard output. An example of such plan in text representation can be seen in listing **??**.

6.3.1 Simulation output

Simulation's output (produced execution plan) is printed to standard output. Data displayed in the table 6.2 are a formalized output produced by the simulation with ten jobs being scheduled at once. The optimization jobs, which were scheduled in the table 6.2, have their parameters displayed in the table 6.1. The parameters can be visualized by the application as well, please refer to the file `AllocationsPlanExtensions.kt`.

Table 6.1: Table with the input and output parameters of the jobs

j	0	1	2	3	4	5	6	7	8	9
D^j	638	650	373	371	519	624	621	407	322	725
P^j	75	52	96	51	34	144	103	46	18	29
T^j	600	540	120	360	460	600	420	300	120	540
C^j	27.4	40.84	29.6	26.54	19.8	30.54	15.74	27.78	14.8	19.3

The variables D, P, T and C are defined in section 3.1 and represents following data related to the one job.

- D^j - maximal duration of the job execution which cannot be exceeded
- P^j - maximal used resources cost per job, or in other words highest possible price paid for the job execution which cannot be exceeded
- T^j - time taken, duration of the actual job execution
- C^j - resource costs, how much money was actually paid for the job execution

The following data output in the table 6.2 is the result of the first scheduling window (how does the load balancing algorithm work is described in section 4.3).

Time axis shows time units (t value described in section 3.1). Second, vertical axis visualizes resources. r value is in the format $x.y$ where the x

Table 6.2: Simulation data output

t:	0	1	2	3	4	5	6	7	8	9	10
${}^{1.1}x_t^j$	1	1	2	2	2	1	5	5	1	1	0
${}^{1.2}x_t^j$	9	4	6	2	3	3	7	5	8	9	8
${}^{2.1}x_t^j$	0	0	5	0	7	0	4	4	6	4	3
${}^{2.2}x_t^j$	5	7	7	9	7	9	9	4	6	6	5

value is the resources provider (explained in section 5.3.1), and y is one usage unit - meaning that it could be one physical core or for example percentage of the shared processor. In the implementation, this value is called CPU core, but in fact, it is dimensionless value expressing usage of some system resources.

Each cell contains either number or is empty. The number is job ID and indicates that this resource is allocated to the job with displayed ID. This is effectively ${}^{r}x_t^j$ value explained in section 3.1.1.

Total plan cost is the sum of all particular costs of all jobs - their resources allocations. Therefore this is the cost of the created execution plan.

Chapter 7

Conclusion

All goals of this thesis were fulfilled. This thesis analyzed the state-of-the-art solution for the load balancing systems in section 2.1. It brought an overview of various optimization problems, techniques, and algorithms that are being used to solve diverse problems in section 2.2.

The thesis described and formalized the problem of the load balancing of optimization algorithms in chapter 3. The solution to the formalized problem was then proposed in chapter 4. The main goal of the thesis was then fulfilled in the following chapter 5, where the implementation of the new system was described. This implementation is attached to the thesis.

The solution with 6767 lines of Kotlin code[19] proved, that it is possible to use it as a load balancing algorithm for optimization algorithms.

7.1 Future work

To achieve production ready load balancing system, an execution module, which would transform the created plan into physical actions such as running, stopping and moving the jobs between the physical execution nodes, is needed.

Infrastructure development

As soon as the previously mentioned execution module is implemented, the application can be deployed into a real-life environment. The proposition is to make optimization algorithms running as Docker containers and the execution module would operate with Docker machines, which would be running on the execution nodes. In this way, the assigned resources would be very easily changed on one execution node. Migration to the next execution node would not be a problem as well, since the containers could be wrapped and transported through the network.

[19] Lines provided by `find . -name "*.kt"| xargs cat | wc -l` bash command in the root folder of the project

Another infrastructure related feature is the full JSON-enabled REST API. Currently, only binary serialization is supported. In the future, full REST with JSON as its transport format should be supported.

Routes discovery library

Routes discovery library was very handy during the development of the application's server parts and we believe that this way of routes registration in Ktor would suit to many developers as well.

Therefore we would like to refactor it from the base project and create an open source project which will ensure future library development. We would like to also make it more generic, because right now, it depends on specific Ktor and Koin version. Although Ktor dependency is necessary since it is library developed specifically for Ktor, Koin should be replaced by a generic way, how to obtain dependencies for routes.

Extension functions

During the application development, we created and tested many Kotlin extension functions. In general, these functions are not domain specific and for that reason, we decided that we will publish them as well as Routes discovery library. These extensions could be useful when starting a new project, because they are able to perform many operations in the single line of Kotlin code.

Bibliography

[1] B. Marr, "What is industry 4.0?." https://www.forbes.com/sites/bernardmarr/2018/09/02/what-is-industry-4-0-heres-a-super-easy-explanation-for-anyone/. [Online; accessed 18-May-2019].

[2] A. Networks, "Hardware load balancer." https://avinetworks.com/glossary/hardware-load-balancer/. [Online; accessed 30-January-2019].

[3] D. S. K. Ramesh Prajapati, Dushyantsinh Rathod, "Comparison of static and dynamic load balancing in grid computing," *International Journal For Technological Research In Engineering*, 2015.

[4] IBM, "Algorithms for making load-balancing decisions." https://www.ibm.com/support/knowledgecenter/SS9H2Y_7.7.0/com.ibm.dp.doc/lbg_algorithms.html. [Online; accessed 29-January-2019].

[5] A. G. Payal Beniwal, "A comparative study of static and dynamic load balancing algorithms," *International Journal of Advance Research in Computer Science and Management Studies*. [Online; accessed 29-January-2019].

[6] Malik, Shahzad, "Dynamic load balancing in a network of workstations," *95.515 F Research Report*, 2000.

[7] S. Sharma, S. Singh, and M. Sharma, "Performance analysis of load balancing algorithms," *World Academy of Science, Engineering and Technology*, vol. 38, no. 3, pp. 269–272, 2008.

[8] A. Makhorin, "Glpk (gnu linear programming kit)." https://www.gnu.org/software/glpk/. [Online; accessed 16-January-2019].

[9] Google, "About or-tools." https://developers.google.com/optimization/. [Online; accessed 16-January-2019].

[10] Papanikolaou, A., *A Holistic Approach to Ship Design: Volume 1: Optimisation of Ship Design and Operation for Life Cycle*. Springer International Publishing, 2018. 296-301.

[11] "Blindspot solutions s.r.o.." http://www.blindspot.ai.

[12] "Amazon web services." https://aws.amazon.com/. [Online; accessed 7-May-2019].

[13] "Microsoft cloud computing services." https://azure.microsoft.com/. [Online; accessed 7-May-2019].

[14] J.-A. LeFevre and J. Morris, "More on the relation between division and multiplication in simple arithmetic: Evidence for mediation of division solutions via multiplication," *Memory & Cognition*, vol. 27, pp. 803–812, Sep 1999.

[15] S. Gratton, A. S. Lawless, and N. K. Nichols, "Approximate gauss–newton methods for nonlinear least squares problems," *SIAM Journal on Optimization*, vol. 18, no. 1, pp. 106–132, 2007.

[16] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *Journal of the society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963.

[17] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture," *International Journal of Open Information Technologies*, vol. 2, no. 9, pp. 24–27, 2014.

[18] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[19] "Kotlin reference." https://kotlinlang.org/docs/reference/. [Online; accessed 7-May-2019].

[20] "Docker compose reference." https://docs.docker.com/compose/overview/. [Online; accessed 7-May-2019].

[21] "Spring boot reference." https://spring.io/projects/spring-boot/. [Online; accessed 7-May-2019].

[22] "Spring boot github reference." https://github.com/spring-projects/spring-boot/tree/5aeb31700df8e15d1aa625b987ecca93385a7c2c. [Online; accessed 7-May-2019].

[23] "Ktor web." https://ktor.io/. [Online; accessed 7-May-2019].

[24] "Spring boot github reference." https://api.ktor.io/1.1.5/. [Online; accessed 7-May-2019].

[25] "Koin github repository." https://github.com/InsertKoinIO/koin/tree/50929af636d1956a45882b795a29ace00eeac49d. [Online; accessed 7-May-2019].

[26] "Reflection library github repository." https://github.com/ronmamo/reflections/tree/084cf4a759a06d88e88753ac00397478c2e0ed52. [Online; accessed 7-May-2019].

[27] "Apache commons mathematics library." https://commons.apache.org/proper/commons-math.

[28] "Mathematical finance library." https://github.com/finmath/finmath-lib.

[29] "Optaplanner documentation." https://docs.optaplanner.org/7.6.0.Final/optaplanner-docs/html_single/. [Online; accessed 11-May-2019].

[30] "Construction heuristics." https://docs.optaplanner.org/7.6.0.Final/optaplanner-docs/html_single/#constructionHeuristics. [Online; accessed 11-May-2019].

[31] F. Glover, "Tabu search—part i," *ORSA Journal on computing*, vol. 1, no. 3, pp. 190–206, 1989.

[32] P. Eichler, "Stochastic dynamic vehicle routing problem," Master's thesis, Czech Technical University in Prague, Czech Republic, 2018.

Appendix A

List of attachments

1. CD with implementation described in chapter 5